

# Holistic Use of Analysis Models in Model-Based System Testing

by  
**Michael Mlynarski**

A dissertation submitted to the  
Faculty of Computer Science, Electrical Engineering, and Mathematics of the  
University of Paderborn

in partial fulfillment of the requirements for the degree of Dr. rer. nat.

Supervisors:  
Prof. Dr. rer. nat. Gregor Engels  
(University of Paderborn)  
Prof. Dr. rer. nat. Mario Winter  
(Cologne University of Applied Sciences)

Munich, Germany in September 2011



---

## ABSTRACT

---

Nowadays software testing techniques have to fulfill the requirements of growing complexity of evolving software systems. To handle the requirements current research is strongly interested in the field of model-based testing (MBT). While MBT is becoming the next generation of testing, using it in practice at the system test level two main problems arise: missing use of analysis models for testing purposes and low internal quality (as completeness, understandability, analysability or traceability) of automatically generated test artefacts. Those problems arise in the context of the model-driven development, when several inter-related modelling viewpoints of an analysis model (e.g. structure, behaviour and interaction) are not used while generating a test model. Such a holistic view on all viewpoints is needed to ensure the testability of the analysis model and to generate high-quality test artefacts from it. Therefore, a holistic usage of analysis models in functional model-based system testing is missing.

In order to tackle the mentioned problems, we introduce a novel model-based test specification process. It consists of four steps, which result in automatically generated high quality test cases. In the first two steps a test model is automatically generated from a manually annotated analysis model. Afterwards, the test model has to be manually reviewed and extended with test data. In the last step concrete test cases are automatically generated. In our approach we use the meta-models of a representative analysis model from an industry research project and the customized meta-model of the UML Testing Profile. The whole approach is prototypically implemented and an experiment providing empirical evidence for the improvement of the internal test quality and improvement of the modelling effort is conducted.

---

## ZUSAMMENFASSUNG

---

Die heutigen Techniken des Softwaretestens müssen der steigenden Komplexität von langlebigen Systemen gerecht werden. Derzeit erfreuen sich die Techniken des modellbasierten Testens (MBT) starkem Interesse sowohl in der Forschung als auch der Industrie. Obwohl MBT seit Jahrzehnten erforscht wird, gibt es praxisrelevante Probleme die bislang nicht gelöst worden sind. Im Kontext der modellbasierten Softwareentwicklung fehlt es an Ansätzen die Entwicklermodelle automatisch wiederverwenden, dabei aber die interne Testqualität im Sinne der Vollständigkeit, Verständlichkeit, Analysierbarkeit oder Verfolgbarkeit der automatisch generierten Testartefakte betrachten. Insbesondere fehlt auf der Stufe des funktionalen Systemtests in MBT eine ganzheitliche Sicht auf Analysemodelle die unterschiedliche Modellierungssichten (wie Struktur, Verhalten und Interaktion) bei der Generierung eines Testmodells betrachten würde. Wir nennen es die holistische Sicht und untersuchen dessen Anwendung in MBT und die dabei entstehende Korrelation zu Verbesserung der internen Testqualität sowie des sinkenden Modellierungsaufwands.

In dieser Arbeit stellen wir den holistischen MBT Testspezifikationsprozess vor, der die genannten Probleme behandelt. Der Prozess besteht aus vier Schritten, von denen zwei automatisch und zwei manuell durchgeführt werden. Im ersten Schritt wird das Analysemodell, welches durch Business-Analysten erstellt wurde, von Testdesignern auf Testbarkeit geprüft und mit einer Annotationssprache prorsiert. Im zweiten Schritt wird mit Hilfe mehrerer Algorithmen ein Testmodell generiert und die erreichte Modellabdeckung des Analysemodells automatisch berechnet. Das Testmodell wird im dritten Schritt manuell untersucht und um Testdaten ergänzt. Am Ende werden konkrete Testfälle in platformsspezifischen Formaten automatisch generiert. In dieser Arbeit wird ein repräsentatives Analysemetamodell aus einem Industrieforschungsprojekt verwendet, sowie ein angepasstes Metamodell des UML Testing Profiles. Der holistische Ansatz wurde prototypisch implementiert. Für die Evaluierung wurde ein Experiment durchgeführt, welcher die empirische Evidenz für die Verbesserung der internen Testqualität sowie die Minderung des Modellierungsaufwandes belegt.

---

## ACKNOWLEDGMENT

---

First, I would like to thank Gregor Engels, the best phd supervisor and mentor ever. Without his patience and experience-based advisory, I would not be able to write this thesis in three years while at the same time working full time for Capgemini. Each session made me wiser and motivated me to put more effort into my research.

Second, my dear wife Eva. Only she knows how many sleepless nights and weekends were necessary to write and submit my thesis. Thank you for your great support. Especially in those days were working / writing was the last thing I wanted to do.

To my family, which supported me from Poland. Thank you for always believing in me and for making me the person who I am.

I also want to thank Waltraut and Heinz Gelhoit for enabling my computer science studies in Paderborn back in 2003. Up to my phd defense they supported me in various ways for which I am very thankful.

Finally, the best phd mates Baris Güldali, Andreas Wübbeke, Yavuz Sancar in Paderborn and Daniel Méndez Fernández in Munich. Through the (sometimes) never-ending, late night discussions I have learned that questioning my own and others research is always the best way to succeed!



---

## CONTENTS

---

<b>I</b>	<b>PROBLEM DEFINITION AND RELATED WORK</b>	<b>1</b>
<b>1</b>	<b>INTRODUCTION</b>	<b>3</b>
1.1	Problem statement . . . . .	9
1.2	Contribution . . . . .	14
1.2.1	Methodology . . . . .	15
1.2.2	Practice . . . . .	17
1.3	Publications . . . . .	17
1.4	Outline . . . . .	19
<b>2</b>	<b>DEFINITIONS AND PRELIMINARIES</b>	<b>21</b>
2.1	Dynamic Software Testing . . . . .	21
2.1.1	Process and Artefacts . . . . .	24
2.1.2	Test roles . . . . .	26
2.1.3	Meta-Model . . . . .	27
2.1.4	Risk-Based Testing . . . . .	28
2.2	Model-Based Testing . . . . .	29
2.2.1	Definition . . . . .	29
2.2.2	Methodological Issues . . . . .	30
2.2.3	Process and Artefacts . . . . .	32
2.2.4	Test selection algorithms . . . . .	34
2.3	Test Modelling Language . . . . .	38
2.3.1	UML Testing Profile . . . . .	40
2.3.2	Artefact Meta-Model . . . . .	42
2.4	Modelling Business Information Systems . . . . .	45
2.4.1	General definitions . . . . .	45
2.4.2	Motivation . . . . .	47
2.4.3	Representative industry modelling approach . . . . .	48
2.4.4	Running example "Gabi's Ski School" . . . . .	49
2.4.11	Artefact Meta-Model . . . . .	63
2.5	Model Transformations . . . . .	65
2.5.1	Definitions . . . . .	66
2.5.2	Categorization . . . . .	66
2.5.3	Traceability Issue . . . . .	68
2.5.4	Model Transformation Languages . . . . .	69
2.6	Summary . . . . .	71
<b>3</b>	<b>RELATED WORK</b>	<b>73</b>
3.1	Evaluation criteria . . . . .	74
3.1.1	UML for system modelling . . . . .	75
3.1.2	Modelling viewpoints . . . . .	76

3.1.3	Integrated interaction viewpoint . . . . .	77
3.1.4	Model relations . . . . .	78
3.1.5	UML for test modelling . . . . .	78
3.1.6	Test Model . . . . .	79
3.1.7	Developer Model . . . . .	79
3.1.8	Understandability . . . . .	81
3.1.9	Analysability . . . . .	82
3.1.10	Completeness . . . . .	82
3.1.11	Traceability . . . . .	83
3.1.12	Case study and tool support . . . . .	83
3.2	Identified related work . . . . .	84
3.2.1	Generation from system models . . . . .	84
3.2.2	Generation from several modelling view- points . . . . .	88
3.2.3	Generation from test models . . . . .	91
3.2.4	Generation of test models from developer models . . . . .	93
3.2.5	Generation using model relations . . . . .	95
3.2.6	Generation from GUI models . . . . .	97
3.2.7	Test case quality attributes . . . . .	99
3.3	Summary . . . . .	101
<b>II</b>	<b>APPROACH AND EVALUATION</b>	<b>103</b>
<b>4</b>	<b>META-MODEL ALGEBRA</b>	<b>105</b>
4.1	Motivation . . . . .	106
4.2	Definitions . . . . .	108
4.3	Algebra Meta-Model . . . . .	109
4.4	Related work . . . . .	110
4.5	Meta-Model Properties . . . . .	112
4.5.1	Traceability . . . . .	112
4.5.2	Modelling viewpoints . . . . .	113
4.5.3	Model relation . . . . .	114
4.5.4	Structural mapping . . . . .	114
4.5.5	Traversability . . . . .	115
4.6	Algebra Operations . . . . .	116
4.6.1	transform . . . . .	118
4.6.2	select . . . . .	119
4.6.3	extract . . . . .	119
4.6.4	cover . . . . .	119
4.7	Algebra Specification Language . . . . .	120
4.8	Algebra Instantiation . . . . .	122
4.9	Applicability discussion . . . . .	123
4.10	Summary . . . . .	124
<b>5</b>	<b>MODEL-BASED TEST SPECIFICATION PROCESS</b>	<b>125</b>
5.1	Requirements . . . . .	125



5.2	Approach overview . . . . .	126
5.2.1	Process . . . . .	126
5.2.2	Artefacts . . . . .	129
5.3	Step 1. Analyze and annotate the Analysis Model . . . . .	130
5.3.1	Manual testability checks . . . . .	132
5.3.2	Test prioritization through model annotation . . . . .	135
5.4	Step 2. Generate Basic Test Model . . . . .	138
5.4.1	Test Case Selection . . . . .	139
5.4.2	Automated Model Analysis . . . . .	146
5.4.3	Model Transformations . . . . .	154
5.4.4	Model Coverage Measurement . . . . .	165
5.5	Step 3. Extend the Basic Test Model . . . . .	179
5.5.1	Basic vs. extended test model . . . . .	180
5.5.2	Manual extension process . . . . .	181
5.6	Step 4. Generate Concrete Test Cases . . . . .	186
5.6.1	Excursion: Constraints in test data . . . . .	187
5.6.2	Test Data Selection . . . . .	189
5.6.3	Platform-specific test case generation . . . . .	192
5.7	Summary . . . . .	194
6	EVALUATION . . . . .	197
6.1	Evaluation planning . . . . .	197
6.1.1	Evaluation goals . . . . .	198
6.1.2	Experiment design . . . . .	200
6.1.3	Setting . . . . .	202
6.1.4	Null Hypotheses . . . . .	203
6.1.5	Alternative Hypotheses . . . . .	203
6.2	Tool support . . . . .	204
6.2.1	Motivation . . . . .	204
6.2.2	Test Model Generator . . . . .	204
6.2.3	Test Case Generator . . . . .	206
6.2.4	Used technology stack . . . . .	207
6.2.5	Used environment . . . . .	208
6.3	Experiment "Gabi's Ski School" . . . . .	208
6.3.1	Input model . . . . .	208
6.3.2	Results . . . . .	210
6.3.3	Interpretation of results . . . . .	210
6.4	Discussion of the results . . . . .	220
6.4.1	Internal validity . . . . .	220
6.4.2	Construct validity . . . . .	221
6.4.3	External validity . . . . .	223
6.5	Summary . . . . .	224
7	SUMMARY AND OUTLOOK . . . . .	227
7.1	Summary of Contributions . . . . .	228
7.2	Outlook . . . . .	232
7.3	Final statement . . . . .	234

A	EXPERIMENT RESULTS	237
A.1	Understandability questionnaires . . . . .	237
A.2	Coverage reports . . . . .	238
A.2.1	Report for Set 1 and 2 . . . . .	240
A.2.2	Report for Set 5 and 6 . . . . .	244
	Bibliography	248

---

## LIST OF FIGURES

---

Figure 1	Fundamental Test Process . . . . .	4
Figure 2	Model-driven engineering process . . . . .	5
Figure 3	Logical test case derived from analysis model . . . . .	7
Figure 4	Model-Based Testing Scenarios . . . . .	12
Figure 5	Test independency problem . . . . .	13
Figure 6	Contribution of the phd thesis . . . . .	16
Figure 7	Fundamental test process after [SL05] . . . . .	23
Figure 8	Meta-model for functional software testing . . . . .	28
Figure 9	Artefact meta-model for the MBT process . . . . .	33
Figure 10	Coverage criteria subsumption . . . . .	36
Figure 11	Test model structure . . . . .	39
Figure 12	Artefact meta-model for the UTP . . . . .	41
Figure 13	Example for a logical test case . . . . .	43
Figure 14	Example of a test architecture . . . . .	44
Figure 15	Example of test data viewpoint . . . . .	45
Figure 16	Modelling ontology . . . . .	46
Figure 17	Example of the use case overview . . . . .	53
Figure 18	Example of a use case . . . . .	54
Figure 19	Example of a logical data type model . . . . .	58
Figure 20	Example of dialog layouting . . . . .	60
Figure 21	Example of dialogs behaviour . . . . .	61
Figure 22	Example of conceptual components . . . . .	62
Figure 23	Analysis meta-model . . . . .	64
Figure 24	Evaluation criteria . . . . .	74
Figure 25	Test Quality Attributes . . . . .	81
Figure 26	Evaluation Table . . . . .	85
Figure 27	Approach characteristics . . . . .	86
Figure 28	TOTEM meta-model from [BL02, p.30] . . . . .	90
Figure 29	Meta-Model Algebra . . . . .	107
Figure 30	Meta-Model of the Meta-Model Algebra . . . . .	109
Figure 31	Meta-model property <i>traceability</i> . . . . .	113
Figure 32	Meta-model property <i>model relation</i> . . . . .	114
Figure 33	Meta-model property <i>structural mapping</i> . . . . .	115
Figure 34	Meta-model algebra operation . . . . .	116
Figure 35	Algebra specification language . . . . .	121
Figure 36	Example for the <i>transform</i> operation . . . . .	121
Figure 37	Algebra instantiation process . . . . .	122
Figure 38	Solution requirements . . . . .	127

