What the experts have to say about Model-Based Testing for Embedded Systems

“This book is exactly what is needed at the exact right time in this fast-growing area. From its beginnings over 10 years ago of deriving tests from UML statecharts, model-based testing has matured into a topic with both breadth and depth. Testing embedded systems is a natural application of MBT, and this book hits the nail exactly on the head. Numerous topics are presented clearly, thoroughly, and concisely in this cutting-edge book. The authors are world-class leading experts in this area and teach us well-used and validated techniques, along with new ideas for solving hard problems.

“It is rare that a book can take recent research advances and present them in a form ready for practical use, but this book accomplishes that and more. I am anxious to recommend this in my consulting and to teach a new class to my students.”

—DR. JEFF OFFUTT, Professor of Software Engineering, George Mason University, Fairfax, Virginia, USA

“This handbook is the best resource I am aware of on the automated testing of embedded systems. It is thorough, comprehensive, and authoritative. It covers all important technical and scientific aspects but also provides highly interesting insights into the state of practice of model-based testing for embedded systems.”

—DR. LIONEL C. BRIAND, IEEE Fellow, Simula Research Laboratory, Lysaker, Norway, and Professor at the University of Oslo, Norway

“As model-based testing is entering the mainstream, such a comprehensive and intelligible book is a must-read for anyone looking for more information about improved testing methods for embedded systems. Illustrated with numerous aspects of these techniques from many contributors, it gives a clear picture of what the state of the art is today.”

—DR. BRUNO LEGEARD, CTO of Smartesting, Professor of Software Engineering at the University of Franche-Comté, Besançon, France, and coauthor of Practical Model-Based Testing
Model-Based Testing for Embedded Systems
Computational Analysis, Synthesis, and Design of Dynamic Systems Series

Series Editor

**Pieter J. Mosterman**

*MathWorks*
*Natick, Massachusetts*
*McGill University*
*Montréal, Québec*

Discrete-Event Modeling and Simulation: A Practitioner’s Approach, *Gabriel A. Wainer*


Model-Based Design for Embedded Systems, *edited by Gabriela Nicolescu and Pieter J. Mosterman*


Multi-Agent Systems: Simulation and Applications, *edited by Adelinde M. Uhrmacher and Danny Weyns*

Forthcoming Titles:

Computation for Humanity: Information Technology to Advance Society, *edited by Justyna Zander and Pieter J. Mosterman*

Model-Based Testing for Embedded Systems

EDITED BY Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman
## Contents

Preface ix  
Editors xi  
MATLAB Statement xiv  
Contributors xv  
Technical Review Committee xix  
Book Introduction xxi  

### Part I  Introduction  

1   A Taxonomy of Model-Based Testing for Embedded Systems from Multiple Industry Domains  
    Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman  
   3  

2   Behavioral System Models versus Models of Testing Strategies in Functional Test Generation  
    Antti Huima  
   23  

3   Test Framework Architectures for Model-Based Embedded System Testing  
    Stephen P. Masticola and Michael Gall  
   49  

### Part II  Automatic Test Generation  

4   Automatic Model-Based Test Generation from UML State Machines  
    Stephan Weißleder and Holger Schlingloff  
   77  

5   Automated Statistical Testing for Embedded Systems  
    Jesse H. Poore, Lan Lin, Robert Eschbach, and Thomas Bauer  
   111  

6   How to Design Extended Finite State Machine Test Models in Java  
    Mark Utting  
   147  

7   Automatic Testing of Lustre/SCADE Programs  
    Virginia Papailiopoulou, Besnik Seljimi, and Ioannis Parissis  
   171  

8   Test Generation Using Symbolic Animation of Models  
    Frédéric Dadeau, Fabien Peureux, Bruno Legeard, Régis Tissot, Jacques Julliand, Pierre-Alain Masson, and Fabrice Bouquet  
   195
Part III  Integration and Multilevel Testing

9  Model-Based Integration Testing with Communication Sequence Graphs
Fevzi Belli, Axel Hollmann, and Sascha Padberg 223

10 A Model-Based View onto Testing: Criteria for the Derivation of Entry Tests for Integration Testing
Manfred Broy and Alexander Pretschner 245

11 Multilevel Testing for Embedded Systems
Abel Marrero Pérez and Stefan Kaiser 269

12 Model-Based X-in-the-Loop Testing
Jürgen Großmann, Philip Makedonski, Hans-Werner Wiesbrock, Jaroslav Svacina, Ina Schieferdecker, and Jens Grabowski 299

Part IV  Specific Approaches

13 A Survey of Model-Based Software Product Lines Testing
Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr 339

14 Model-Based Testing of Hybrid Systems
Thao Dang 383

15 Reactive Testing of Nondeterministic Systems by Test Purpose-Directed Tester
Jüri Vain, Andres Kull, Marko Käärämäes, Maili Markvardt, and Kullo Raiend 425

16 Model-Based Passive Testing of Safety-Critical Components
Stefan Gruner and Bruce Watson 453

Part V  Testing in Industry

17 Applying Model-Based Testing in the Telecommunication Domain
Fredrik Abbors, Veli-Matti Aho, Jani Koivulainen, Risto Teittinen, and Dragos Truscan 487

18 Model-Based GUI Testing of Smartphone Applications: Case S60™ and Linux®
Antti Jääskeläinen, Tommi Takala, and Mika Kataru 525

19 Model-Based Testing in Embedded Automotive Systems
Paweł Skruch, Mirosław Panek, and Bogdan Kowalczyk 545

Part VI  Testing at the Lower Levels of Development

20 Testing-Based Translation Validation of Generated Code
Mirko Conrad 579
Contents

21 Model-Based Testing of Analog Embedded Systems Components 601
Lee Barford

22 Dynamic Verification of SystemC Transactional Models 619
Laurence Pierre and Luca Ferro
Preface

The ever-growing pervasion of software-intensive systems into technical, business, and social areas not only consistently increases the number of requirements on system functionality and features but also puts forward ever-stricter demands on system quality and reliability. In order to successfully develop such software systems and to remain competitive on top of that, early and continuous consideration and assurance of system quality and reliability are becoming vitally important.

To achieve effective quality assurance, model-based testing has become an essential ingredient that covers a broad spectrum of concepts, including, for example, automatic test generation, test execution, test evaluation, test control, and test management. Model-based testing results in tests that can already be utilized in the early design stages and that contribute to high test coverage, thus providing great value by reducing cost and risk. These observations are a testimony to both the effectiveness and the efficiency of testing that can be derived from model-based approaches with opportunities for better integration of system and test development.

Model-based test activities comprise different methods that are best applied complementing one another in order to scale with respect to the size and conceptual complexity of industry systems. This book presents model-based testing from a number of different perspectives that combine various aspects of embedded systems, embedded software, their models, and their quality assurance. As system integration has become critical to dealing with the complexity of modern systems (and, indeed, systems of systems), with software as the universal integration glue, model-based testing has come to present a persuasive value proposition in system development. This holds, in particular, in the case of heterogeneity such as components and subsystems that are partially developed in software and partially in hardware or that are developed by different vendors with off-the-shelf components.

This book provides a collection of internationally renowned work on current technological achievements that assure the high-quality development of embedded systems. Each chapter contributes to the currently most advanced methods of model-based testing, not in the least because the respective authors excel in their expertise in system verification and validation. Their contributions deliver supreme improvements to current practice both in a qualitative as well as in a quantitative sense, by automation of the various test activities, exploitation of combined model-based testing aspects, integration into model-based design process, and focus on overall usability. We are thrilled and honored by the participation of this select group of experts. They made it a pleasure to compile and edit all of the material, and we sincerely hope that the reader will find the endeavor of intellectual excellence as enjoyable, gratifying, and valuable as we have.

In closing, we would like to express our genuine appreciation and gratitude for all the time and effort that each author has put into his or her chapter. We gladly recognize that the high quality of this book is solely thanks to their common effort, collaboration, and communication. In addition, we would like to acknowledge the volunteer services of those who joined the technical review committee and to extend our genuine appreciation for their involvement. Clearly, none of this would have been possible had it not been for the
Preface

continuous support of Nora Konopka and her wonderful team at Taylor & Francis. Many thanks to all of you! Finally, we would like to gratefully acknowledge support by the Alfried Krupp von Bohlen und Halbach Stiftung.

Justyna Zander
Ina Schieferdecker
Pieter J. Mosterman
Applying Model-Based Testing in the Telecommunication Domain

Fredrik Abbors, Veli-Matti Aho, Jani Koivulainen, Risto Teittinen, and Dragos Truscan

CONTENTS

17.1 Overview ........................................................................... 488
  17.1.1 Process and tools .......................................................... 488
  17.1.2 System under test ......................................................... 490
17.2 UML/SysML Modeling Process ........................................... 491
17.3 Model Validation .............................................................. 498
17.4 Model Transformation—From UML to QML ......................... 500
  17.4.1 Generating the interfaces and ports of the system .............. 501
  17.4.2 From UML data models to QML message types .............. 502
  17.4.3 Mapping the UML state machine to the QML state machine .. 503
  17.4.4 Generating the QML test configuration ......................... 504
  17.4.5 Assigning the state model to the SUT specification .......... 505
17.5 Test Generation ............................................................... 506
  17.5.1 QML ......................................................................... 506
  17.5.2 Test generation criteria ..................................................... 506
  17.5.3 Requirements traceability ................................................. 508
  17.5.4 Test concretization ......................................................... 509
17.6 Test Execution .................................................................... 511
  17.6.1 Load testing mode .......................................................... 512
  17.6.2 Concurrency in model-based testing ............................... 512
  17.6.3 Executable test case ......................................................... 512
  17.6.4 Context ...................................................................... 513
  17.6.5 Run-time behavior ......................................................... 514
17.7 Requirement Traceability ................................................... 514
  17.7.1 Tracing requirements to models ....................................... 516
  17.7.2 Tracing requirements to tests ........................................... 516
  17.7.3 Back-tracing of requirements ......................................... 518
17.8 Related Work ..................................................................... 519
17.9 Conclusions ................................................................. 521
References ............................................................................. 522

It is in the public domain that model-based testing (MBT) has been used for years and its benefits have been emphasized by numerous publications.* Despite this, there is still one question pending. If MBT is an excellent way to test, why has MBT not made a major breakthrough in the telecommunication industry? In order to find the answer to this

*Results of case studies on industrial context have been explained in numerous publications. Dalal et al. described case studies and experiences in (Dalal et al. 1999), and Prenninger, El-Ramly, and Horstmann listed and evaluated selected case studies in Prenninger, El-Ramly, and Horstmann (2005).
A systematic methodology was developed during the project. The methodology uses a system model instead of test models in contrast with many other MBT systems. In the context of the methodology, the term system model refers to models that define the behavior of the system under test (SUT) instead of the behavior of test cases. In addition, during the development, the methodology was constantly evaluated to understand benefits and problematic aspects of the MBT technology.

The methodology includes a process and the supporting tool chain. The methodology was developed for testing functionality of an MSC Server (Mobile Services Switching Centre Server), a network element, using a functional testing approach, that is, the product was tested using a black-box testing technique. The methodology exploited a so-called offline testing approach, where test cases were generated from the models before execution, instead of using an online testing approach, where models are interpreted step by step and each step is executed, instantly.

The content of this chapter will first provide an overview of the project in Section 17.1 that helps understand the following subsections within the chapter. Sections 17.2 through 17.4 focus on process, model development, validation, and transformation aspects. Sections 17.5 through 17.6 describe test generation and test execution aspects. Section 17.7 is devoted to requirement traceability as it spans across the entire process and the tool chain. Conclusions are provided in Section 17.9.

17.1 Overview

Firstly, the overview will provide a high-level view of the process and tools. Next, the SUT and its characteristics are explained. These together provide a detailed context for the work.

17.1.1 Process and tools

The high-level process used in the project is depicted in Figure 17.1. The top level of the process uses the MATERA approach (Abboz 2009a). Four major phases can be identified from the process. First, the requirements are processed and models describing the SUT are created. The models are validated using a set of validation rules in order to improve the quality of the models. Second, the tests are generated from the models. The test generation phase produces executable test scripts. Third, the test scripts are executed with the help of a test execution system. The execution phase produces test logs that are used for further analysis. Fourth, the tests are analyzed in case of failures and requirement coverage tracing is performed. The analysis exploits the test logs and the models. The phases of the process are described in detail in Sections 17.2 through 17.5. In addition, requirement traceability...
is discussed in Section 17.6 as it is not restricted to any particular phase but spans across all the phases.

The process is supported by the tool chain depicted in Figure 17.2. The figure resembles Figure 17.1 and therefore indicates the roles of the tools with respect to the described process. The top level is supported by the MATERA framework (Abbors, Bäcklund, and
Truscan 2010), which is developed as a plug-in to the No Magic MagicDraw tool (Magic 2009). In MATERA, the Unified Modeling Language (UML) (Object Management Group models d) is edited, validated, and transformed to Qtronic Modeling Language (QML) (Conformiq 2009b) models and given as an input into the Conformiq Qtronic tool (Conformiq 2009a) for test generation. Qtronic outputs test scripts that are executed with Nethawk EAST (Nethawk 2009). The test logs produced by EAST are analyzed and evaluated against the original models using the MATERA test evaluation function. The relevant details of the tools are provided in Sections 17.2 through 17.5.

The UML models are edited with No Magic MagicDraw tool (Magic 2009). In addition, MagicDraw is used for model validation via custom rules implemented using the Object Constraint Language (OCL) (Object Management Group b). The UML models are transformed with a script into QML models and given as an input into the Conformiq Qtronic tool (Conformiq 2009a) for test generation. Qtronic outputs test scripts are executed with Nethawk EAST (Nethawk 2009). The test logs produced by EAST are analyzed and evaluated against the original models using a test evaluation script. The relevant details of the tools are provided in Sections 17.2 through 17.5.

17.1.2 System under test

The project focused on testing the Mobility Management (MM) feature of a MSC Server, that is, the MSC Server acted as the SUT. It is a key network element of second and third generation mobile telecommunication networks that establishes calls and controls handovers during calls. The MSC Server is capable of handling up to several million users and at the same time provides a near zero downtime. The MSC Server communicates with a number of other network elements as illustrated in Figure 17.3.

![Figure 17.3](image)

**FIGURE 17.3**
Project-related mobile telecommunication network elements.
The MSC Server is a typical telecommunication network element from a testing point of view. It has multiple interfaces with other network elements. In addition, the MSC Server communicates with multiple network elements of the same kind, for example, a MSC Server connects to many Radio Network Controllers (RNCs) and Base Station Controllers (BSCs). Communication between the network elements is concurrent and performance scalability aspects are already taken into account in the architecture of the network. The MSC Server also communicates with mobile phones using logical connections, that is, the MSC Server does not have a physical connection with the mobile phones, but the MSC Server uses services provided by other network elements. These elements are part of the radio access networks, that is, Base-Station Subsystem in the second generation network and Radio Network Subsystem (RNS) in the third generation network. The details of the network architecture are specified in the 3GPP Technical Specification 23.002 (The 3rd Generation Partnership Project 2005). Evolution from the second generation GSM systems to the third generation UMTS networks and a detailed description of the latter technology are provided by Kaaranen et al. (2005).

17.2 UML/SysML Modeling Process

The models of the SUT are created following the MATERA approach. The approach starts from the textual requirements of the system, the MSC Server, and incrementally builds a collection of models describing the SUT from different perspectives. In this context, textual requirements refer to the collection of stakeholder requirements, as well as additional documents such as protocol specifications, standards, etc. The modeling phase captures several perspectives (architecture, behavior, data, and test configuration) of the system, at several abstraction levels, by spanning the Functional View and Logical View layers described in Figure 17.4. The functional view defines how the system is expected to behave when it interacts with its users. The logical view describes the logical parts of the system, with behaviors and interactions, without relating to the actual implementation of the system. Each perspective is initially specified on the functional view via the feature, requirements, use case, and sequence diagrams, and it is subsequently refined on the logical view (state machine diagrams, class diagrams, and object diagrams). There are both horizontal (between the perspectives on the same level) and vertical relationships (refinements) among the specification artifacts in this process, as will be illustrated throughout this section.