Figure 39	Model-Based Test Specification Process . . .	129
Figure 40	Artefacts meta-model . . . . .	130
Figure 41	Analyze and annotate test basis . . . . .	131
Figure 42	Annotation example . . . . .	137
Figure 43	Generate basic test model . . . . .	139
Figure 44	Algebra operation <i>select</i> . . . . .	141
Figure 45	Main algorithm for the test selection . . . .	142
Figure 46	Recursive algorithm traverse . . . . .	143
Figure 47	Subalgorithm traverse decision . . . . .	145
Figure 48	Path selection example . . . . .	147
Figure 49	Relevant model relations . . . . .	149
Figure 50	Algebra operation extract . . . . .	150
Figure 51	Algorithm for Automated Model Analysis .	152
Figure 52	Relations with the mapping table . . . . .	155
Figure 53	Relations modelling viewpoints . . . . .	157
Figure 54	MOF-based overview . . . . .	159
Figure 55	Algebra operation transform . . . . .	160
Figure 56	Algorithm for Model Transformation . . . .	161
Figure 57	Path example . . . . .	163
Figure 58	LTC example . . . . .	164
Figure 59	Test architecture example . . . . .	164
Figure 60	Coverage problem . . . . .	166
Figure 61	Trace meta-model . . . . .	167
Figure 62	Algebra operation cover . . . . .	169
Figure 63	Model coverage algorithm . . . . .	171
Figure 64	Hierarchy of the model coverage metrics . .	174
Figure 65	Coverage report example . . . . .	176
Figure 66	Coverage report example . . . . .	177
Figure 67	Extend basic test model . . . . .	180
Figure 68	Missing LTC information example . . . . .	183
Figure 69	Linking test data with LTC . . . . .	185
Figure 70	Generate CTC . . . . .	186
Figure 71	Test data combination example . . . . .	190
Figure 72	SimpleCombination algorithm . . . . .	191
Figure 73	Thesis mapping . . . . .	195
Figure 74	Architecture of the Test Model Generator . .	205
Figure 75	Architecture of the Test Case Generator . .	207
Figure 76	Comparison of the global model coverage .	211
Figure 77	Logical data type coverage . . . . .	213
Figure 78	Complete and incomplete LTC . . . . .	215
Figure 79	Annotated use case <i>Book_Attendee_On_Course</i>	219
Figure 80	Reached contributions of this phd thesis . .	228
Figure 81	Questionnaire template . . . . .	239
Figure 82	Global coverage Set 1 and 2 . . . . .	240
Figure 83	Use case coverage Set 1 and 2 . . . . .	241

Figure 84	Dialog Coverage Set 1 and 2 . . . . .	242
Figure 85	Logical data type coverage Set 1 and 2 . . . . .	243
Figure 86	Global coverage Set 5 and 6 . . . . .	244
Figure 87	Use case coverage Set 5 and 6 . . . . .	245
Figure 88	Dialog coverage Set 5 and 6 . . . . .	246
Figure 89	Logical data type coverage Set 5 and 6 . . . . .	247

---

## LIST OF TABLES

---

Table 1	Coverage of modelling viewpoints . . . . .	9
Table 2	New MBT activities in the FTP . . . . .	32
Table 3	Structure of the running example . . . . .	50
Table 4	Model transformation languages . . . . .	70
Table 5	Mapping table between meta-models . . . . .	156
Table 6	Model coverage measurement goals . . . . .	172
Table 7	Test adapter mapping . . . . .	194
Table 8	Evaluation goal 1 . . . . .	198
Table 9	Evaluation goal 2 . . . . .	199
Table 10	Evaluation goal 3 . . . . .	199
Table 11	Sets definition . . . . .	202
Table 12	Experiment's analysis model . . . . .	209
Table 13	Experiment results . . . . .	210
Table 14	Time effort in different MBT-scenarios . . . . .	216
Table 15	Interview answers - complete LTC . . . . .	237
Table 16	Interview answers - incomplete LTC . . . . .	238

---

## ABBREVIATIONS

---

MBT Model-Based Testing  
MDD Model-Driven Development  
MDA Model-Driven Architecture  
MMA Meta-Model Algebra

UML	Unified Modeling Language
UTP	UML Testing Profile
BIS	Business Information System
FTP	Fundamental Test Process
SUT	System Under Test
OCL	Object Constraint Language
LTC	Logical Test Case
CTC	Concrete Test Case
ISTQB	International Software Testing Qualifications Board
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
SRS	Software Requirements Specification

## Part I

### PROBLEM DEFINITION AND RELATED WORK





---

## INTRODUCTION

---

Quality of products each one of us is using plays an important role. We do not like to use products which fail or lack usability, performance or functionality. This can be also applied to software products. As we are surrounded by software, users expect it to be of high quality. On the other hand, the complexity of software continuously grows. Software systems evolve over time, which results in high maintenance. This leads to a very important problem in present software engineering: How to develop and maintain complex software products and guarantee their high quality?

*Quality of evolving software*

Since decades (see [Moo56] or [GG75]) the analytical methods of software engineering have been providing solutions for the mentioned problem by analyzing the quality reached during the development process. The most known analytical method is the dynamic software testing where test cases are designed, executed and their results are evaluated with regard to bugs found during the execution. The three mentioned steps can be detailed into several phases which together group a test process. A good example of an industry test process is the ISTQB Fundamental Test Process (FTP) [SL05] shown in Figure 1. The FTP is divided into several phases like test planning and controlling, test analysis and design, test implementation and execution, test evaluation and test closure. As highlighted in Figure 1, we will focus on test analysis, design and implementation phase and use the synonym *test specification* for the three mentioned test phases.

*Industrial test process*

Dynamical testing visualizes the software quality in a project. Reaching high test quality with respect to the test process and its artefacts is needed to provide a reliable visualization of the software quality [Wag06]. High test quality is mostly influenced by the test design phase of the FTP as shown by [Bin99] or

*Test quality and software quality*

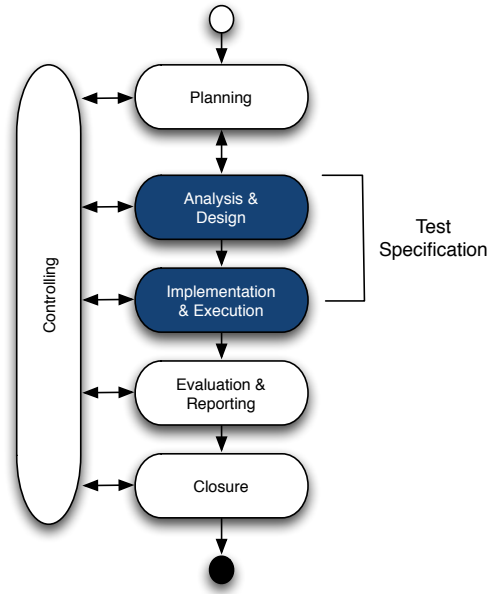
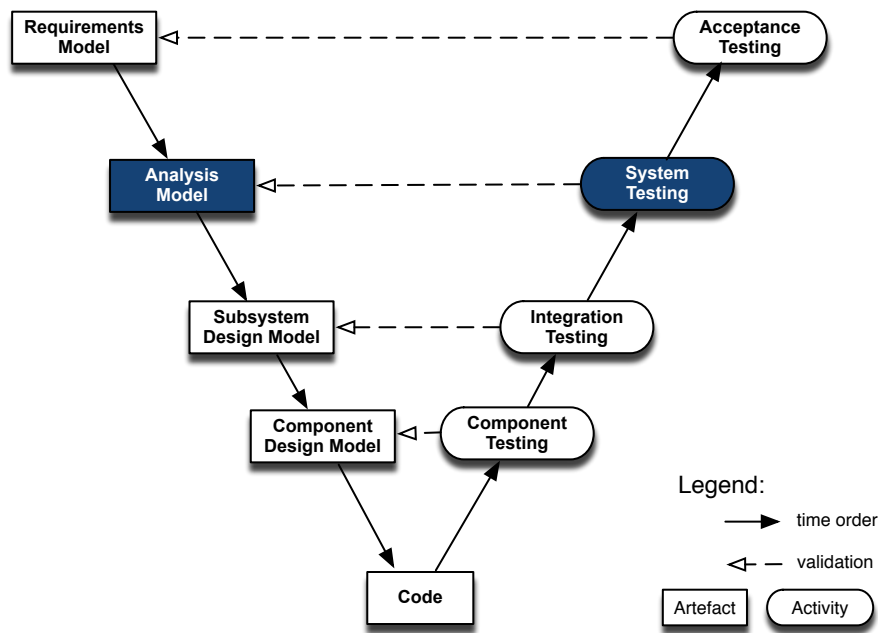


Figure 1: Thesis focus within the simplified FTP

[Lig09]. The quality of test cases designed in this phase can be described by external (like fault-detection rate or reached coverage) and internal (completeness, understandability, analysability, etc.) quality attributes [ZVS<sup>+</sup>07]. Most of the approaches in the current literature in the domain of software testing focuses on the external quality attributes as shown in surveys like [Wag06, DNSVT07, DNSV<sup>+</sup>08, MN10, GECMT05, DM03, McMo4]. In this thesis we focus on the **internal test quality**, which is crucial for the maintainability of test artefacts in long-term software engineering projects.

*Model-driven  
development and  
model-based testing*

In a testing process as the FTP, test designers use the textual requirements and system specifications created by other project members to manually derive test cases. In a model-driven development process [Obj03] the mentioned documents are replaced by models created with more precise languages like the Unified Modeling Language (UML) [Obj09]. By using algorithms which analyze the models, test designers are able to automate a great part of the test design task. Especially the automatic test case generation influences its efficiency as shown in [PPW<sup>+</sup>05, NFTJ06] or [Wei09]. If test cases are automatically generated from models, then we speak of **model-based testing (MBT)** [UL07].



**Figure 2:** Adjusted V-Model of the model-driven software engineering process

In this thesis we understand MBT as the automation of the test design phase of the FTP.

The FTP can be conducted on several test levels as component, integration, system and acceptance testing [SL05]. The design of test cases for each test level is based on artefacts created by the constructive disciplines of the software engineering process as requirements engineering, business analysis or design [Kru03]. In the context of model-driven development, those disciplines result in several models. This connection is defined in the development model adjusted to the original V-Model [Boe79], which is shown in Figure 2.

As stated by several literature surveys (see [DNSVT07, DNSV<sup>+</sup>08, DM03, MN10]) model-based testing can achieve great benefit on this testing level, since system testing is the most complex and costly type of testing [PKS02]. In this thesis, we focus on the system testing level. Since we do not cover non-functional tests (as performance, usability, etc.), we focus our work solely on functional system tests.

*Focus on functional  
model-based system  
testing*

As shown in Figure 2, system testing is primary based upon the information from the analysis model. Within this thesis we understand the **analysis model** as a Software Requirements Specification (see SRS in IEEE 830 [IEE98]) which is specified by using a modelling language (like the Unified Modelling Language [Obj09]) according to a predefined modelling approach. The analysis model specifies the conceptual solution from different viewpoints and does not contain technical or architectural information. Based on this model further design models are created and at the end code is implemented manually or partially generated from the design models [Obj03].

*Example of an  
analysis modelling  
approach*

In this thesis we inspect the usage of analysis models for model-based system testing. In an industrial research project conducted within this phd thesis, we have used a real-life analysis model for the domain of business information systems to develop a test design method for manual testing. We use the collected observations to show the usage of this exemplary analysis model for model-based system testing. The underlying modelling approach was introduced by Salger et al. in [SSE09] and is used by Capgemini Technology Services a custom software development and IT consulting company. According to [SSE09] the analysis model consists of the following main viewpoints:

- *Structure* - defines the functional decomposition of the system into conceptual components and its data model
- *Behaviour* - defines the system's behaviour
- *Interaction* - defines the user interfaces and their usage

*Different modelling  
viewpoints*

Each of the mentioned modelling viewpoints can be described with UML diagrams. The modelling approach gives guidelines for choosing the appropriate diagram. In the top of Figure 3 we show an example of models describing the structure, behaviour and interaction with the system. The models should describe part of a fictive ski course booking system which will be used as a running example in this thesis. The behaviour is modelled with a use case diagram consisting of three use cases. The first one *Create course* is refined with an activity diagram. The structure model consists of a class diagram which defines the data model consisting of three entity types. Last, the interaction model defines the static layout (proprietary notation) and dynamic behaviour (using activity diagrams) of the dialogs which have to be implemented.

As shown with the dotted edges in Figure 3 the use case is related to the dialogs' behaviour, namely the dialog action *Search Course*. This action is related to certain dialog elements and further to the elements of the underlying data model. The relations for this analysis model are clearly defined and provided with an meta-model<sup>1</sup>. Each relation from the meta-model results in a link between models by using the UML concept of Association [Obj09, p.39]. This is done by business analysts while modelling.

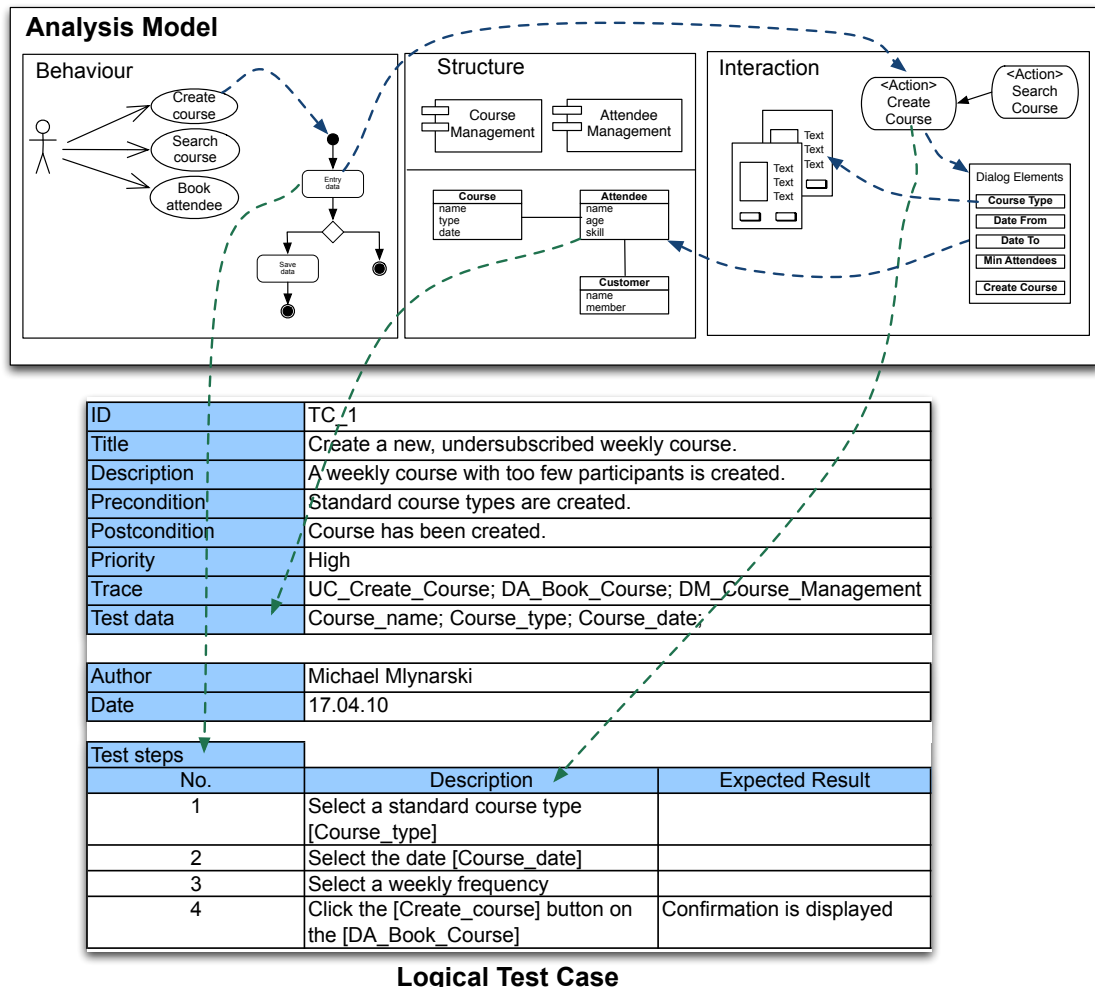


Figure 3: Logical test case derived from analysis model

Knowing how a typical analysis model looks like, the question is how to use it for model-based system testing? The automation of

<sup>1</sup> The analysis meta-model describes all artefacts, which have to be created and the relations between them. It is also called the domain meta-model.

test design in MBT by using sophisticated algorithms can be applied primarily to the behaviour models [DNSVT07]. The output of the automatic test generation are logical test cases, which do not contain concrete values for input data and expected results [SL05]. An example of a logical test case is shown in the bottom part of Figure 3. It consists of several attributes as title, description, pre- and post-condition, logical test data definition and step description. During the test generation, the attribute values have to be derived from different viewpoints of an analysis model to reach high internal test quality in terms of completeness, understandability and analysability.