FIGURE 17.4
Modeling perspectives in the NSN case study.
The UML (Object Management Group) is used as a specification language for system modeling in the project. Additionally, the requirements diagrams of the Systems Modeling Language (SysML) (Object Management Group) are used to capture the requirements of the system in a graphical manner. The No Magic MagicDraw tool was employed to create the models. MagicDraw is a commercial software and system modeling tool, which offers support for both UML and SysML. The tool also offers support for automatic code generation, model validation, model analysis, reporting, etc., and can be extended with the use of various plug-ins.

The system models are created in a systematic manner, based on the MATERA guidelines (Figure 17.5), starting from the textual requirements. The approach consists of five phases. In each phase, a new set of models is created. Each model describes the system from a different perspective. The approach is iterative, so each phase can be visited several times, and the models are constructed incrementally.

The first phase deals with the identification of stakeholder requirements, standards, and associated specifications. Since the models are derived from requirements, it is necessary to identify and collect as much relevant information about the system as possible. The system models are later built based on the collected information.

In phase two, feature models and requirements models are created from the information collected in the previous phase. Initially, the features of the SUT are specified using UML Class diagrams (Figure 17.6). The feature models are mainly derived from product requirements and they give a rough outline of which features and functionality the system must be able to perform. Each class describes one feature, whereas mandatory and optional relationships between features are modeled using aggregation and composition relationships between classes. The feature diagram follows the principles of functional decomposition, where high-level features are decomposed into subfeatures.

![System modeling process diagram](image-url)

**FIGURE 17.5**
System modeling process.
A set of requirement models are created using SysML Requirements Diagrams (Figure 17.7). The requirements are specified on several levels of abstraction following principles of functional decomposition. One such model is created for every leaf in the feature diagram. The purpose of these models is to structure the requirement specifications corresponding to each feature in a graphical manner. They are structured in requirements diagrams similar to a UML class diagram in which the classes are annotated with the ≪requirement≫ stereotype. A requirement in SysML is specified using different properties including an id field, a textual description, and the source of the requirement. The textual description gives a brief explanation of the requirement, while additional details (e.g., technical specifications) are added in the documentation field of each requirement (not visible in the previous figure). The source field directs the document to where the requirement has been extracted from.

FIGURE 17.6
A feature diagram of the MSC Server.

FIGURE 17.7
Requirements are traced to other requirements on the same abstraction level or between requirements and other model elements in UML by using relationships such as DeriveReqt, Satisfy, Verify, Refine, Trace, or Copy. If necessary, requirements are also arranged into different categories such as functional, architectural, and communication (data).

In the third phase, a use case model and a set of sequence diagrams are created. The purpose of the use case model (Figure 17.8) is to present a graphical overview of the main functionality of the system. The use case model also identifies the border of the system and the external entities (actors) with which the system must interact. Each use case has a detailed textual description using a tabular format (see Figure 17.9). The description includes fields for precondition, postcondition, actors using the use case, possible sub-use cases, as well as an abstract textual description of the sequence of actions performed by the system when the scenario modeled by the use case is in use. Message sequence charts (MSCs), or sequence diagrams, are illustrated in Figure 17.10 and are primarily used to describe the interactions between different entities 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 entities for a given use case. Basically the intended usage of MSC is twofold: to discover entities communicating with the system and to identify the message exchange between these entities. The messages exchanged between entities (referred to as lifelines in the context of sequence diagrams) are extracted from the protocol specifications referenced by the requirements.

In the fourth phase, we define the domain, data, and state models of the SUT. The domain model (Figure 17.11) is represented as a class diagram showing the domain entities as classes and their properties (attributes). The domain model also describes the interfaces that the domain entities use for communicating with one another. Each interface contains a set of messages that can be received by an entity, modeled as class operations. The names of the operations are prefixed with the acronym of the protocol level at which they are used, similar to the approach followed in the sequence diagrams. For instance, the

![Use case diagram of the MSC server.](image-url)
Applying MBT in the Telecommunication Domain

Name: Location Update
Author: Fredrik Abbors
Date: 1.10.2008
Actors: MS, HLR, BSS, RNS
Sub-cases: Authentication, Ciphering
Description:
The MS requests a location update from MSS. The MSS can in some cases initiate authentication and ciphering of the MSS. Finally, the MSS will respond to MS with the location area.
Pre-conditions: Connection established between MS and MSS.
Post-conditions: The location area of MS stored/updated in MSS's registers.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Actor input</th>
<th>System response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MS sends requests to MSS to update its location</td>
<td>MSS responds and accept message containing information about the location area</td>
</tr>
<tr>
<td>2</td>
<td>MS responds with an acknowledge message</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 17.9**
Tabular description of the location update use case.

**FIGURE 17.10**
Sequence diagram describing the location update procedure.
FIGURE 17.11
Applying MBT in the Telecommunication Domain

prefix depicts a class method used at the MM protocol. The set of messages that can be exchanged between the MSC Server (MSS) and the Mobile Subscriber (MS) at the MM protocol are modeled by two different class interface elements (i.e., MM), one for each direction of communication. The approach allows a clear separation of the communication between different network elements and on different protocol levels. The domain model is built iteratively and is mainly derived from the sequence diagrams and the architectural requirements. Each lifeline in the sequence diagrams generates a new class, whereas the interfaces of the class are obtained from the messages that each lifeline in the sequence diagrams receives.

The data model (Figure 17.12) describes the different message types used in the domain model. That is, every message on each interface in the domain model is linked to the corresponding message in the data model. The messages are modeled explicitly via class diagrams. Since, in the telecommunication domain, the main unit of data exchanged between entities is the message (or PDU), we focus our attention on how different message types and their structure (parameters) can be described. By analyzing the communication requirements and the domain models (where messages have been described as class operation with parameters), we create a data model of the system in which each message type is represented by a class, while the parameters of the message are represented as class attributes. We structure the message definition based on their corresponding protocols (one diagram per protocol) and use inheritance to model common parameters for a given message. Figure 17.12 shows a class diagram specifying the messages used by the MM protocol. The MM super-class defines the parameters common to all messages, whereas leaf classes define mandatory parameters for each message type. Optional parameters can be added following a similar approach. One important aspect of the data model is that it does not contain all the mandatory fields of a given PDU, but only those necessary to model the SUT at the current abstraction level. The rest of the parameters will be set up during the test generation level when the abstract test cases will be transformed into executable ones (see Section 17.5, Test Concretization).

State models are used to describe behavior of the SUT. The state model of the MSC Server is derived by analyzing sequence diagrams of each use case one by one. That is, for

FIGURE 17.12
each sequence diagram, the state of a given object in which each message is received and what messages are sent from that particular state is identified. The former will become *trigger messages*, while the latter will become *actions* on the state machine transitions. By overlapping the states and transitions extracted from each sequence diagram, the full state model of the SUT is obtained. The resulting state model may also contain hierarchical states that will help in reducing the complexity of the model. Figure 17.13 shows a state machine model of the SUT for the location update procedure.

In the *last phase*, a test configuration model is created. This model is represented using a UML object diagram and serves to specify the test setup. The elements of this diagram are basically instances of the entities defined in the domain model, showing a particular configuration at a particular point in time. Figure 17.14 shows a test configuration with two mobile phones connected to a 2G network and a 3G network, respectively.

### 17.3 Model Validation

Humans tend to make mistakes and omit things. Therefore, in modeling, it is necessary for the models to be validated before using them to, for example, automatically generate code or test cases. In our methodology, we take advantage of the model validation functionality of MATERA to check the models for consistency, correctness, and completeness before proceeding to the next step of the process. The idea behind consistency validation is to check for contradictions in the model, for example, a message name in a sequence diagram should match the name of the operation in the class corresponding to that lifeline. Correctness ensures that the models conform to the modeling language (e.g., UML), whereas completeness checks that all necessary information fields have been properly filled out for each
FIGURE 17.14
Test configuration model with two mobile phones connected to the SUT.

Element. Model validation can be considered “best practice” in modeling since it increases the quality of the models by ensuring that all relevant information is present or dependencies between elements are correct.

The MATERA framework utilizes the validation engine of MagicDraw for model validation. The engine uses the OCL (Object Management Group b), a formal language for specifying rules that apply to UML models and elements. These rules typically specify invariant conditions that hold true for the system being modeled. Rules written in OCL can be checked against UML models and it can be proved that nothing in the model is violating them. UML, the main modeling language supported by MagicDraw, is accompanied by several predefined suites of validation rules. For example, UML models can be validated for correctness and completeness. There are also additional validation suites for different contexts that can be used. For example, in the SysML context there is a set of validation rules that apply to SysML diagrams. The different validation suites can be checked against either all models or selected models.

Beside predefined validation rules provided by MagicDraw, custom (domain-specific) validation rules can also be created and executed, using the MATERA framework. A complementary set of validation rules was defined (Abbors 2009b) in order to increase the quality of the models with respect to the modeling process. The main purpose of these rules is to ensure a smooth transition to the subsequent steps in the testing process such as generating the input specifications for the test generation tool, or the test generation itself. Next, the validation suites in MATERA and the creation of custom validation rules are described.

An OCL rule normally consist of three parts: (1) a context that defines to which language elements the rule applies (e.g., class or state), (2) a type that specifies if the rule is a, for instance, an invariant, a precondition, or a postcondition, and (3) the rule itself. Optionally, an OCL rule can also contain a name. An example of an OCL rule is shown in Listing 17.1.

Listing 17.1
Example of an OCL rule

context Region inv initial_and_final_state :
  subvertex → exists(v:Vertex | v.oclAsType(Pseudostate).kind = PseudostateKind::initial) and subvertex → exists(v:Vertex | v.oclIsTypeOf(FinalState))
The OCL rule in this example checks that every state machine has an initial and a final state. As one can see, the context of the rule in this example is a Region and the type of the rule is invariant (inv), which means that the expression must always hold true for every instance of type Region. In this example, the OCL rule also has a name, “initial_and_final_state,” specified next to the type. By assigning proper names to OCL constraints, it becomes easier to understand the purpose of the constraint and, in addition, it allows the constraint to be referenced by name.

In the validation engine, OCL constraints are treated similarly to model elements. As a result, each constraint has a number of editable properties such as name, specification, constrained element, etc. (see Figure 17.15). The name property specifies the name of the constraint. The specification property specifies the rule itself and its type, while the constrained element specifies the context of the constraint.

The validation suites can be invoked at any time during the model creation process. Upon invocation, each rule will be run against the element types or element instances for which it has been defined. If elements violating any rule are found, the user is notified in a Validation Results editor (Figure 17.16). By clicking a failed rule, the elements violating the rules are presented to the user.

17.4 Model Transformation—From UML to QML

The resulting collection of UML models is used for generating the input model for the Conformiq’s Qtronic test generation tool, using the MATERA model transformation module. The model in Qtronic is specified using the QML which is discussed more in Section 17.5.1. QML is a mixture of UML state machines and Java, the latter being used as an action
language. As such, the system models created in MagicDraw are not directly compatible with the Qtronic tool and hence must be transformed into a representation understood by Qtronic, namely QML. The MATERA model transformation module (Abbors et al. 2009) automatically transforms the UML models into the corresponding QML representation. Figure 17.17 shows how different models are mapped onto QML.

17.4.1 Generating the interfaces and ports of the system

In QML, interfaces are specified within a system-block. The interfaces describe the ports that can be used to communicate with the environment and which message types can be sent and received on each port. The Inbound ports declare messages to be received by the SUT from the environment, whereas the Outbound ports declare messages to be sent from the SUT to the outside world.

The ports of the SUT are obtained directly from interface classes in the domain model (see Figure 17.11). Inbound messages are taken from the interface realization offered by the SUT, and Outbound messages are taken from the interface realization used by the SUT. The name of the ports will be composed of two components, the direction and the interface name. UML operations are listed as messages that are transferred through the ports. The structure of the messages is declared elsewhere as records. The partial result of applying the transformation on the MM interfaces in Figure 17.11 is shown in Listing 17.2.
Model-Based Testing for Embedded Systems

Defining Domain Models

- Network configuration model
- State machines + requirements

State machines

FIGURE 17.17
Mappings from UML to QML.

Listing 17.2
An example of generated Example of QML system-block

```c
//System block example
system {
    Inbound MM_in: location_updating_request,
    authentication_response, identity_response,
    TMSI_reallocation_complete, CM_service_request, alerting,
    call_confirmed, connect, connect_ack, setup, disconnect,
    release, release_complete;
    Outbound MM_out: location_updating_accept,
    authentication_request, identity_request, alerting,
    call_proceeding, connect, connect_ack, setup, disconnect,
    release, release_complete;
}
```

17.4.2 From UML data models to QML message types

In QML, messages are described as records that are used for communicating with the environment. QML records are user-defined types similar to classes. The fields of a record can be of type: `byte, int, boolean, long, float, double, char, array, String`, or of another record type. In the transformation, records are obtained from classes in the UML data model. Attributes of the UML classes are transformed into the fields of the record. Inheritance in UML is reflected in QML using the `extends` relationship. For instance, the `location_update_request` record in Listing 17.3 is obtained from a class with the same name in Figure 17.12 following the described approach. The model does not indicate value ranges of the fields. Instead, the value ranges can be checked by the protocol codecs provided by the test system.
Listing 17.3
QML record declaration for LOCATION_UPDATING_REQUEST

record MM_messages{
    public String protocol_discriminator;
    public String skip_indicator;
    public String message_type;
}

//record inheritance
record location_updating_request extends MM_messages{
    public int location_updating_type;
    public String ciphering_key_sequence_number;
    public String location_area_identification;
    public String mobile_station_classmark_1;
    public String mobile_identity;
    public String domain;
}

17.4.3 Mapping the UML state machine to the QML state machine

As mentioned previously, the behavior of the SUT can be specified in Qtronic either textually in QML or graphically using a restricted version of the UML state machines. For simplicity, the latter option was chosen as the target of our transformation. Thus, the transformation is basically a matter of transforming the UML state machine into the corresponding state machine used by the Qtronic tool, which in practice is equivalent with a transformation at the XMI-level. Figure 17.18 shows the same state machine transformed to QML. As one can see, there is a strong similarity between the two models, albeit with small differences. For instance, both state and substate machines are supported and propagated at the Qtronic level. In UML, triggers and actions are declared as methods (selected only from the operations of the interface classes in the domain model).

In QML, triggers are implemented by messages (record instantiations) received on a certain port, whereas actions can be seen as methods of the SUT class definition. This approach allows one to perform further processing of the system data before sending a given message to the output port.* The method generated for the MM_LOCATION_UPDATING_ACCEPT() in Figure 17.11 is shown in Listing 17.4.

Listing 17.4
Example of a generated QML method

void MM_LOCATION_UPDATING_ACCEPT(){
    MM_LOCATION_UPDATING_ACCEPT location_updating_accept;
    MM_out.send(location_updating_accept);
    return;
}

*If necessary, additional QML instructions can be manually inserted into the generated methods, before the .send() statement.
17.4.4 Generating the QML test configuration

As illustrated in Figures 17.11 and 17.14, a :MS is a user of the SUT who communicates with it via the MM interface. The test configuration in our example consists of two MS. In QML, a subscriber is modeled as a record (Listing 17.5) with the same attributes as in the domain model (Figure 17.11). The test configuration is translated into QML in two steps. In the first step, the properties of the test components are extracted from the UML domain model and are declared in the constructor method of the SUT specification class (MSS in our case) as shown in Listing 17.6. In the second step, each test component is instantiated and the properties are initialized with the values taken from the configuration diagram (Listing 17.7). The test components are stored in an array, which is later used for starting concurrent test components of the test system in separate threads.