*Internal test quality  
of test cases*

Let us consider the information flow showed with the dotted edges from the analysis model to the test case in Figure 3. The test steps, pre- and post-conditions needed to execute the test case are derived from the use case and activity diagrams. But this description is incomplete, because important information about the usage of the user interface is missing. That is why the description of the test steps has to contain names of used dialog elements from the static and dynamic interaction models. Also the action to be triggered on the dialog for each test case step has to be derived. This way, the complete description of the test steps has to be derived from two different modelling viewpoints. Further, the logical test data definition is partially derived from the data model, which is related to the static and dynamic interaction models. This kind of information could not be derived only from the use case or only from the dialog model, because their modelling purpose is not to define the data structure, but the behaviour or interaction. The identification of the appropriate context through model relations of the analysis model is needed. **The conclusion is that information needed to specify logical test cases which are complete, understandable and analyzable is spread across several parts of the analysis model.**

*Model relations are  
crucial for test  
design!*

The relations between single models enable the identification of related model elements. For example the identification of information about the user interface in the test case steps (as dialog elements to be filled with content or buttons to be triggered) is done by navigating through relations between the behaviour and interaction models. The relations between modelling viewpoints are created by business analysts according to the meta-model definition.

## 1.1 PROBLEM STATEMENT

This phd thesis thematises the following research questions:

1. How to use all three modelling viewpoints (structure, behaviour and interaction) to generate high-quality test cases?
2. How to use analysis models for test generation while guaranteeing independency from developers in order to still find faults in the analysis model?
3. How to measure the test coverage of all modelling viewpoints of the analysis model?

The first question consider the main research problem of this thesis, namely the *missing holistic test view on analysis model* in current model-based testing approaches. The second question depicts a related problem called *use of analysis models*. The third question concerns the related problem of *test coverage of analysis models*. In the subsequent paragraphs, we will briefly introduce all three research problems and define the phd hypothesis of this work.

### Holistic Test View on the Analysis Model

According to the survey from Dias Neto et al. [DNSVT07] the majority of model-based testing approaches uses UML diagrams for test generation. The authors identified several types of UML diagrams used in 47 UML-based approaches. Those diagrams cover only two modelling viewpoints as shown in Table 1.

**Table 1:** Approach coverage of modelling viewpoints from [DNSVT07]

Modelling Viewpoint	Approaches and UML diagrams
Structure	19 with UML class diagrams
Behaviour	27 with UML state machine diagrams, 19 with UML sequence diagrams, 11 with UML use case diagrams, 9 with UML activity diagrams

In the literature survey performed within this phd thesis (see Chapter 3), we have also found approaches like [MSP01, BM10, NRP05, SS97] or [BBWo6] covering the interaction viewpoint with single diagrams describing the system's GUI. To the best of our knowledge, we have found that most of the known approaches can generate logical test cases only from one model viewpoint, namely the structure, behaviour or interaction view.

*Refinement of  
analysis model for  
test generation*

Regarding the description of the information flow from Figure 3, the single models used in most approaches have to contain information normally spread across the structure, behaviour and interaction model viewpoints. In this case the single models (as in [BL02], [VLH<sup>+</sup>06] or [HNo4]) were explicitly created for test purposes and contain all test-related information from all modelling viewpoints. Otherwise the analysis model has to be manually refined by test designers in order to use approaches as [HVFR05] or [NFTJ06] for test case generation. A detailed literature survey can be found in Chapter 3 of this thesis.

*Holistic view is  
needed to guarantee  
the test case quality!*

As the test design phase for system testing is based on the analysis model, test designers have to use several and not only one modelling viewpoint for designing test cases. There is a strong need for a test view throughout the whole model landscape including structure, behaviour and interaction model viewpoints together with the relations between those models in current MBT approaches. A test view considering the three model viewpoints during test generation is called **holistic**. The holistic view is needed, because only then all information spread across modelling viewpoints, which is relevant for test design can be collected. The current approaches for model-based testing concentrate on single modelling viewpoints, which contain only partial information. This way low-quality test cases are generated and missing information from the analysis model has to be extended manually.

The missing holistic view influences several quality attributes of the internal test quality. In thesis, we use the test quality models introduced by Zeiss et al. in [ZVS<sup>+</sup>07], Voigt et al. in [VGE08] and Deng et al. in [DSWO04]. The following quality attributes are related to the usage of a holistic view in MBT:

- completeness (*Does the test case contain all information needed for test execution?*)



- understandability (*Is it clear what is the purpose of the generated test case?*)
- analysability (*Can the test case be diagnosed for deficiencies?*)
- traceability (*Is the test case traceable to the elements of the analysis model?*)

Based on our industry observations (see [MGSE09] and [Mly10]) and literature research, we identify the following research question: *How to use a holistic view on analysis models in model-based system testing?*

### Use of Analysis Models for Test Generation

So far, we tackled the the holistic usage of analysis models for creating logical test cases for system testing. Those models contain several modelling viewpoints, which can be used for the case design. On the other hand those models were created by business analysts to specify the high level functionality of the system. This specification does not contain detailed information about the concrete data for input and output of each system function [MJV<sup>+</sup>10]. That is why only logical test cases can be derived from the analysis model.

Several approaches for model-based system testing as [UL07, BBW06, DM03, MSP01] propose to create a separate test model from which concrete test cases (with input and output data) can be generated. Compared to analysis models, those test models should contain all information needed for the generation of concrete test cases. There are several possibilities of how this test model can be created. Pretschner and Philipps define in [PP05] four scenarios for test model creation. The first one called *common model* uses a single model for generating test cases and code. The second one called *manual modelling* assumes that the test model is created manually from the requirements and system specification. The third one called *separate models* proposes to create a test model directly from requirements and to omit developer models (like our analysis model) created by other teams. In the last one called *automatic model extraction* the test model is reengineered from the existing code. Figure 4 depicts the four scenarios.

*Scenarios for test  
model creation*

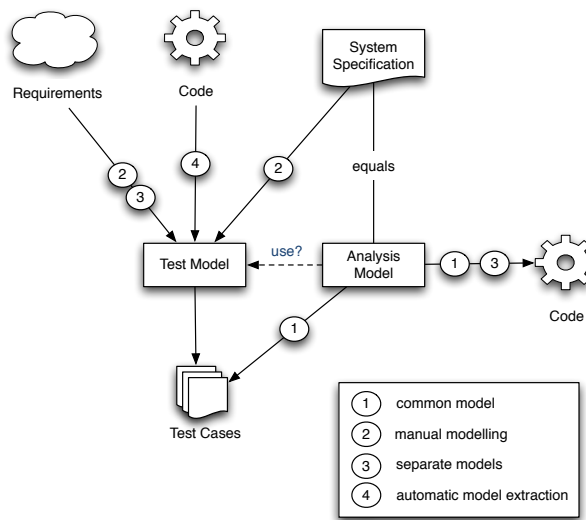


Figure 4: Model-Based Testing Scenarios

The main difference between those scenarios lies within the level of independency (also called redundancy in [PP05]) needed for test design. This is based on the observation that different or additional information with respect to the development specification artefacts are given in the test model. The independency level in test models varies depending on how those are created. If development models are directly used as test models, there is low independency and therefore no additional information is given. If test models are newly created from user requirements and if they are independent of the development models the amount of additional or different information is high. A test model created by using analysis models jointly with independent test information is shown in Figure 5. Independency has a direct impact on the fault-detecting capability of a MBT approach. The more independency a test model contains, the more requirement-related errors (including faults in the analysis model) can be detected and high requirements coverage can be reached. This way, it has a direct impact on the test quality, but also on the needed effort [GMS10, BGM10].

*Independency from  
developers needed to  
find faults*

On the other hand business analysts create an analysis model which contain several information, which could be used for test design. In practice the use of existing models is strongly accepted because of lowering the test design effort. The acceptance

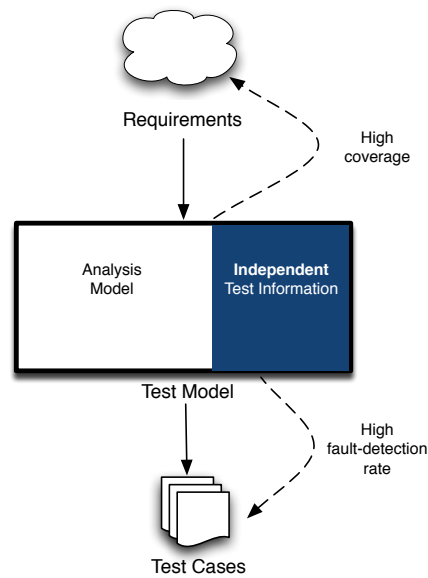


Figure 5: Independent test information in the test model

increases even more if the usage can be automated. Regarding the independency problem and the industry need for usage of existing models, the following question for the arises: *How to use existing analysis models for test case generation, while ensuring high independency from those models?*

#### Test Coverage of Analysis Model

The most important attribute describing the test quality in the test design process is the reached test coverage. Coverage can be measured according to the code tested by test cases [SL05]. Studies as [RS05] show that very high code coverage (for example near 100%) does not necessary guarantee a high fault detection rate. On the other hand, strong code coverage criteria as branch or path coverage require high effort to reach all possible code parts. The other kind of coverage criteria concentrate on the basis for test design. In the case of model-based system testing, the basis is the analysis model (see Figure 2). Here, we are interested in the measurement of the model coverage reached by the test case generation process. As several modelling viewpoints have

*Code vs. model  
coverage*

to be used for test generation, the coverage metrics have to consider this.

*Need for combined  
coverage criteria*

The state-of-the-art coverage measurement concentrates on single modelling viewpoints as shown in the survey from McQuil-  
lan and Power [MQP05]. In the context of UML there exist sev-  
eral coverage criteria for each of the UML diagrams (see [MQP05,  
AFGC03, UL07]). Because of the fact that the most known MBT  
approaches use single viewpoints for test generation those cov-  
erage criteria are applied on single UML diagrams. In order to  
measure the test quality of a holistic model-based testing ap-  
proach, the known coverage criteria have to be combined. Those  
coverage criteria have to consider the information flow needed to  
fully specify logical test cases (see earlier discussion on this topic  
in this section), not only the aspect of test case selection. That is  
why the following research question for the second subproblem  
arises: *How to measure the model coverage for a holistic model-  
based system testing approach based on analysis models?*

*Phd hypotheses*

Summarizing the research questions described above, we define  
the following phd hypotheses:

- H1 **High internal test quality** with respect to completeness,  
analysability, understandability and traceability of test mod-  
els derived from analysis models can be achieved by using  
a holistic view.
- H2 The holistic view has an impact on the **reached coverage**  
of the analysis model.
- H3 The **efficiency** of the **test generation process** can be im-  
proved by automatically using the analysis model to create  
a test model.

## 1.2 CONTRIBUTION

This phd thesis answers the research questions defined in the  
last subsection. The solutions presented in this document should  
contribute to the field of software testing research. We categorize  
this contribution into the following categories:

- Methodology
- Practice

The contribution categories are not orthogonal and have to be seen as a whole.

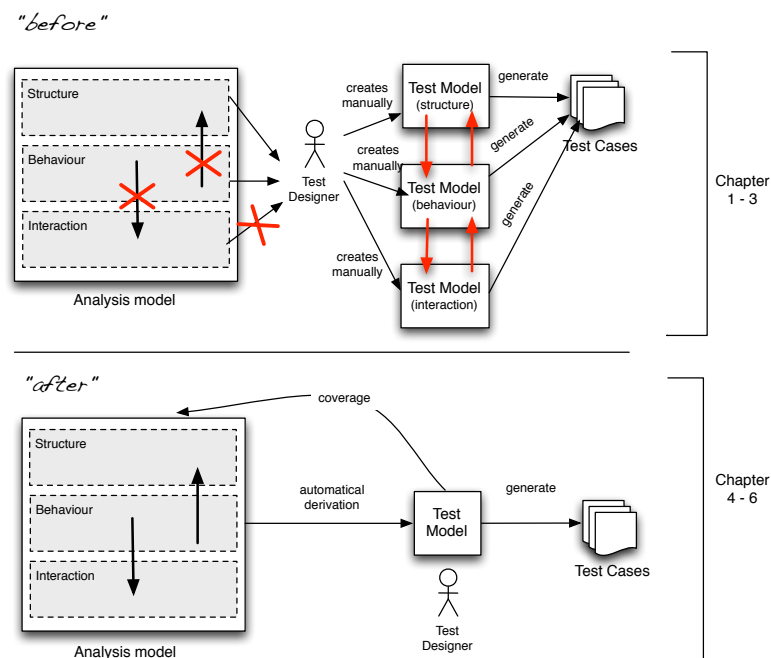
### 1.2.1 Methodology

The most important contribution is the introduction of a holistic approach for model-based system testing in a model-driven development process. To better explain the planned contribution we compare the current and future approach in Figure 6. The current approaches for model-based system testing consider only the structure and behaviour viewpoints of the analysis model. The missing use of the interaction viewpoint is depicted with the red struck out arrow. The use of analysis models is done manually by creating separate test models. The relations between the different viewpoints are incorporated manually by test designers. This is depicted with the red struck out arrows in the analysis model and the red arrows in the test models (see upper part of Figure 6). Compared to the current situation the new holistic approach considers all viewpoints, with the model relations (according to their meta-model definition) in an automated manner. This is shown with the filled arrows within the analysis model in the bottom part of the figure. To avoid the missing test independency problem mentioned in Section 1.1, test designers review the automatically generated test model and extend it with additional information. The reached model coverage is measured automatically.