Listing 17.5
Example of subscriber record

```java
record Subscribers {
  public String my_name;
  public boolean followOn;
  public String domain;
  public String role;
  public boolean registered;
  public String imsi;
}
```
Listing 17.6
Example of subscriber record initialization

```java
public InitializeSubscriber(Subscribers sub) {
    my_name = sub.my_name;
    followOn = sub.followOn;
    domain = sub.domain;
    role = sub.role;
    registered = sub.registered;
    imsi = sub.imsi;
}
```

Listing 17.7
main method of the SUT specification class

```java
// *** MAIN ***
void main() {
    Subscribers mySubscribers[] = new Subscribers[3];
    mySubscribers[0].my_name = "ms#1";
    mySubscribers[0].followOn = false;
    mySubscribers[0].domain = "2G";
    mySubscribers[0].role = "MOC";
    mySubscribers[0].registered = true;
    mySubscribers[0].imsi = "234800000000921";

    mySubscribers[1].my_name = "ms#2";
    mySubscribers[1].followOn = false;
    mySubscribers[1].domain = "2G";
    mySubscribers[1].role = "MSC";
    mySubscribers[1].registered = true;
    mySubscribers[1].imsi = "234800000000922";

    for (int i = 0; i <= 1; i++) {
        MSS mss = new MSS(mySubscribers[i]);
        Thread t = new Thread(mss);
        t.start();
    }
}
```

17.4.5 Assigning the state model to the SUT specification

At this point, the only thing that remains to be done is to connect the SUT class specification to the graphical state model. This is done by calling the constructor method for the state machine. Once the state machine has been constructed, the concurrency of the SUT can be tested by starting (via the `Thread.start()` method) separate execution threads for each test component (i.e., subscriber) (Listing 17.8). The approach allows for different MSs to concurrently communicate with the SUT using different configuration parameters. This is necessary as in telecom systems such as the MSC Server, one must test the presence of
multiple MSs interacting with the MSC Server (in practice, one MSC Server can serve up to several million users). In addition, we must test calls between pairs of subscribers, where one call requires two subscribers, that is, the caller and the receiver (known as A and B subscribers).

**Listing 17.8**
State machine instantiation in Qtronic

```java
for (int i = 0; i < 1; i++) {
    MSS mss = new MSS(mySubscribers[i]);
    Thread t = new Thread(mss);
    t.start();
}
```

### 17.5 Test Generation

The test generation phase follows the modeling phase in the overall process. The test generation exploits the QML models transformed from the UML models. The supporting tool chain implements the test generation phase using the Conformiq Qtronic tool. Qtronic is a tool for Automated Test Design. It derives tests automatically from system models that represent the desired behavior of the SUT. The generated tests are black-box tests and so they evaluate the SUT based on its external behavior, not by monitoring its internal workings.

#### 17.5.1 QML

The systems model given as input into Qtronic is expressed in terms of a language called QML. A model is a collection of the following:

- Textual source files in a Java-compatible but extended notation that describe data types, constants, classes, and their methods.
- UML state-chart diagrams representing the behavioral logic of active classes as an alternative to representing the logic textually.
- Class diagrams as a graphical alternative to declare classes and their relationships.

A QML model is therefore essentially an object-oriented computer program, an abstract implementation of the system to be tested.

The diagrams can be drawn using various tools that Qtronic works with, such as Conformiq Modeler, Enterprise Architect, IBM Rational Software Developer, or IBM Rhapsody. It is also possible to create models completely textually, that is, all the diagram types are optional.

#### 17.5.2 Test generation criteria

Given a system model, Qtronic automatically identifies a number of test cases that together cover the testing goals selected for test generation. Appropriate test input data as well as the correct expected output is automatically calculated and generated by the tool without 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 an analysis of the behavior implied by the model, instead of being based on syntactic analysis or simple heuristics. Qtronic uses model-based coverage criteria to select a set of test cases to form a good test suite. The coverage or testing goals are used to guide Qtronic to look for certain behaviors from models or to enable certain behaviors. A test case covers a certain testing 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 to construct test cases, and at the same time, it maps the test cases to the different test goals that result from the coverage settings. It then selects from the constructed test cases a set that covers all of the resulting test goals using a minimal cost test suite. This ensures that the suite is reasonably small and compact, and 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 covering all test goals as early as possible, that is, after as few messages as possible, providing better separation of concerns between test cases.

A test suite generated by Qtronic has the following characteristics:

- In order to have good error detection capabilities, the generated test suite covers as many testing goals as possible.

- In order to avoid redundant testing, the generated test suite is as compact as possible while individual test cases in the suite are relatively short in order to ease the test execution and debugging.

- In order to provide better separation of concerns between test cases, the test goals are covered as early as possible in test cases.

Qtronic makes the testing goals accessible to the user in the Qtronic user interface (UI). By selecting different testing goals, the user can affect how Qtronic generates test cases. This is the primary vehicle in Qtronic for a user to have a say in how the tests are generated. Figure 17.19 shows testing goals in the Qtronic UI. From this view in the coverage editor the user can see, for example, that the generated test suite covers all requirements (more on requirements below) related to category “1.2” but not all requirements in some other categories. One can also see that 62% of all the states and 60% of the transitions are covered by the test suite.

The selection testing goals that are used depend on how extensive a test suite the user wishes to generate and also on the characteristics of the model and hence the SUT itself. For example, if the model has lots of interesting boundary conditions related to inputs to the system, it is highly recommended for one to enable Boundary Value Analysis as a test generation criterion (see Figure 17.19). With boundary value analysis enabled, Qtronic will attempt to exhaustively generate all possible boundary values based on the various conditions in the model.

Not all testing goals are always reachable. One can, for example, have conflicting if-statements in the model, which cause a certain boundary value case to be statically unreachable. An important aspect of how Qtronic works is that Qtronic will give the user a precise account of which coverage goals were covered by the generated test suite and which ones were left uncovered. Therefore, after test generation, the user knows exactly how well the generated test suite covers the different functionalities of the modeled system (and can react to uncovered areas and change the model if it turns out that there was a defect in the model itself).

*Both of the terms, coverage goal and testing goal, are used in this chapter and they mean the same thing.
17.5.3 Requirements traceability

Most of the testing goals are related to various properties of the model itself (such as state, transitions, conditional branches, etc.). In addition to these structural coverage criteria, user-defined requirements can be used in models to guide the test generation.

Technically, the requirements are embedded in the model using the requirement keyword, as illustrated in Listing 17.9.

Listing 17.9
An example of a requirement in a model

```c
// A requirement embedded in a model
requirement "This is a requirement";
```

For the Qtronic algorithm, a requirement is one additional coverage goal in the model to be covered (see discussion on testing goals in Section 17.5). In generating tests, Qtronic will attempt to generate a test suite that covers all requirements embedded in the model at least once, assuming that the user has chosen requirements as a testing goal prior to starting test generation. The notion “covering a requirement” means that there is an execution in the model that passes through the point where the requirement is defined.

At the end of test generation, Qtronic will report how well the requirements in the model were covered by the generated test suite (just like it does for all other testing goals). What is particularly interesting about the requirements coverage is that it provides automatic traceability from the functional requirements of the system all the way through to the generated test cases and test execution. A traceability matrix maps requirements to test cases, allowing a tester to easily pick up test cases to exercise certain functional areas identified by the corresponding functional requirements. An example of a traceability matrix can be seen in Figure 17.20. For example, if the user wishes to execute a test case that
exercises a “mobile-originating call from a 2G network to a 3G network,” then test case 1 would have to be executed as can be seen from the “Xs” in the highlighted column for test case 1. Furthermore, a failure in test execution can be easily attributed to a functional area by looking at the traceability matrix.