The research approach presented in this thesis will show how the internal test quality, in terms of completeness, understandability, analysability and traceability together with the reached test coverage of the automated test design activity, can be improved. The improvement of the test design process efficiency through the automatic use of analysis models for test purposes will be shown. Further, the application of combined model coverage criteria for a holistic model coverage measurement will be defined.

The contribution of the holistic approach can be decomposed into the following contribution points:

1. **Improvement of the internal test quality** - The generated test cases are of high quality with respect to completeness, analyzability, understandability and traceability.



**Figure 6:** Contribution of the phd thesis explained by comparing the current and future approach

2. **Use of analysis models for test model generation** - Rather than creating test models manually, this task is fully automated.
3. **Use of several modelling viewpoints** - Different as in many state-of-the-art approaches, several modelling viewpoints (structure, behaviour and interaction) with the according UML diagrams types of the analysis model are used. This way, a high coverage of the analysis model is reached.
4. **Use of the integrated interaction viewpoint** - As shown in Figure 6 the interaction modelling viewpoint (especially models describing the GUI) integrated into the analysis model together with structure and behaviour models are used. This is not the case in the state-of-the-art approaches.
5. **Use of model relations created by business analysts** - Rather than creating the model relations manually after the test model is created, the relations modeled by business analysts are used.

6. **Holistic model coverage measurement** - the coverage of several modelling viewpoints is measured by using combined coverage criteria.

### *Requirements*

The research approach will be clearly defined by means of process descriptions with input and output artefacts. The process descriptions have to be understandable for software testers. Each proposed process has to be manageable in terms of number of manual and automated executed steps. If possible, for all process steps and artefacts a template or tool support will be provided. The work related to the underlying research problems and the developed approach will be shown by means of a literature survey.

#### 1.2.2 Practice

In order to guarantee the practical usage of the research approach, the underlying research problems have to be based on a literature survey and observations of real-life software engineering projects. Furthermore, the developed approach has to be evaluated based on realistic project data. The data used for the evaluation should consist of documents and models representative for industry projects. The evaluation results should lead to the acceptance or rejection of defined phd hypothesis.

## 1.3 PUBLICATIONS

The results of this dissertation were reviewed and published in the following conferences and workshops:

- Michael Mlynarski, Baris Güldali, Melanie Späth, and Gregor Engels. From Design Models to Test Models by Means of Test Ideas. In *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10, New York, NY, USA, 2009. ACM
- Michael Mlynarski. Holistic Model-Based Testing for Business Information Systems. In *Proceedings of 3rd Interna-*

*tional Conference on Software Testing, Verification and Validation*, pages 327–330, Paris, France, April 2010. IEEE

- Baris Güldali, Michael Mlynarski, and Yavuz Sancar. Effort Comparison of Model-based Testing Scenarios. In *Proceedings of 1st International Workshop on Quality of Model-Based Testing (QuoMBaT 2010)*, pages 28–36, Paris, France, 2010

Further publications related to the concepts of this thesis were published in:

- Baris Güldali, Michael Mlynarski, Andreas Wübbeke, and Gregor Engels. Model-Based System Testing Using Visual Contracts. In *Proceedings of Euromicro SEAA Conference 2009, Special Session on Model Driven Engineering*, pages 121–124, Washington, DC, USA, 2009. IEEE Computer Society
- Baris Güldali, Stefan Jungmayr, Michael Mlynarski, Stefan Neumann, and Mario Winter. Starthilfe für modellbasiertes Testen. *OBJEKTSpektrum*, 3:63–69, 2010
- Dominik Beulen, Baris Güldali, and Michael Mlynarski. Tabellarischer Vergleich der Prozessmodelle für modellbasiertes Testen aus Managementsicht. *Softwaretechnik-Trends*, 30(2):6–9, 2010

Furthermore, the author supervised the following students thesis with direct relation to this phd thesis:

- Benjamin Niebuhr. Test case generation from UML models described with the UML Testing Profile. Bachelor thesis, University of Applied Sciences Brandenburg, Faculty for Applied Computer Science, December 2009
- Annette Heym. A model-based testing approach for business information systems. Master thesis, University of Augsburg, Faculty of applied computer science, Chair of Software Engineering and Programming Languages, February 2010
- Andreas Fichter. Messung und Bewertung der Modellabdeckung anhand der Traceability-Informationen eines Modelltransformationsprozesses. Master thesis, Hochschule Furtwangen University, Fakultät Wirtschaftsinformatik, Studiengang Application Architectures, September 2010



## 1.4 OUTLINE

This document begins in Chapter 2 with the introduction of foundations needed to understand the proposed research approach. Next, the underlying research problems are clearly defined based on the conducted literature survey and shown in Chapter 3. In Chapter 4, a meta-model algebra is presented, which is used for the method engineering process of this thesis. The holistic approach for model-based system testing is presented in Chapter 5. In Chapter 6, the evaluation of the research approach by conducting an experiment is presented. The reached contributions, summary of the research work and an outlook on further research is presented in Chapter 7.



---

## DEFINITIONS AND PRELIMINARIES

---

In this chapter we briefly introduce the basic definitions and preliminaries needed to understand the research approach of this phd thesis. Also, several terms used in the last chapter are explained and the relevant literature is referenced. First, we introduce the field of dynamic software testing. We then explain how business information systems can be modelled and provide a short introduction in the representative approach used in this thesis. Next, the basic concepts of model-based testing are described. Besides the modelling approach for analysis models, we also introduce the modelling language used for the test model. Finally, we briefly introduce the model-driven development context and especially the model transformation technique.

### CONTENTS

2.1	Dynamic Software Testing . . . . .	21
2.2	Model-Based Testing . . . . .	29
2.3	Test Modelling Language . . . . .	38
2.4	Modelling Business Information Systems . . . . .	45
2.5	Model Transformations . . . . .	65
2.6	Summary . . . . .	71

### 2.1 DYNAMIC SOFTWARE TESTING

Software testing is one of the oldest disciplines of software engineering. We distinguish between statical and dynamical testing [SL05]. The first one groups all quality assurance techniques, which are performed without executing the system. Against it dynamical testing relies on executing tests on the system under test (SUT). Both test kinds can be further categorized in func-

tional and non-functional testing (for example performance or usability testing) according to the type of requirements they intend to validate. In this thesis we will focus on dynamical testing of functional requirements.

While testing small systems can rely on the testers knowledge, testing large-scale business information systems requires a systematic test process. Such a process helps to coordinate the testing activities in large teams during the project lifecycle. This need is independent of the development process used in a project. For example the same need exists in RUP-like [Kru03] but also in SCRUM-like [SB01] projects. Certain test roles, artefacts and methods are always necessary.

In the last years several test process models were introduced in the literature. There exists an international IEEE standard 829 [IEE08], which shows how to specify and manage tests in a project. But there are also process models not invented by a standardization organization. The first one is called TMap Next [KVdABVo8] and was created by Sogeti<sup>1</sup>. It consists of a very detailed test process definition with several task descriptions. It incorporates several best practices used in projects at Sogeti. Another well-known test process model was introduced by the International Software Testing Qualifications Board (ISTQB) (for example in Spillner et al. [SL05] and Linz et al. in [LSS07]). It is one of the most acknowledged definitions of a test process, test roles and glossary in the industry. In this thesis, we will concentrate on the ISTQB test process model from [SL05] because of its customization possibilities and awareness in the test and research community.

The so called *fundamental test process* is shown in Figure 7. It is divided into six test phases. A **Test Phase** "is a distinct set of test activities collected into a manageable phase of a project, e.g. the execution activities of a test level" [IST, p.40]. Within the *test planning* phase the strategy and effort estimation for all test activities is done. The *test controlling* phase measures and influences the current state of all other activities. Within *test analysis and design* phase the basis for tests is analyzed and logical test cases are designed. The *test implementation* phase refines the logical into concrete, executable test cases. In this thesis, we use the synonym **test specification** for the test phases analysis, design and

---

<sup>1</sup> <http://www.sogeti.com>

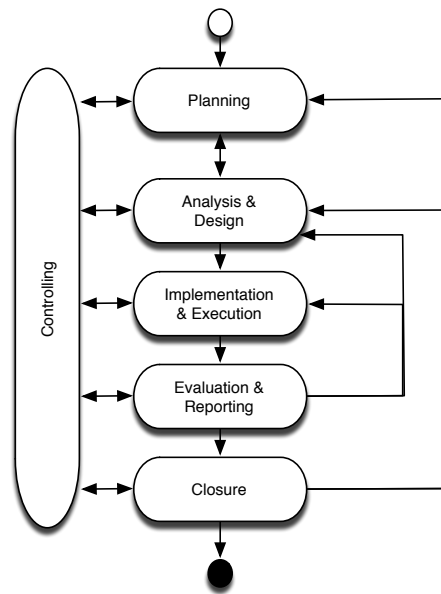


Figure 7: Fundamental test process after [SL05]

implementation. Within *test execution* phase each test case is executed and its result is analyzed in the *test evaluation* phase. After all mentioned activities are done, the *test closure* phase is conducted to collect and archive all information for improvements in further projects.

The test phases in Figure 7 have several dependencies. First, there exist the execution order (from test planning to test closure). Second, each test phase has a dependency on the *test controlling* phase. In this phase the work status of each phase is controlled and measured according to the goals definition from the test planning phase. The *test controlling* has the same purpose as the according activity in the field of project management. Finally, three loops in the process exist. Depending on the results of the *test execution* phase, the two prior test phases (*analysis & design* and *implementation & execution*) can be repeated. Also the observations from the *test closure* phase can lead to changes in the *test planning* and *analysis&design* phase.

### 2.1.1 Process and Artefacts

To assure a common understanding of the concepts presented in this thesis, we provide the basic definitions in the context of the fundamental testing process according to the ISTQB glossary [IST] here.

In order to speak about model-based system testing, we first have to define it. First, a **Test Level** "is a group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project" [IST, p.42]. The system test level in this context is defined as "the process of testing an integrated system to verify that it meets specified requirements" [IST, p.43]. For this thesis, we assume a model-driven development process, where the requirement verification and validation in system testing is based on the analysis model. In particular, model-based testing supports both validation and verification of requirements by using explicit models as stated in [PP05].

The analysis model is used as the basis for test design. The output of this test phase is a test case. The general definition of a test case is given by the ISTQB glossary as follows:

**Definition 1 Test Case** is "a set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement". [IST, p.40]

Test cases are divided into two types, namely:

- **Logical Test Case (LTC)** defined as "A test case without concrete (implementation level) values for input data and expected results. Logical operators are used; instances of the actual values are not yet defined and/or available" [IST, p.27]
- **Concrete Test Case (CTC)** defined as "A test case with concrete (implementation level) values for input data and expected results. Logical operators from high level test cases are replaced by actual values that correspond to the objectives of the logical operators" [IST, p.15]

The IEEE 829 Standard for Test Documentation extends the categorization of test cases according to the test level (for example unit level or system level test cases). According to the IEEE 829

[IEEo8] the following attributes are necessary to specify a Level Test Case:

1. Test case identifier - unique identifier
2. Objective - describes the focus, purpose and possible risks of a test case
3. Inputs - specifies each input needed to execute the test case
4. Outcome(s) - detailed specification of all outputs and expected behaviour
5. Environmental needs - describes the test environment (hardware and software) needed to execute the test case
6. Special procedural requirements - specifies constraints needed for test execution
7. Intercase dependencies - list of all other test cases which have to be executed prior this one

Since the IEEE 829 definition of a test case already requires the specification of several concrete (implementation level) details, the distinction between logical and concrete test cases is difficult in practice. Within the research project conducted as the part of this phd thesis, we have developed a test case definition and template based on the observation of several real-life testing projects.

For the purpose of this thesis we use the following structure of a test case:

1. Test case identifier - unique identifier
2. Title - meaningful title describing its purpose
3. Description - short description of the test case
4. Priority - test priority for test execution
5. Trace links - backward traceability to the requirements specification
6. Precondition - condition needed to execute the test case
7. Postcondition - condition after the test execution
8. Test data sets - set of logical or concrete test data
9. Steps
  - a) Identifier - unique test step identified

- b) Description - short description of the task, which is performed by the tester
- c) Expected results - precise definition of the expected result

This definition is part of the Capgemini CSD Test Methodology, which is one of the results of the research project conducted within this phd thesis. This structure can be used for the specification of LTC and CTC. The attribute *test data sets* contains only the names of equivalence classes to be used in the case of LTC. After applying the boundary-value analysis [SL05] several CTCs out of one LTC are created. In this case the equivalence classes are replaced with concrete test data values.

Different as in the IEEE 829 definition, we extend it with single test steps and clear pre/postconditions. Also, the additional prioritization is an important requirement for the execution of test cases in large-scale projects.

Test cases can be grouped to test suites. A **Test Suite** "is a set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one" [IST, p.45].

All test cases (LTC and CTC) specified in the test design and implementation phases are part of a test case specification document. A **Test Case Specification** "is a document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item" [IST, p.40].

Finally, a common term used in the context of test cases is a test oracle. A **Test Oracle** "is a source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but should not be the code" [IST, p.43].

### 2.1.2 Test roles

In the fundamental test process several roles are responsible for the different test phases introduced in the last subsection. For



the purpose of this thesis we define the following test roles according to the ISTQB glossary [IST]:

**Definition 2 Tester** A skilled professional who is involved in the testing of a component or system.

Since the tester can be involved in several phases of the test process, we refine this definition especially for the test design phase as follows:

**Definition 3 Test designer** A skilled professional who is responsible for the design and maintenance of the test case specification.

**Definition 4 Test manager** The person responsible for project management of testing activities and resources, and evaluation of a test object. The individual who directs, controls, administers, plans and regulates the evaluation of a test object.

### 2.1.3 Meta-Model

In Figure 8, we have summarized the relations between the artefact definitions and the different test roles provided so far. The central artefact is a *Test Case* which consists of several *Test Steps*. Test cases can be of type *Logical Test Case* and its refinement *Concrete Test Case*. Test cases aim to stimulate or execute the system under test with *Test Data Sets* and observe the behaviour. Test cases have to fulfill certain *Conditions* in order to be executed. Those conditions are described by test data, which reflects the state before and after the execution of a system. The verdict (pass or fail) is decided by a *Test Oracle*, which can have different sources (for example human or the SUT). The test oracle is not a test artefact.

Several test cases can be grouped to *Test Suites* and several test suites build up a *Test Case Specification*. A *Test Designer* is responsible for the design of single test cases. The *Test Manager* reponses the planning of the test design phase which results in a test case specification.

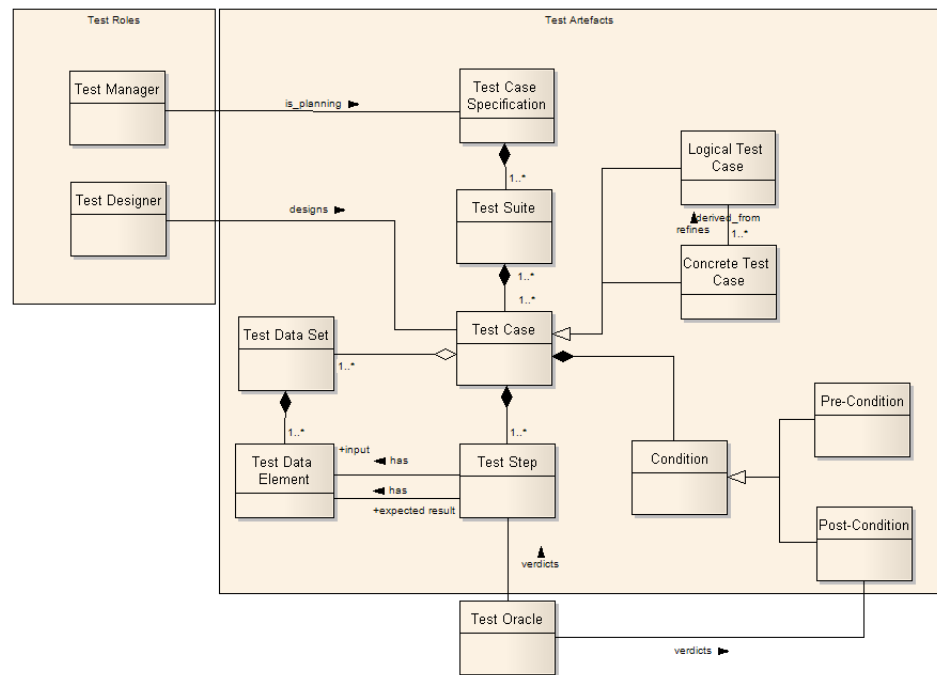


Figure 8: Meta-model for functional software testing

#### 2.1.4 Risk-Based Testing

Within a testing project not all requirements and system functionality can be tested. Complete testing is not possible because of time and effort limitations. Since testing always covers only an excerpt of the system functionality, the selection of its most important parts for testing is crucial.

In practice each test method has to incorporate risk management for the reasons described above. Risk-based testing is defined as "an approach to testing to reduce the level of product risks and inform stakeholders of their status, starting in the initial stages of a project. It involves the identification of product risks and the use of risk levels to guide the test process" [IST, p.35].

There exist several techniques for implementing risk-based testing in a project. Since system testing has to validate the specified requirements within the analysis model, those requirements have to incorporate the definition of risks and priorities. Guided by the risk level of the requirements manual or automated test design can be conducted. In both cases a test design method

with coverage-oriented or fault-oriented goals has to be defined. For requirements with high risk level strong coverage criteria as all possible use case scenarios, all pairwise input data combinations, etc. are defined.

In this thesis the topic of risk-based testing is very important. We introduce a new approach for automated generation of test cases. Since this automation can result in infinite number of test cases, the execution effort can be not practicable. To solve this problem, a risk-based approach has to guide the test case generation.

## 2.2 MODEL-BASED TESTING

In this section we briefly introduce our understanding of model-based testing with its basic concepts and artefacts used in this thesis.

### 2.2.1 Definition

For a common understanding of model-based testing, we provide the following main definition of this advanced testing technique:

**Definition 5 Model-Based Testing (MBT)** is the automation of black-box test design [UL07, p.17].

To refine this definition we define the goals of MBT according to Heckel and Lohmann [HL03]:

- Generation of test cases from models according to a test selection criteria
- Generation of a test oracle to determine the expected results of a test
- The execution of tests in test environments, possibly also generated from models

Despite the wide range of advantages propagated by several researchers as deeper understanding of requirements through formal models, the main characteristic of MBT is still the automatic generation of test artefacts. The main test artefact are test cases normally designed by testers. The manual usage of test design

methods like equivalence class analysis [SL05] or process cycle testing [KVdABVo8] is replaced in MBT by the usage of test selection criteria. We provide an introduction of such criteria later in this section.

Besides the automatic generation of test cases also the test oracles used for test evaluation purposes can be generated. This generation step can be only done if the expected results can be clearly modelled within the model used for generation purposes. Finally, the test environment in which test cases are executed can be generated from models. The assumption is that architectural (structure and behaviour) aspects of the test environment can be modelled. In Güldali et al. [GJM<sup>+</sup>10], we have introduced a method to support the decision process for finding the suitable test artefacts to be generated in MBT.

In this thesis we focus on the generation of test artefacts (especially test models and test cases) from analysis models. This is one of the different scenarios for model-based testing known from the literature. We will briefly introduce two of them in the next subsection.

### 2.2.2 Methodological Issues

Since there exist hundreds of MBT approaches in the literature, several categorization approaches have been undertaken. Some authors tried to define a clear taxonomy for MBT. Utting et al. in [UPL06] categorizes MBT approaches according to three general aspects: models used, the test generation and test execution technology. Such a taxonomy allows us to group approaches with respect to the mentioned aspects. There exist however methodological issues related to the source of the models used for generation, which are not sufficiently covered with such a taxonomy.

Before we discuss the methodological issues and its relation to this thesis, we first provide a clear definition of the different

models used in model-based testing. We distinguish between system and test models. Those models can be defined as follows:

**Definition 6 System Model** describes the system to be developed. Especially its structure and behaviour. [RBGW10, p.387] In this thesis we use the analysis model as a system model since we focus on system testing.

**Definition 7 Test Model** defines the behaviour of the SUT from an external point of view and explicitly state what events the SUT should accept at a certain moment. [MJV<sup>+</sup>10, p.292]

Depending on the source for the test model several scenarios for model-based testing can be defined. Two general scenarios where the model used for test generation is created separately or is shared with the developers were introduced by the mentioned taxonomy in [UPLo6] and in the book from Utting and Legeard in [ULo7]. Pretschner and Philipps introduce in [PP05] the following four scenarios where: one model is used for both code and test generation (*common model*), the model used for test generation is automatically extracted from the source code (*automatic model extraction*), the mentioned model is created manually (*manual modeling*) and finally two *separate models* for code and test generation exists. Within a literature survey performed in this thesis, further two scenarios where models used for test generation are created by model transformations or reengineered from existing test cases were identified. This scenarios were introduced in Gldali et al. [GMS10].

The distinction between several scenarios in MBT is important, because the level of test independency differs within scenarios. As described in the introduction of this thesis, we focus on the *separate models* scenario from Pretschner and Philipps [PP05, p.8] with the additional technique of model transformations to generate test models from system models. In [GMS10] we name this scenario *model from model*. Besides the level of test independency and its influence on test quality, the distinction between several scenarios is needed to select the project- or company-specific MBT approach and estimate the needed effort. Both aspects (test independency and effort) are strictly related to the research problems and contribution of this thesis. Therefore this categorization is important for the understanding of the thesis.

*Model from model  
scenario*

### 2.2.3 Process and Artefacts

The methodological issues of MBT introduced before result in several scenarios depending on the model usage in MBT. Since in each of those scenarios different activities as model reengineering versus manual model creation can be identified, the testing process differs in each case. In the literature several high-level process models for model-based testing are propagated. El-Far and Whittaker introduce in [EFW01] five general MBT activities (build model, generate tests, run scripts, get actual and expected results and analyze data), which are executed sequentially. Utting and Legeard in [UL07, p.26-30] refine the test generation activity in generating abstract tests and its concretization for test execution. Further refinements of the mentioned approaches can be found in [AD97, DNT09]. All referenced publications try to define a new process for model-based testing.

A different view on MBT and process definition was introduced by Spijkerman and Eckardt [SE09]. The authors analyzed the publications mentioned before and identified new or changed activities in the fundamental testing process (FTP) from [SL05]. This view aims to integrate MBT into the standard testing process rather than define a new one.

Based on the ideas from Spijkerman and Eckardt [SE09], we identified several MBT activities which extend the FTP introduced in Section 2.1. The result is shown in Table 2.

**Table 2:** New MBT activities in the FTP

<b>FTP</b>	<b>MBT activity</b>
Planning	Choose test selection criteria
Controlling	Control test model quality
Analysis & Design	Create test model, Generate logical test cases
Implementation & Execution	Create test data sets, Generate concrete test cases
Evaluation & Reporting	none
Closure	none

In order to generate logical test cases, one or more test selection criteria have to be selected. This activity has to be done in the *planning* phase. Additionally to controlling test cases, test results, etc. the controlling of test model quality has to be done in the *controlling* phase. The main addition to the FTP is the creation of test models and automated generation of logical test cases. In the context of this thesis, we aim to derive test models automatically from analysis models rather than create them manually. Both activities are part of the *analysis & design* phase. In order to *implement & execute* tests concrete test cases have to be generated. For this, test data sets have to be created manually or automatically. In the last two phases of the FTP no new activities are needed. The only difference in MBT context is that new artefacts as test models have to be taken into account.

MBT activities

Based on the introduction provided so far, we define a artefact meta-model for MBT. In Figure 9 the derivation of the *test model* from the *analysis model* depicts the *model from model* scenario from [GMS10]. We also define a *test model* to be composed of several *logical test cases*. This kind of test model definition is also known as the test specification model [RBGW10]. The refinement from logical to *concrete test cases* with *test data sets* was already part of the test meta-model from Figure 8. Additionally the concrete test cases are automatically derived from the logical test cases.

Test specification model

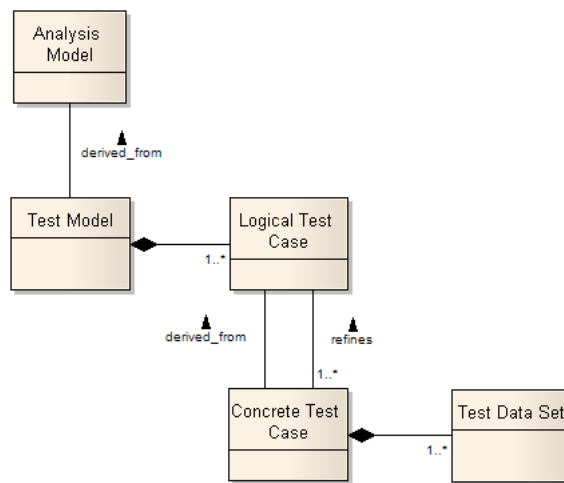


Figure 9: Artefact meta-model for the MBT process

The derivation of the test model and therefore the logical test cases from the analysis model is done by using test selection

algorithms. A brief introduction will be provided in the next subsection.

#### 2.2.4 Test selection algorithms

Automated test case generation in MBT is possible through algorithms, which select test cases from behavioural models of the SUT. For better understanding of the test selection used in this thesis, we provide some basic definitions.

**Definition 8** **Test selection criteria** are the means of communicating the choice of tests to a model-based testing tool. Examples of test selection criteria are: requirements coverage, code coverage, fault type coverage or model coverage [UL07, p.107]. In this thesis we use structural and data coverage criteria as test selection criteria.

**Definition 9** **Structural coverage criteria** describe the parts of a model, which have to be covered by tests.

**Definition 10** **Data coverage criteria** deal with the coverage of the input data space of an operation or transition in the model [UL07, p.109].

**Definition 11** **Test selection algorithm** is the implementation, which tries to satisfy a given test selection criteria for a given test model. One test selection criteria can be satisfied by several test selection algorithms depending on the used heuristic.

In the introduction of this thesis, we have described the problem of measuring the coverage of the analysis model (see Section 1.1). The selection of logical test cases from the analysis model mainly influence the reached model coverage. By definition the usage of structural coverage criteria guarantees the coverage of the desired model parts. As stated by several publications like [DNSVTo7, Bin99, GECMT05], test case selection in MBT is based mostly on behavioural models of the SUT. This way, the behavioural modelling viewpoint of the analysis model is always covered. The research problem of this thesis deals with the coverage of all three modelling viewpoints by using the holistic view in MBT.



In order to use the holistic view, we introduce a new kind of model coverage called holistic model coverage. It is defined as follows:

**Definition 12 Holistic Model Coverage** is the combined coverage of all modelling viewpoints, which depends on the used test selection criteria and the usage of model relations within the test generation process.

This definition is based on the test selection criteria described in this section. Further, it is based on model relations as the integration of different modelling viewpoints, which was already introduced in Subsection 2.4.11.

The structural coverage criteria in MBT can be further decomposed into:

- Control-flow oriented coverage criteria
- Data-flow oriented coverage criteria
- Transition-based coverage criteria
- UML-based coverage criteria

The *control-flow oriented coverage criteria* are based on the coverage of statements, decisions or branches of code. Since we aim to cover parts of the analysis model, this type of coverage criteria is not considered. The *data-flow oriented coverage criteria* aim to cover the data used in a control-flow. The topic of (test) data is important for the generation of concrete test cases. Rather than the structural coverage criteria, we use the data coverage criteria here. The *transition-based coverage criteria* can be applied on all models using a transition system. This is the case in behavioural models like the UML activity diagrams. This criteria type is important for the generation of logical test cases in our approach. Finally, there exist *UML-based coverage criteria* which focus especially on UML diagrams and their structure. The last type of structural coverage criteria can be seen as a refinement of the other mentioned types for UML.

*Test selection  
criteria in this thesis*

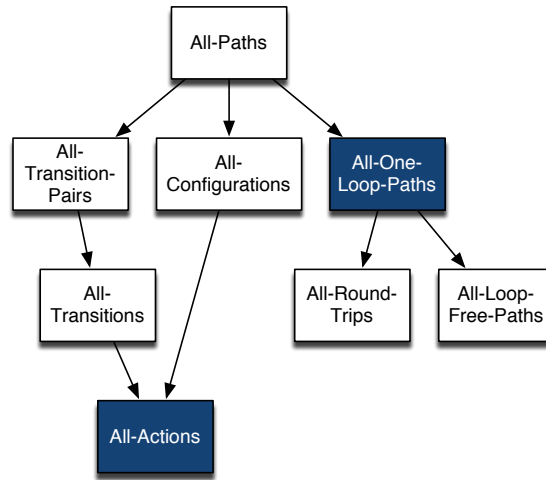
As mentioned above, we use transition-based and data coverage criteria for the purpose of this thesis. In the next paragraphs we briefly introduce the concrete criteria.