17.5.4 Test concretization

Once the QML system model is created and tests are generated, we have a set of valid test artifacts in terms of test cases, messages, parameters, and testing goals. Test artifacts, from the Qtroamic database, must be transformed into an executable format for test execution.

NetHawk’s Environment for Automated System Testing (EAST) [3] is used for test execution. EAST embeds the SUT in a virtual network incorporating protocols over interfaces under test. EAST provides a Test Creation Environment (TCE) with graphical programming language for defining tests, test suites, message templates etc., and a Test Execution Environment (TEE) for test execution. In the project described in this chapter, TCE is utilized for creation of the message reference library. TEE is used for executing test cases in Load Testing mode, which provides an execution performance close to real network elements. EAST TEE connects a test execution engine to the desired protocol server required by a protocol under test. A Protocol server is a standalone program simulating a protocol stack below the protocol under test. In this context, Protocol under test refers to the level of the modeled behavior of the SUT.

The message reference library is a collection of message descriptions and encoding rules of telecom protocols under test. In other words, it is a definition of the run-time behavior of messages. EAST provides most Protocol Data Units (PDU) of telecom protocols in the form of message templates with default content. Each value of the template field in the reference library is accessible through an API. It is up to the user to select the necessary
Model-Based Testing for Embedded Systems

PDUs from the reference library and create mappings between the parameters from test artifacts and reference library messages. Data models, depicted in Figure 17.12, define the interfaces under test on an abstract level when the message reference library is the actual implementation of PDUs. One could think of the reference library as a refinement model for the data model.

For example, the data model defines a message `MM_LocationUpdateRequest` and Qtronic has generated test artifacts defining parameters and behavior of when the message is sent by the test system. In an executable test case, the event of sending the `MM_LocationUpdateRequest` message will be composed from the implementation of the message in the reference library and of test artifacts calculated by Qtronic. Figure 17.21 shows the relation between reference library and generated test artifacts.

In order to concretize such an executable test case, it is required that there be a tool to combine Qtronic test artifacts and the message reference library refinement data producing EAST executable test cases. For this purpose, a test scripting back-end was implemented. The back-end is connected to Qtronic using an open API. Through the open API, it is possible to create a custom output format (e.g., EAST scripts) from test artifacts and utilize external test libraries such as the EAST reference library. The back-end creates a set of test cases that can be executed on EAST TEE.

![Diagram showing the relation between reference library and generated test artifacts.](image)

**FIGURE 17.21**
Test concretization.
Applying MBT in the Telecommunication Domain

The message reference library may have several versions supporting several protocol versions. All versions implement the same API in order to keep maintenance work to a minimum. The API defines the mapping between the reference library messages and the abstract definitions of messages. If the specification version of the protocol under test changes, the reference library version also must be changed. Abstract definitions and the data model should not have to change.

17.6 Test Execution

Test execution is performed with the help of a test system. The test system is illustrated in Figure 17.22. The test system operation is automated with a custom-made Test Automation Script (TAS). The test system is based on the NetHawk EAST Test Execution Environment (TEE) and on protocol servers providing connectivity to the SUT. In addition, the message reference library, described in Section 17.4, is providing run-time behavior for PDUs called from EAST test scripts. EAST produces test logs on each test run. The logs are used in the analysis and for coverage tracing purposes later on as will be discussed in Section 17.6. The communication between the MSC Server and the test system is monitored with the Wireshark (Wireshark) protocol analyzer. The message reference library was discussed earlier in Section 17.4 from the test concretization point of view. During the test execution, the message reference library is providing an API for encoding and decoding rules for messages sent and received. For example, if a message \texttt{MM\_LocationUpdateRequest} is about to be sent by the TEE, the message reference library provides information about online encoding of the message.

The TAS is used for starting related test steps at the same time, for collecting combined coverage results, and for calculating the final verdict of the test case. After the system model is created and test generation performed, the TAS takes care of all steps required in the test execution phase.

![FIGURE 17.22](image)

Architecture of the test system.
17.6.1 Load testing mode

The Load Testing mode has been selected from different execution environments provided by EAST TEE. The Load Testing mode provides an execution engine allowing several simultaneous calls and has a means for identifying different calls by a special Context. The Context in our environment defines a sequence of actions or events having certain unique characteristics. For example, when simulating an originating phone call, the calling party has its own phone number and the called party has its own number. Both, originating and terminating, calls have their own Context that can be addressed by a phone number as phone numbers are unique. Also, while a phone call is ongoing, the Context can be referred to at a protocol level through a specific connection in case of a connection-oriented communication.

17.6.2 Concurrency in model-based testing

Modeling concurrent systems, such as telecom network elements, requires some extra attention in the modeling phase in order to obtain executable tests from the test generation. The SUT is concurrent by nature, which means that several sequences of events are happening at the same time without synchronization between them. This leads to a situation where it is possible to have large amounts of different, but correct, interleaved sequences of elementary procedures. For example, Event A (EA) has 10 elementary procedures that always happen in a known order. Respectively, Event B (EB) has 10 elementary procedures. In a case where EA and EB are happening in parallel, there are $2^{20} - 1 = 524,288$ possibilities for interleaved elementary procedures of EA and EB. In reality, there are some relationships between EA and EB which narrow down the possibilities. But, when looking from a test generation point of view, it is difficult to create information on models that rule out the possibilities that do not take place in reality. And even after that, it is neither feasible nor wise to generate all possible variations. The most important point is to not generate a test that has one fixed order of parallel events. That is because it is not known beforehand in which order these parallel events happen in the SUT.

For example, originating and terminating calls receive a confirmation when they are connected to each other, but it is not specified which phone receives the confirmation first. Some branching would be necessary in a single test case to cover these types of situations. This will lead to a very complex test case that requires a lot of computational power from the test generation tools. Instead of branching, the solution is to treat originating and terminating calls as separate threads and generate independent test steps for both calls. These steps together form a test case. Both steps are increasing the test coverage (see Section 17.4) of the test case while they are running.

17.6.3 Executable test case

MBT test generation tools such as Qtronic can handle only deterministic behavior. An individual test case should go through a deterministic path on each execution round. But as illustrated in the above example, EA and EB as one test case, there are several options on how a valid test case could behave on interleaved execution leading to nondeterministic behavior. At first, this sounds like a restriction from the test generation tool perspective, but when thinking about this dilemma openly, this is the only way that a test generation tool could produce something reasonable. Concurrency must be handled in the test execution environment and taken into account when creating system models. The system model should define a separate thread for both the originating and terminating phone call. The test generation tool produces one test case where EA and EB are combined in one interleaved
sequence of events. In the creation of an executable test in the test concretization phase, described in Section 17.4, executable events of EA and EB are separated on their own test steps. The entire process is depicted in Figure 17.23.

17.6.4 Context

Figure 17.24 depicts the message routing from the SUT to the correct test step in the test system and vice versa. In the picture, the Router is a book keeper of the active Contexts in the test system. It compares the received message type, header, and possible payload to infer the correct Context. The EAST test case from the example above has two test steps with unique phone numbers. There is also a possibility to define a unique protocol transport layer identifier for each test step, for simplicity, is referred to as connectionID. Most of the communication between the SUT and the test system is connection oriented and it is easy to determine an identifier, connectionID, for each connection during the connection establishment phase. However, because of the nature of the telecom network, there are some connectionless messages exchanged within the network. These messages are mainly used for paging other phones or resources before the connection is known. Here, it is assumed that these broadcast messages carry an address of an entity to which the message belongs. To simplify the example, the address is referred to as phoneNumber.

Generated EAST scripts, one script for each test step, are uploaded to the Load Runner. Each test step registers their Context with a run-time database storing the relation between the Context and the test step. In the example Listing 17.10, the Context can be addressed through two context identifiers, connectionID and phoneNumber.