### Transition-based test selection

The goal of the transition-based test selection is to use the transition system of behavioural models to select paths. In this thesis, we use UML activity diagrams as the behavioural model from which test cases represented as paths are selected. In this context, we provide the following definition of a path.

**Definition 13 Path** is a node sequence from the initial to the final node. It does not consist of decision, merge or join/fork nodes. Two paths are equal if their node sequences are equal. A path is equal to a logical test case.

To select paths from transition-based behavioural models several coverage criteria exist. In Figure 10 we use the categorization introduced in Utting and Legeard [UL07]. In general coverage criteria have a hierarchy, which is depicted in Figure 10 by using a tree structure with directed edges. An edge  $A \rightarrow B$  means, that A is stronger than (subsumes) criterion B. This subsumption depends on how hard it is to satisfy a given criterion with respect to the number of paths.



**Figure 10:** Coverage criteria subsumption derived from Utting and Legeard [UL07]

The strongest transition-based coverage criterion is *All-Paths*, where each path has to be traversed at least once. Due to the existence of infinite loops, this criterion is not practical. The restriction to visit paths only once is provided by the *All-One-Loop-Paths* (also

called *AllPathOneLoop*) criterion. Two further criteria *All-Round-Trips* (traverse a loop only once without traversing the preade of follow loop) and *All-Loop-Free-Paths* (traverse only loop free paths) refine this definition.

The *All-Paths* criterion subsumes also another category of criteria. First, the *All-Transition-Pairs* criteria exists, where each pair of adjacent transitions has to be traversed only once. This criterion subsumes the *All-Transitions* criterion, which requires all transitions to be covered. The weakest criterion is the *All-Actions* criterion, where all action nodes of an UML activity diagram have to be covered by the traversed paths. Finally, *All-Paths* subsumes the *All-Configurations* criterion, which is specific for finite state machines and requires all configurations (with parallelism) to be visited at least once.

For the purpose of this thesis, we select two transition-based coverage criteria highlighted in Figure 10. The goal is to use a weaker criterion like *All-Actions* and a stronger one like *All-One-Loop-Paths*. This way, we want to evaluate the impact on the holistic model coverage while selecting logical test cases from analysis models based on different coverage criteria.

*Chosen transition  
criteria*

### *Test data selection*

The transition-based coverage criteria are used to select logical test cases in this thesis. To select concrete test cases, the selection of test data rather than paths is needed. Test data is defined as the values of equivalence classes derived from the input domain (in our case the analysis model). For this, we use the *data coverage criteria* defined at the beginning of this subsection. As within the structural coverage criteria, also several types of data coverage criteria exist. Utting and Legeard [UL07] differentiate between three criteria categories like *boundary value testing* (select test data from the boundaries of equivalence classes), *statistical data coverage* (select random test data from the equivalence classes) and *pairwise testing* (select test data by combining equivalence classes of the input domain). For this thesis the last type of data coverage criteria is interesting.

There exist three criteria for the pairwise testing category:

- *Pairwise coverage* (combination of all pairs of equivalence classes )

- *N-wise coverage* (combination of all N equivalence classes)
- *All-combinations coverage* (each possible combination of all equivalence classes)

#### Chosen data criteria

In practice, the *All-combinations* coverage criterion is not used, since it leads to the combinatorial explosion. For the purpose of this thesis, we use weaker coverage criteria like *pairwise* or *N-wise* coverage. Further, simplified data coverage criteria like a sequential combination of parameters is possible. For example values for equivalence classes A,B,C,D are combined to {A<sub>1</sub>, B<sub>1</sub>, C<sub>1</sub>, D<sub>1</sub>}, {A<sub>2</sub>, B<sub>2</sub>, C<sub>2</sub>, D<sub>2</sub>}, etc.

In this section we have introduced the basic definitions of MBT together with its process and artefacts. We have briefly described the test selection criteria used in this thesis. Since we focus on the *model from model* MBT scenario, we will now introduce the modelling language used for describing test models.

## 2.3 TEST MODELLING LANGUAGE

In the last two sections, we have introduced several criteria for selecting logical test cases, which are part of a test model. In the context of this thesis, we use a MBT scenario in which a test model is derived from an analysis model. To enable the automated derivation of test models, we need to select a test modelling language.

#### Different test modelling scenarios

To select a test modelling language, we first have to consider the structure of the analysis model. Especially the structure of the behavioural models used for test selection influence the structure of the test model. In Figure 11 we depicted two possible scenarios for the test model structure. In *complete models* the behavioural model (in the exemplary analysis model used in this thesis the UML activity diagram) is transferred completely into a new test model, where additional test information (depicted with notes on each node) is appended. Logical test cases are selected from the test model. Another possibility is the *selected*

*parts* scenario, where logical test cases are directly selected from the behavioural model.

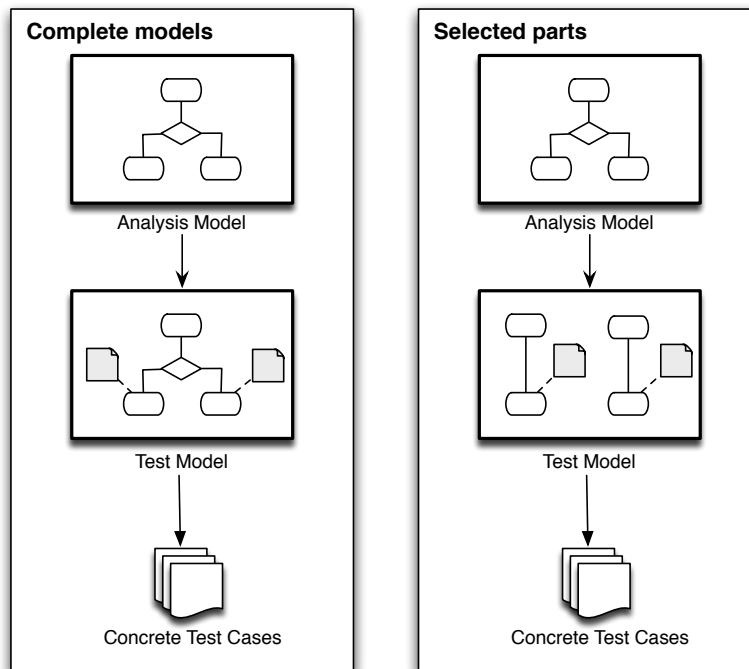


Figure 11: Different scenarios for the test model structure

Since both scenarios can be potentially used to solve the problems of this thesis, the most suitable has to be selected. In *complete models* the test designer appends test related information (like equivalence classes or expected results) to a single behaviour model. The main advantage of this test model is that different test selection criteria (transition-based and data coverage criteria) can be applied to generate logical test cases. There is also a common modelling language used in the analysis and test model, which supports the work of the test designer. Unfortunately, the transfer of behaviour models into the test model without prior selecting logical test cases does not uncover faults in the analysis model, nor does it support the graphical modelling of single test cases. Also, if changes in the different modelling viewpoints of the analysis model occur, they have to be implemented manually in the test model.

While the main disadvantage of the first scenario is the missing fault-detection in the analysis model, the *selected parts* scenario

*Complete models  
scenario*

*Selected parts  
scenario*

solves this problem by providing a means of modelling logical test cases. This way the test selection has to be performed first. Since we assume that the analysis model does not contain any information about test data (for example equivalence classes), only the transition-based coverage criteria can be applied. Through the graphical modelling of logical test cases, further ones can be created in addition to the generated ones. Since logical test cases are linear flows through the behaviour model, the branches of behaviour models are not visible anymore. This problem can be solved by appending notes to each test case step describing the selected guards of the mentioned branches. The main disadvantage of this scenario is the effort resulting from changes within the analysis model.

In this thesis, we use the *selected parts* scenario because of the advantage of fault-detection in the analysis model. The problem of implementing changes from the analysis model in the test model exists in both mentioned scenarios.

### 2.3.1 UML Testing Profile

In order to select a test modelling language, which can be used in the *selected parts* scenario, we define the following requirements:

- REQ1 **Modelling viewpoint for the test behaviour** - the language should support the modelling of logical test cases
- REQ2 **Modelling viewpoint for the test data** - the language should support the modelling of equivalence classes for test data design
- REQ3 **Relation between the behaviour and data viewpoints** - the language should support the relation between logical test cases and equivalence classes
- REQ4 **Standardized modelling language** - the language should be widely-known in terms of a standardization

Within the literature survey performed as part of this phd thesis, we have identified the UML Testing Profile (UTP) [Obj07b] as the most suitable for all mentioned requirements. The UTP was introduced by the Object Management Group as the standardized test modelling language. UTP uses the profiling mechanism of the UML [Obj09, p.653] to cover the domain of software testing.

The language provides four concepts for modelling different aspects of tests performed on the SUT:

- **Test Behaviour** - model the behaviour in terms of test cases
- **Test Architecture** - model the structure of the test environment and configuration of tests
- **Test Data** - model the data used in test cases
- **Timers** - model the time constraints to control the test behaviour

The mentioned concepts are the synonym for modelling viewpoints. Each viewpoint groups the aspects needed to specify tests. Since the viewpoints aim to specify test cases and the environment in which they are executed the model created with UTP is also called a *test specification model* [RBGW10].

The UTP was developed for all kind of systems. A broad range of artefacts have to be created for each viewpoint exists. For the purpose of this thesis, we have selected a subset of them and extended some definitions (for example of the test case). Since we focus on functional testing, the *Timers* viewpoint is not considered at all. This way load or performance testing is not covered in our approach.

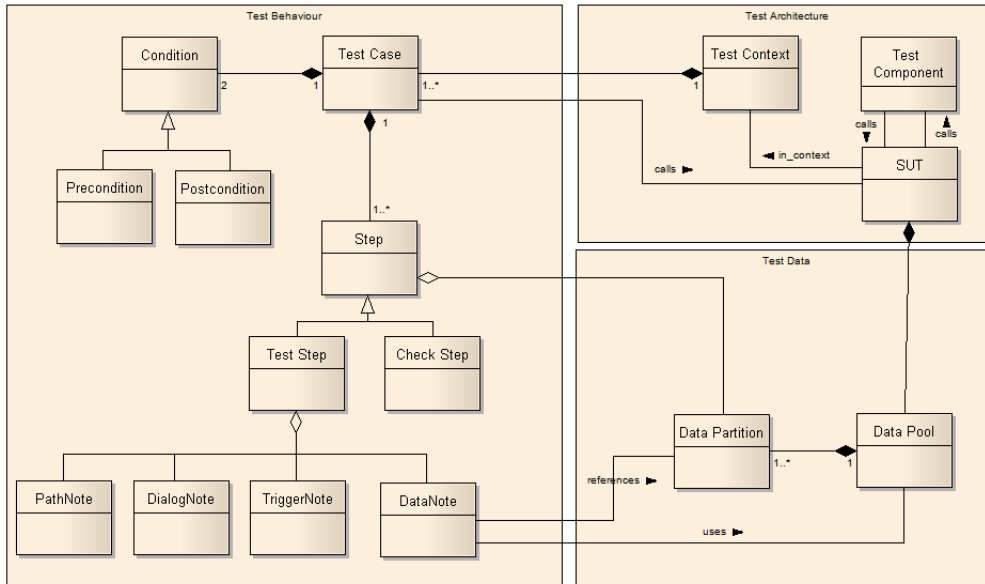


Figure 12: Artefact meta-model for the UML Testing Profile

### 2.3.2 Artefact Meta-Model

In the following we introduce each artefact used in our customized version of UTP. Within Figure 12, we have depicted the underlying artefact meta-model. The meta-model is grouped by the three modelling viewpoints mentioned above.

#### *Test Behaviour*

The main artefact of our test model are *Test Cases*. Those test cases do not contain concrete test data and are called logical test cases. A test case has one *Precondition* and *Postcondition*. Each test case consists of several *Steps*. We distinguish between *Test Steps* (step performed by the tester during execution) and *Check Steps* (step executed by the SUT). Each test step contains of several notes for the information, which will be derived automatically using the holistic view. The *DialogNote* references the dialog used in the test step. *TriggerNote* references the trigger to be invoked to stimulate the SUT (for example an action performed on the dialog). The input data derived from the analysis model is placed in the *DataNote*. Finally, the *PathNote* references the branch and guard taken during the test selection. Since the UTP does not prescribe how to precisely model test steps, we added the concept of several notes to integrate the information collected from the interaction (*DialogNote* and *TriggerNote*) and structure (*DataNote*) modelling viewpoint of the analysis model.

In Figure 13 an example of a logical test case is shown. The test case consists of three steps, which test the creation of a new attendee in a ski course system (see thesis running example in Subsection 2.4.4). The first step called *Enter\_Attendee\_Data* is executed by the tester (*TestStep* stereotype). The following steps (*CheckAction* stereotype) are only triggered by the tester and executed by the SUT. As mentioned earlier, several notes define the pre- and postconditions, the needed input data (like *FirstName*, *LastName*, *Age*, etc.). The first test step is triggered by clicking a button called *SaveAttendee* on the *BookAttendeeOnCourse* dialog. To remain the traceability to the analysis model from which the test case was derived, a path note describes the branch taken in the behavioural model of the analysis model.

The structure of a *Test Case* in our customized version of UTP covers the definition and template of a logical test case from Sub-



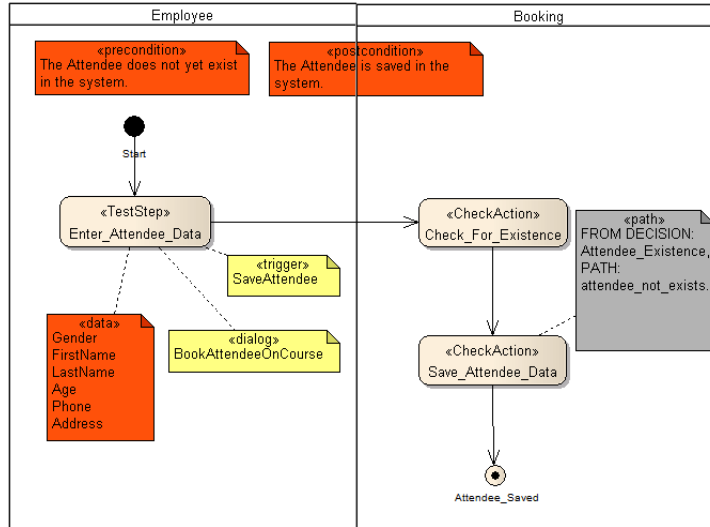


Figure 13: Example for a logical test case

section 2.1.1. Only the concrete test data sets are not part of the test model. Based on the observations of several large-scale industry projects during this phd thesis, we identified the need to manage concrete test data in an external source like database system. The management of large amount of such test data within a model results in high effort, which should be optimized.

### Test Architecture

The main artefact of the test architecture viewpoint is the *Test Context*. As the name states, it defines the context to be used during testing. In UTP the test context groups the test cases from the behavioural viewpoint. During the derivation of the test model, the use cases from the analysis model are used as test context. Further, testing is performed in a test environment on the *SUT* and especially in system testing with several *Test Components*. The eponymous artefacts in Figure 12 can call each other during test execution. The SUT is always called from a test case.

In Figure 14, we provide an example for the test architecture viewpoint. There exist one test context, which was derived from the use case *Book\_Attendee\_on\_Course\_Course* (see running example in the next section). The test context groups four logical test cases. Additionally, three test components (*Employee*, *Course* and *Customer*) and one SUT (*Booking*) exist. The SUT component com-

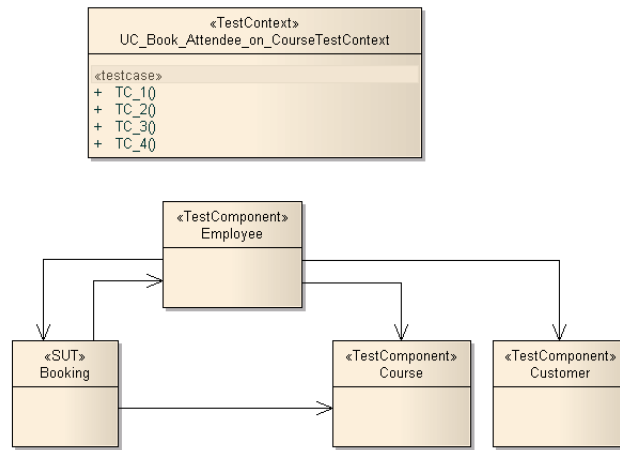


Figure 14: Example of a test architecture viewpoint in the test model

municates with the the *Course* and *Employee* component. The *Customer* test component is only invoked by the *Employee* component.

### Test Data

The design of test data in our test model is done with *Data Pools*, which contain of several *Data Partitions*. The application of equivalence class analysis (see [SL05]) results in several data partitions. Since the analysis model does not contain of partitions, only suggestions like the logical data types can be automatically derived. Based on those suggestions, the test designer can manually perform equivalence and boundary value analysis. A data pool with several data partitions is always created for one test context.

In Figure 15 an example of the test data viewpoint is shown. Two data pools *BookingData* and *CourseData* exist. Each data pool contains of several data types, which are used as input data in the logical test cases. For each data pool a valid and invalid data partition (known as equivalence classes from [SL05]) is defined. Each data partition is refined with concrete data sets (for example Age=50, FirstName=Michael, etc.) by applying the boundary-value analysis method.

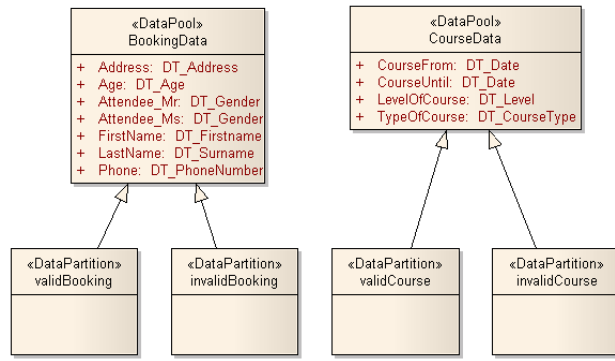


Figure 15: Example of a test data viewpoint in the test model

## 2.4 MODELLING BUSINESS INFORMATION SYSTEMS

In this section, we will briefly introduce the topic of modelling business information systems. We focus on the modelling task done performed by business analysts, which create the analysis model. To explain the usage of the holistic view in model-based system testing, we use an exemplary modelling approach used in the industry research project conducted within this thesis.

We first provide some general definitions in the context of modelling. Then, we briefly introduce the exemplary modelling approach and the running example used in this document.

### 2.4.1 General definitions

To gain a common understanding of the modelling domain used in this thesis, we created a simple ontology, which is depicted in Figure 16. In general, we distinguish between the *Language* part and the *Modelling Notation* part. This abstraction is needed to point out that the holistic view can be applied on different modelling languages. In this thesis we use the UML as an exemplary modelling language.

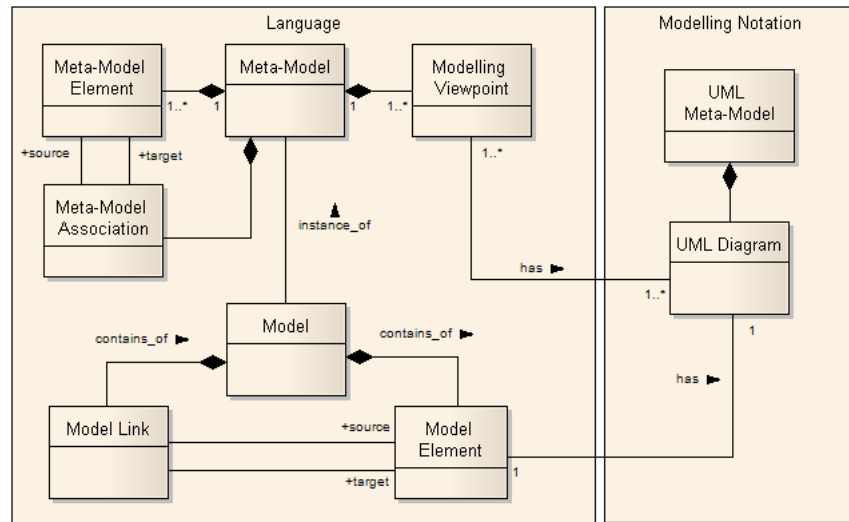


Figure 16: Ontology of the modelling domain used in this thesis

Two basic elements of the ontology are the *model* and *meta-model*. In this thesis we use the following definitions:

**Definition 14 Model** - partial reproduction of the real-world created by using abstraction for a certain purpose. Example are analysis models, which reproduce the requirements for a software to be build.

**Definition 15 Meta-Model** - specification of the language used to create a model, which itself is also model. Example is the meta-model of the analysis model.

The central element of the *Language* part of our ontology is a *meta-model*. The *meta-model* consists of several *meta-model elements*, which are related by *meta-model associations*. Further, each meta-model has one or more *modelling viewpoints*. Examples are the structure, behaviour and interaction modelling viewpoint mentioned in the introduction of this thesis. Since the exemplary modelling approach introduced in this section uses the UML, each modelling viewpoint is described by several *UML diagrams*. The diagrams are part of the *UML meta-model* [Obj09]. This way the meta-model together with its modelling viewpoints is described with the UML. This particular modelling language can be replaced by any other modelling language under the assumption that several representation forms (here diagrams) can be used to create models with different viewpoints.

*Modelling notation  
is changeable*

According to the Meta-Object Facility (MOF) framework [Obj06a], models are created by instantiating meta-models. Each model consists of *model elements* and *model links*. Both elements are created by the instantiation of the *meta-model elements* and *meta-model associations*.

#### 2.4.2 Motivation

Each system is implemented according to some requirements. The first step is to clearly specify the user requirements from the problem view. This is the main task of the requirements engineering discipline [PR10]. The software requirements from the solution view (see IEEE 830 [IEE98]) are specified within the business analysis discipline (see RUP discipline "Analysis & Design" in [Kru03]). This way, a clear distinction between the problem-oriented and solution-oriented requirements specification is given. It results in a deeper understanding of the customer needs and prevents the underspecified requirements at the problem level.

*Software  
requirements  
specification*

The requirements at the different levels can be specified with textual descriptions. This often leads to misunderstood requirements specifications, because of the subjective interpretation of such descriptions. To solve this problem, the model-driven development introduces models to specify the requirements [GPR06]. Different notations like the UML can be used to create such models. In the model-driven architecture framework [Obj03] each software engineering discipline results in a new model type (like the requirements model, analysis model and design models)

*Modelling  
requirements*

During the specification and testing of software systems it is important to distinguish the system type. The type influences the system aspects (like usability, timing, parallelism, environment, etc.) which have to be accordingly specified and tested. In general there are two types of systems, namely information and embedded systems. The information systems support the business workflow, the user communicates with the system through a graphical user interface and a persistence layer to the underlying data exists [NRP05]. The embedded systems often do not have a graphical user interface, communicate through data interfaces and are integrated into a hardware component. In this thesis, we focus on the specification and testing of business in-

*Information systems  
vs. embedded  
systems*

formation systems and choose an according real-life modelling approach in the next subsection.

#### 2.4.3 Representative industry modelling approach

To perform system testing we have to derive test cases from an analysis model. In order to define a model-based testing approach, we have to use a certain structure of the analysis model. In the current literature there exists no clear standard for modelling business information systems which is used by several organizations (see discussion in [BL02]). The general concept of the object-oriented analysis and design (OOA/OOD) introduced in [BME<sup>+</sup>07], shows how to model different aspects of the system with the UML. The notion of viewpoints can be found in approaches as Finkelstein et al. [FKN<sup>+</sup>92] or Pohl and Rupp [PR10]. Several approaches in the OOA field as the one from Ivar Jacobson [Jac92] or Alistair Cockburn [Coc01], concentrate on the use-case based modelling. Domain-specific modelling with approaches as Eric Evans [Eva03], focus on modelling the domain with language definitions. Finally, more formal approaches like Hesse and Tilley [HT05] show how to derive objects within the use case or OOD modelling. That is why, we have to select a representative one.

Based on the problem definition from Section 1.1, we define the following requirements for a representative modelling approach for business analysis:

- REQ1 Provide a meta-model definition
- REQ2 Distinction between the structure, behaviour and interaction modelling viewpoints
- REQ3 Support for an integrated interaction modelling
- REQ4 Support for model relationships between the modelling viewpoints

*Exemplary industry modelling approach*

For this thesis we choose the modelling approach introduced in [SSE09]. It fulfills all of the mentioned requirements. The approach is used by Capgemini Technology Solutions (TS). The company develops complex business information systems for customers in different domains as automotive, finance, government, telecommunication and logistics. Since each project at Capgemini TS needs to specify the problem- and solution-oriented

requirements, the company developed his own use-case based specification method. The company-wide approach is based on almost 30 years software engineering experience and introduces a task-oriented approach for specifying large-scale business information systems. In the following parts of this thesis we will use the term *specification method* as a synonym for the Capgemini TS modelling approach.

During the specification process several tasks have to be performed. Each task results in a separate artefact of the analysis model. For example the task *Specify Use Cases* results in the use case specification. The method defines for each artefact its content, form, process and tool support. We will use this categorization to describe the artefacts relevant for this thesis.

The specification method defines several tasks. Not all tasks and therefore artefacts are needed in our approach. For the purpose of this thesis we focus on the following tasks:

- Specify Component Model
- Specify Use Cases
- Specify Dialogs
- Specify Logical Data Model
- Specify Logical Data Types

This reduction was chosen because of the focus on the system testing level. At this level only the workflows, here use cases for the whole system are tested. The dialogs are used as the user interface in the mentioned use cases. Because the logical data model and data types can be used to derive test data, they are considered too. Last, the component model is important to identify the conceptual components involved during testing.

After the short motivation of the specification method, we will now introduce the running example, which is used within this thesis. This example will help us to describe each artefact resulting from the tasks mentioned before.

#### 2.4.4 Running example "Gabi's Ski School"

The running example used in this thesis is a fictive project called "Gabi's Skis School". It was developed at Capgemini, CSD Re-

search for training purposes and to support the method engineering in the internal research department by proving examples of software requirements. The fictive project "Gabi's Ski School" reflects a typical SRS of a business information system in a software engineering project.

The goal of the project is to develop a ski course management system for a small ski school called "Gabi's Ski School". First, user requirements were elicited during interviews with the fictive customer. The following problem was identified: Currently all tasks within the ski school are done manually and are very error-prone and time intensive. To improve the business process in the ski school a business information system, which supports tasks like: booking, course and customer management should be developed. The specification of a conceptual solution is done according to modelling approach introduced in this section.

**Table 3:** Elements of the analysis model for the "Gabi's Ski School"

Modelling Viewpoint	Model Type	Name
Structure	Conceptual Component	Booking, Customer, Course
	Logical Data Type	25 data types
Behaviour	Use Case	Save_Attendee, Book_Attendee_on_Course, Create_Customer, Search_Customer, Search_Course
Interaction	Dialog	Overbooking, BookAttendeeOnCourse

The results of the specification process is an analysis model. As shown in Table 3 the model consists of three conceptual components and one logical data type model. Six use cases together with two dialogs were specified. Especially the use cases specify the way how the tasks of the ski school (like booking a ski course by an attendee) will be performed after the system is implemented.



In the next subsections we briefly introduce the artefacts of the modelling approach based on [SSE09] and the results of the industry research project conducted within this thesis. We will also provide exemplary figures from the running example introduced here. Each artefact will be described through the *definition*, *content* and *form*. We omit the *process* descriptions, because for this thesis we are not interested in how the different artefacts have to be created, but more how to use them for test design.

#### 2.4.5 Use Cases

##### *Definition*

A use case specifies one or more steps of a high-level business process model. Through the specification of several steps performed by an actor or the system, the behaviour of an application is defined. Since the actor interacts with the system, the interaction is also defined. This way use cases show the externally visible as well as the detailed internal application behaviour from the user perspective.

In the literature body of knowledge, early approaches as the one from Ivar Jacobson [Jac92] identified the need for requirements elicitation and SRS specification with use cases. One of the first approaches showing how use cases can be modelled and (model-based) tested was introduced by Mario Winter in his dissertation [Win99]. In the context of agile software development, the use case-based modelling approach from Alistair Cockburn [Coc01] is widely used.

The use cases belong to the behaviour modelling viewpoint. Despite the specification of the interaction with the system, they are not used for modelling the structure and behaviour of the user interface. This aspects are modelled within dialogs (part of the interaction modelling viewpoint), which will be described later.

##### *Content*

Use case models consist of a structured textual description together with a graphical representation as a flowchart. The following attributes are required to specify the content of a use case:

**TITLE** Unique identification of a use case

**BRIEF DESCRIPTION** What are we talking about?

**ACTORS** Who triggers the use case?

**TRIGGER** Who causes the use case to be executed? What is the reason?

**PRECONDITIONS** What conditions need to be met at the outset?

**SCENARIOS** What happens during the process?

**RESULTS** What information is supplied by the use case?

**EXECUTION FREQUENCY** How often is this use case executed within the application?

**QUALITY REQUIREMENTS** What are the quality requirements (for example performance, usability, etc.) for this use case?

**NOTES** What other relevant information should be included?

Some of the mentioned attributes like title, description, trigger, preconditions, execution frequency, quality requirements and notes are self-explanatory. For the other attributes we provide a short description.

Actors play an important role in specifying use cases. They can be human and technical users which trigger the use case. We distinguish between primary and secondary actors. Only the primary users have the permission to execute the whole use case. The distinction is important as there can be several scenarios of how a use case is executed.