Listing 17.10
Example of the relation between the Context and the test system

```java
String assign { destination = $connectionID , strsource = "1" }
LoadEngine.setContextId { Variable = $phoneNumber , type = "reference" }
String assign { destination = $phoneNumber , strsource = "62030614610001" }
LoadEngine.setContextId { Variable = $phoneNumber , type = "reference" }
```

FIGURE 17.23
From the system model to executable test.
17.6.5 Run-time behavior

There are two possibilities for the test step to identify its Context. First, it could do a connection establishment after which connectionID can be used to address messages through the router. Second, the test step waits for a broadcasting message with phoneNumber defining the correct Context after which the connection should be created as in the first case. Broadcast messages are sent to all test steps but discarded if the phoneNumber does not match. After the Context is known, the Router depicted in Figure 17.24 knows the destination of each message.

EAST TEE is writing a textual log that contains information of executed events and covered requirements. The log file is an input for the requirement traceability phase described in Section 17.7.

17.7 Requirement Traceability

Conventional test scripting is based on static test cases, that is, a test case has a name and corresponding test code that is not modified significantly between consecutive versions. However, this is not the case when tests are generated from models automatically. In automatic test generation, there are no guarantees that different test generation rounds produce similar test scripts. Thus, as part of the methodology, an approach was developed for tracing the requirements throughout the entire MBT process, from models to test runs and back to models as illustrated in Figure 17.25. This approach provides a consistent way to monitor the modeling, generation, and test execution status, as well as the test completion rates. In addition, tracing of requirements helps understand what kind of impact new or modified requirements have on different artifacts of the testing process. All in all, the requirement tracing process supports short feedback loops that, in turn, support modern product development conventions such as iterative and agile development practices.

The use of the requirements fits well in product testing because the features of the product are defined as a set of requirements. It does not require any additional information on top of the regular product development information. Instead, it uses information that is readily available in the product development. This makes it easier to integrate the MBT tool chain with the current frameworks and with the tools used in product development. In addition, it is also important to be able to evaluate how many tests cover each requirement to support test prioritization and optimization.
The use of the requirement tracing also helps in the process of test selection and prioritization. Typically in product development, some features (requirements) of the product are more important than others. This affects the testing process in a way that more important features must be tested more extensively compared to others. The prioritization of the test cases can be done in two different phases of the testing process, during test generation or test execution. If the behavioral model of the SUT used for test generation is adorned with priority information, then the test generation tool will be able to generate test cases and calculate their execution order. However, if such a possibility is not supported by the test generation tool, then having the generated test cases tagged with the requirements they are testing will allow the selection of the most important test during execution based on the priority of the requirement. Especially in the telecom domain, where the number of generated test cases tends to be very large, having the possibility to select the most important test cases to be executed based on the requirement to be tested is an important aspect.

At a more detailed level, requirement tracing allows requirements to be propagated to test specifications. Further, functional testing must verify that all requirements have been covered by test. Needless to say, requirements are the keystone in any successful project implementation, and hence, they must be traceable both to models and tests. Tracing requirements during development ensures that all requirements have been implemented and that no functionality has been overlooked. Tracing requirements to tests can even help in identifying missing tests, that is, where critical requirements do not trace to any test. Finally, if a test fails, one can trace the requirement back to the models from where it originated, in order to identify the error. This facilitates the process of identifying which parts of the system model cause a set of test cases to fail. It is described in brief how these aspects are addressed in the following subsections.
17.7.1 Tracing requirements to models

Requirements traceability is one of the key features of MATERA and is based on the approach described in Section 17.2. As presented in Figure 17.7, requirements are structured hierarchically. Top-level requirements are traced to different models (e.g., state machine diagrams), whereas the rest of the requirements are traced to model elements to which they apply (e.g., a transition or a state).

In our modeling process, requirements are propagated to different parts of the models to indicate a relationship between requirements and model elements. For instance, communication-related requirements are traced to data models, architecture-related requirements to architecture models, and functional requirements are initially traced to use case models and then to state models. Figure 17.26 shows an example of how a requirement can be linked to a model element in MagicDraw (e.g., to a transition in a state machine). These links can be useful both for evaluating that all the requirements have been reflected in the models, by showing what elements from different diagrams “implement” a given requirement, and for tracing the requirements to tests. Figure 17.13 presented a partial state model of the SUT in which various transitions have tags depicting requirements that they satisfy.

Once all requirements have been traced to UML elements, validation can be applied to verify that no requirement has been overlooked and that the models are ready for transformation.

17.7.2 Tracing requirements to tests

As mentioned in Section 17.4, Qtronic offers support for tracing requirements during test case generation. MATERA propagates the requirements linked to model elements into QML specifications. During the UML to QML transformation, the requirements are propagated from UML models (namely from state machine models) to QML state machine models. In the current case study, only those requirements that are attached to state transitions are collected from the UML state models. However, nothing prevents the collection of requirements from other UML models.

As such, the requirements are captured from UML transitions and placed on the corresponding transition in QML by using the requirement-statement (as explained in Section 17.5). Hierarchy can also be propagated from UML to Qtronic based on the numbering scheme of the requirement.

FIGURE 17.26
Tracing requirement to a model element.
Upon importing a QML model in Qtronic, requirements are displayed hierarchically in Qtronic’s UI (Figure 17.27) from where requirements-based test derivation can be pursued. In addition, the user can select which requirements should be covered by Qtronic during test case generation.

During the test case generation, Qtronic propagates the requirements to EAST test scripts via the scripting back-end discussed in Section 17.4. After running the generated test scripts in EAST, the output of the test run is stored in EAST test logs (see Listing 17.11). The test logs contain detailed information on test steps, executed methods, covered requirements, etc.

Listing 17.11
Excerpt from an EAST test log.

Log:20
1::1::Symbol: Statement ::Label: Statement :: Script Name: TC_TEST_2G_ms#1_2 :: When: Tue May 19 14:50:53:451 2009 :: Assign( "6 Authentication /1 The MSS must be able to authenticate MSs /1 Authentication of MSs must be supported in GERAN (2G) networks" ) = requirement ( "6 Authentication /1 The MSS must be able to authenticate MSs /1 Authentication of MSs must be supported in GERAN (2G) networks" )

Log:21
1::1::Symbol: Statement ::Label: Statement :: Script Name: TC_TEST_2G_ms#1_2 :: When: Tue May 19 14:50:53:451 2009 ::
Assign( "6 Authentication /3 Authentication is accepted when response is valid" ) = requirement ( "6 Authentication /3 Authentication is accepted when response is valid" )

Log: 22
1::1::Symbol: Statement ::Label: Statement :: Script Name: TC_TEST_2G_ms#1_2 :: When: Tue May 19 14:50:53:451 2009 :: Assign( "6 Authentication /1 The MSS must be able to authenticate MSs" ) = requirement ( "6 Authentication /1 The MSS must be able to authenticate MSs" )

17.7.3 Back-tracing of requirements

This approach is the opposite to the ones presented above and it consists of two parts. Firstly, statistical information about the test run regarding the number of passed/failed test cases, number of requirements covered, validated, etc., is collected in a report. The MATERA test evaluation module is used for analyzing the EAST test logs and generating the report (a simplified version of the report is shown in Figure 17.28). During the analysis,

---

**Statistics of the test output from EAST**

This HTML log has been automatically generated by a script.

The log contains statistical information about test output from EAST as well as requirements coverage information.

Generated on: Fri August 14 14:52:47 2009

**Requirements Information**

**Failed Requirements**

3.1.1 Location updated must be supported in GREAN (2G) networks
6.1.1 Authentication of MS's must be supported in GERAN (2G) networks
6.1.2 Authentication of MS's must be supported in UTRAN (3G) networks
7.1 The MSS must be able to cipher the communication with MS's
7.2 The ciphering procedure is initiated by the MSS after a successful authentication procedure

**Uncovered Requirements**

6.1 The MSS must be able to authenticate MS's

**Test Case Information**

The testrun generated a total of 10 testcases
Requirement 3.1.1 was present in 2 testcases
Requirement 6.1.1 was present in 1 testcases
Requirement 6.1.2 was present in 1 testcases
Requirement 7.1 was present in 1 testcases
Requirement 7.2 was present in 1 testcases

**Model Coverage Information**

The testrun covered 4 of 10 requirements in the model
The testrun covered 1 of 3 use cases in the model
The testrun covered 8 of 24 transitions in the model

---

**FIGURE 17.28**

Example of statistical information from a test run.
MATERA also collects information from the UML models in order to calculate how different parts of the model have been covered.

Secondly, information concerning failed test cases is collected in order to identify from which parts of the system model a given failed test was generated. In this way, one can see which parts of the system model are not in sync with the real implementation, and therefore with the stake holder requirements. The previously mentioned test evaluation module also retrieves information about requirements from the EAST test logs and generates a set of OCL constraints used by MATERA. The generated constraints have two purposes.

- To locate in the model the uncovered requirements by the test run, in which case their parent requirement will also be located.
- To locate the models and their elements that “implement” a given uncovered or failed requirement.

Listing 17.12 shows an example of generated OCL constraints.

**Listing 17.12**
Example of generated OCL rules

```ocl
-- Expressions for tracing uncovered or failed requirements in SysML requirements diagram:
not(Id = '3') and not(Id = '3.1') and not(Id = '3.1.1') and not(Id = '6') and not(Id = '6.1') and not(Id = '6.1.1') and not(Id = '7') and not(Id = '7.1') and not(Id = '7.2')

-- Expressions for finding uncovered or failed requirements placed on transitions:
not(clientDependency->exists(a | (a.supplier.name->includes('3') or a.supplier.name->includes('3.1') or a.supplier.name->includes('3.1.1') or a.supplier.name->includes('6') or a.supplier.name->includes('6.1') or a.supplier.name->includes('6.1.1') or a.supplier.name->includes('7') or a.supplier.name->includes('7.1') or a.supplier.name->includes('7.2')))) and clientDependency->notEmpty()
```

The generated constraints are automatically loaded and interpreted in the MATERA framework, which will highlight the targeted elements. Once loaded and executed in MATERA, the first expression will trace and highlight in the diagram editor the set of requirements to which it is attached (see Figure 17.29). Similarly, the second rule will highlight the elements to which the previous requirements were linked. For instance, Figure 17.30 shows a screenshot of MagicDraw displaying the transitions in the state model to which requirement 3.1.1 is linked.

### 17.8 Related Work

There is much literature on MBT in general and specific aspects in particular. For instance, Utting, Pretschner, and Legeard (2006) suggests a taxonomy for MBT, while in Hartman (2002) and Utting and Legeard (2006), the authors discuss MBT from a tools perspective, both academic and industrial. MBT techniques and tools have also been researched and developed in the context of the **Automated Generation and Execution of Test Suites for Distributed Component-based Software** (AGEDIS) (http://www.agedis.de) and **Deployment of Model-based Technologies to Industrial Testing** (D-MINT) (http://www.d-mint.org).
FIGURE 17.29

FIGURE 17.30
Back-tracing of requirements to a state machine model.
Applying MBT in the Telecommunication Domain

521

projects. The projects focused on the applicability of MBT techniques to industrial environments, and numerous publications and reports can be found on the web pages of each project. Both projects were applied to industrial case studies from different application domains, including telecommunications.

For instance, the authors describe in Born et al. (2004) a Model-driven Development and Testing process for Telecommunication Systems. A top-down approach based on CORBA is proposed in which the system is decomposed into increasingly smaller parts that can individually be implemented and tested. They make use of UML and profiles, in particular the Common Object Request Broker Architecture (CORBA) Component Model (CCM) (Object Management Group a), for specifying system models. These models are later on mapped onto software components using code generators, and executed within the CCM platform. Testing is also supported using specific UML profiles, in particular UML2 Testing Profile (U2TP), and by generators. The test design is tightly coupled with the system design so as to be able to reuse information provided in the system design, as soon as it becomes available. Testing is based on contracts built into different components and is derived from the models of the system. Test models are specified using U2TP and are used for the derivation of Testing and Test Control Notation (TTCN-3) tests. The TTCN-3 test are later automatically translated into executable Java byte code.

An approach similar to ours where several telecommunication case studies are evaluated against the Qtronic tool is discussed in Khan and Shang (2009). The main target of the work is to investigate the applicability of MBT in two telecommunication case studies. Another evaluation of MBT via a set of experiments run on Qtronic is described in Nordlund (2010). Both of those investigations draw the conclusion that MBT brings benefits in terms of automation and improved test coverage compared to traditional software testing.

Requirements traceability is a very popular topic in the software engineering and testing communities and has gained momentum in the context of automated test generation. However, as requirements change during the development life cycles of software systems, updating and managing traces have become tedious tasks. Researchers have addressed this problem by developing methods for automatic generations of traceability relations Hayes, Dekhtyar, and Osborne (2003), Spanoudakis et al. (2004), Cleland-Huang et al. (2005), Duan and Cleland-Huang (2006) by using information retrieval techniques to link artifacts together based on common keywords that occur in both the requirement description and in a set of searchable documents. Other approaches focus on annotating the model with requirements that are propagated through the test generation process in order to obtain a requirement traceability matrix Bouquet et al. (2005).

Work regarding requirements traceability similar to the MATERA approach is found in Bernard and Legereard (2007). There, the authors suggest an approach in which they annotate the UML system models with ad hoc requirement identifiers and use it to automatically generate a Traceability Matrix at the same time as the generated test cases. Their approach is embedded in the LEIROS Test Designer tool (SmartTesting). However, once the generated test cases are executed, requirements covered by tests are not traced back to the model.

17.9 Conclusions

Our experience indicates that MBT can already be used for productive testing in the telecommunication industry but there are some challenges. The challenges are related more to the tool integration than to MBT as a technology itself. If the challenges will be solved
Model-Based Testing for Embedded Systems

properly, MBT will become even more effective technology within the telecommunication domain.

Initial expectations on MBT were that modeling is difficult and requires a lot of expertise and competence. Our experiences indicate the opposite. Modeling as a competence seems to be fairly easy to learn. At the same time, modeling helps gain and retain an overview of the system behavior much easier than using programming and scripting languages.

Automated test generation is clearly significantly different in comparison to manual test scripting. In test generation, the emphasis is on the design phase instead of the implementation of test cases. Based on our experience, the modifications on the model are propagated into the test cases significantly faster using test generation than manually modifying test scripts. By the same token, automated test generation forces exploiting a different strategy on tracing details, such as requirements, throughout testing compared to manual test scripting. Content of manually written test cases tends to be static and hence the names of the test cases are typically human readable names that testing staff learn fairly quickly. Because of these aspects, mapping of the test scripts and the requirements is fairly static. In the case of automatic test generation, the content of test cases is more dynamic. The content of the test cases may change significantly between test generation rounds and because of that, the names of the test cases do not reflect the content of the tests. Because of this, tracking the details of testing requires a new strategy. In the developed methodology, requirements are used to track test coverage, progress of testing, etc. aspects. Using requirements for tracking is natural because the requirements are present in product development. Hence, use of the requirements for tracking does not necessitate any additional and artificial details for the methodology.

During the development of the methodology, a lot of effort was put into tool integration. Despite the fact that the models were described using standardized modeling languages, such as UML and SysML, it was not possible to create a UML model using one tool and open it with another tool. Compared to text-based programming and testing languages (e.g., C++ and TTCN-3), this is a significant drawback for UML modeling. Model transformations between the tools can be used to circumnavigate this problem. However, seamless use of the tools in, for example, model validation, refactoring, and traceability analysis, requires frequent interaction between the tools. Consequently, transformations will be performed frequently, and hence there would be a tool chain-induced performance penalty. In addition, implementing a number of model transformations increases the costs of the tool chain.

References


Applying MBT in the Telecommunication Domain