In a use case description both the standard and alternative scenarios are defined. The standard scenarios describe the error-free execution case, which suits the objective of the use case. The alternative scenarios describe special execution cases, for example in the case of special constraints or logical error cases.

The results of a use case include the description of the affected entities and the description of return values. The first one describes the state of all entities affected by the use case. The second one describes the data which is returned to the use case caller. Both result descriptions are high-level descriptions instead of technical details. The return values are specified for the standard and each alternative scenario separately. The precise specification of use case preconditions and results is important for the testability of use cases [Jun99, Bin99].

### Form

Besides the textual descriptions and the according attributes, the use case can be represented by using a graphical notation. For this purpose the UML use case and activity diagrams are used.

In Figure 17 we show an example of an use case diagram, which gives an overview of all use cases belonging to a conceptual component. Such diagrams include the primary and secondary actors involved in the use case execution and depicts the dependencies between several use cases. In this case there exists only one primary actor *Employee*. He can call the two use cases *Search\_Customer* and *Book\_Attendee\_on\_Course*. The last use case includes *Search\_Customer* and another use case called *Search\_Course*. We restrict the usage of use case dependencies in our modelling approach to *include*. Especially the underspecification of dependencies as *extends* [Obj09, p.592] would lead to additional problems here.

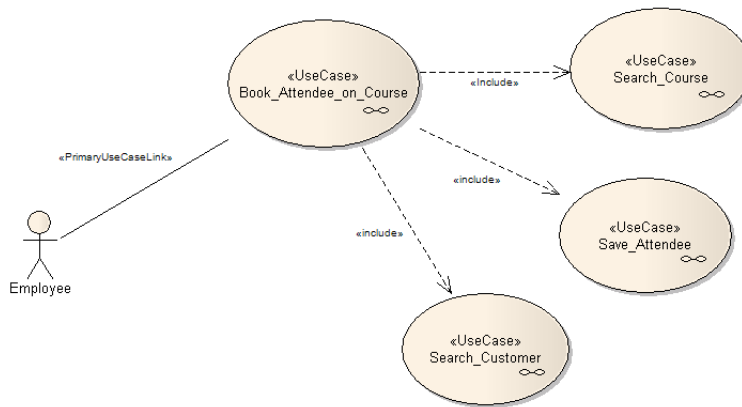
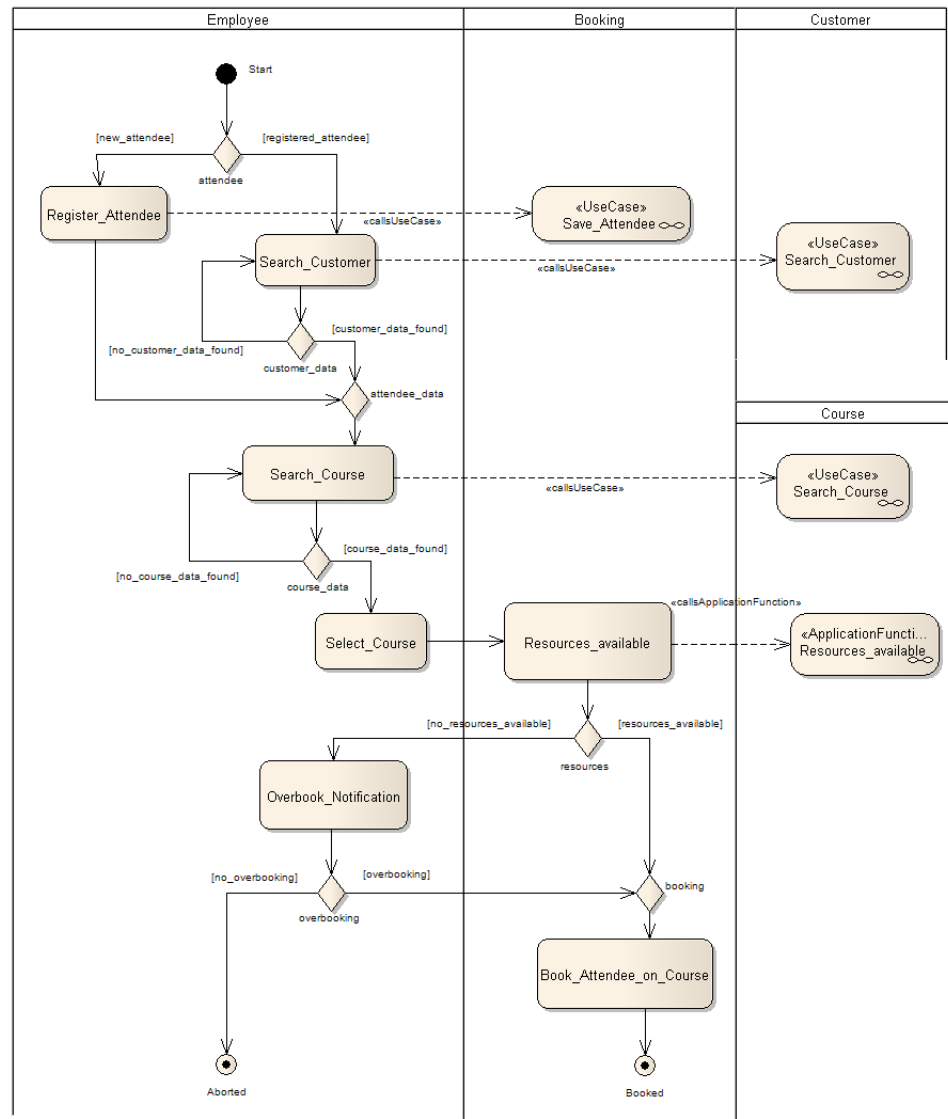


Figure 17: Part of the use case overview from the running example

Each of the use cases from Figure 17 can be refined with a UML activity diagram. An example for the use case *Book\_Attendee\_on\_Course* is shown in Figure 18. Each swimline represents an actor of the use case (here *Employee*) or the involved conceptual components (here *Booking*, *Course* and *Customer*). The action nodes represent single steps performed by the different actors. Each path from the initial to the final node represents a scenario of the use case. As there is no notation for specifying the standard and alternative scenarios in activity diagrams, further textual description is needed.

Figure 18: UML activity diagram for use case *Book\_Attendee\_on\_Course*

The purpose of the use case is to book a attendee on a ski course. First, a new attendee is registered (action *Register\_Attendee*) by the ski school employee or an existing customer is selected. The existing customer is selected from the customer database by using another use case called *Search\_Customer*. Then, according to certain criteria (as course type, date, etc.) the employee searches for relevant courses by using another use case called *Search\_Course*. If the appropriate course is found, the employee selects it and triggers the system to check its availability with the *Resources\_available* system action. Here a so called application function (see next subsection) is invoked. If the course is already overbooked a notification is displayed and the employee has to decide whether to book the attendee on an overbooked course or not (see actor action *Overbook\_Notification*). Finally the system books the attendee on the selected course within the *Book\_Attendee\_on\_Course* system action.

#### 2.4.6 Application Functions

##### *Definition*

Application functions describe the logical calculations of possible complex functions which are executed within a use case. The functions are executed without application interruption, for example by using interfaces of external applications. The interfaces of conceptual components are offered as application functions.

Like the use case description an application function also describes a flow of steps to be executed. The difference is that the steps are primary executed by the application, not the user. The focus of an application function lies on the way how the application processes the request.

In this thesis, we do not perform a detailed analysis of application functions for test generation purposes. They are only used in the context of use case system action calls. Therefore the description of the content and form of application functions is omitted here.

### 2.4.7 Logical Data Model

#### *Definition*

The logical data model describes the application data from the logical point of view. Each conceptual component has a logical data model, which consist of entity types and their relationships. Attributes of entity types are described by logical data types (see next subsection).

#### *Content*

As mentioned the logical data model description consists of the entity types, their attributes and relationships among entity types. For each entity type the following information is required:

**NAME** of the entity type

**DESCRIPTION** of the entity type referring to the logical concept formation of the conceptual component

**SPECIALIZATIONS** is a list of all existing specializations of the corresponding entity type

**STATE MODEL** is an optional description of possible states and the transitions between them for critical entity types

**QUANTITY STRUCTURE** provides information about the resulting data volume that will exist. For example 1000 ski courses are booked per month in the "Gabi's Ski School".

For the attributes of each entity type, minimum the name and description has to be provided. The information about the quantity structure (if specified), can be used for non-functional testing as load testing [BMo8].

The relationship types for the logical data model are associations, compositions or aggregation, specialization or generalization (same as introduced by the UML [Obj09]).

#### *Form*

For the specification of the logical data model the graphical representation in form of UML class diagrams together with further textual descriptions are used.

### 2.4.8 Logical Data Types

#### *Definition*

As mentioned above, the logical data types are used during the definition of entity type attributes. They define the types and value ranges of them. The main difference between entity and data types is that a data type is not modifiable, but represents a pure value.

#### *Content*

The specification method distinguishes between the following data types:

**BASIC DATA TYPES** form the atoms of the conceptual data model.

Examples are number, string, date or time

**ENUMERATION TYPES** are explicit declarations of data type. Ex-

ample is the currency type = euro, dollar or ruble

**SEMANTICALLY MORE SIGNIFICANT TYPES** are usually formed

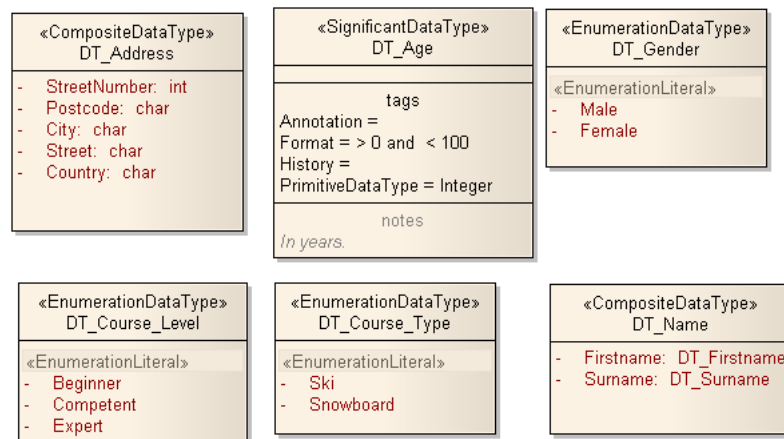
by establishing concrete restrictions for the appearance of their elements. Example is the telephone number

**SUBTYPES** are equivalent to the specialization concept of entity types

**ASSEMBLED DATA TYPES** are conceptual compositions of existing data types

**COLLECTIONS** hold several instances of the same data type

In Figure 19 a subset of the "Gabi's Ski School" logical data type model is shown. The type *DT\_Address* specifies the customer address with a street number, postcode, city, street and country. The significant data type *DT\_Age* defines the allowed age for a ski course attendee within the boundary (0,100) of years. The gender of an attendee can be male or female, which is specified by the *DT\_Gender* data type. There are three levels of ski courses specified by the *DT\_Course\_Level* enumeration data type. The two main course types are specified by *DT\_Course\_Type*. Finally, the name of the attendee is composed by the first and the surname, which is specified by the *DT\_Name* composite data type.



**Figure 19:** Subset of the logical data type model for the "Gabi's Ski School" project

### Form

The logical data types are not defined for single conceptual components, but for the whole application scope. As in the case of the logical data model the UML class diagram together with textual descriptions are used.

### 2.4.9 Dialogs

#### Definition

A dialog is a self-contained element of the user interface, which is used by the application to support a user action by providing a meaningful part of its functionality for interactive utilization. Within business information systems, dialogs mainly enable the interaction of the user with the application.

Single dialogs can be structured into sub-dialogs. Each sub-dialog is defined as an cohesive area of the screen, which the user can interact with. In this thesis we omit the topic of sub-dialogs.

#### Content

A dialog specification consists of static (layout) and dynamic (behaviour) characteristics of the dialog. Both views are strongly re-



lated as the static elements are used and are influenced by the dynamic view.

The static view consists of **dialog elements**, which are defined as individual fields for the input or output of information by or for the user. Depending on the window type in which the dialog is used, information about the resizing and model/non-modal dialogs have to be specified. Further, plausibility checks and calculations for dialogs can be specified.

The main element of the dynamic view are **dialog actions**. Such actions are triggered by a user when activating a specific control element, keyboard event or while all predefined dialog elements are filled with data. The dialog actions correspond to one or more user actions of a use case.

The dynamic view is further described by presentation and dialog states. The first one usually refers to sub-dialogs, for example "the checkbox is selected". The dialog states refer to the dialog as a whole. They describe the different states of a dialog and the transitions between them.

### *Form*

As for other modules also the dialogs can be described by structured text supplemented by graphics. This is mainly the case while specifying the static information. As shown in Figure 20 this specification can be supported by using a custom UML diagram. In this case it is a custom diagram created with the Enterprise Architect <sup>2</sup> modelling tool. The dialog elements are depicted by rectangles, which correspond to objects with predefined stereotypes (like button, input field, checkbox, etc.). The Employee actor of the "Gabi's Ski School" project can search for courses by selecting a course type (see selectbox of *TypeOfCourse*), filling data in the input fields *LevelOfCourse*, *CourseFrom* and *CourseUntil*. Finally, he can trigger the search by clicking on the *SearchCourse* button. The results are displayed in the table according to the *CourseName*, *DateFrom* *DateUntil* and *CourseUnits* columns. After selecting a course, he can choose the detailed course units in the second table. The Employee can also register a new attendee in the elements placed in the left-bottom part of the dialog. He can also search for an existing customer with the

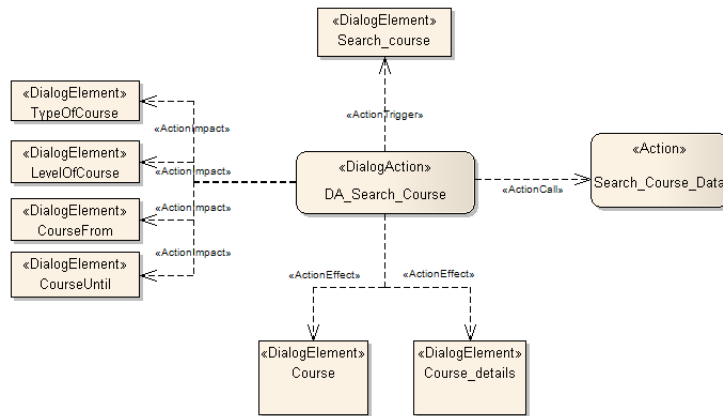
<sup>2</sup> <http://www.sparxsystems.com/>

*SearchCustomer* button. Finally, he can book the attendee with the *BookAttendee* button or quit the dialog.

For the dynamic view the UML activity and state diagrams can be used. In Figure 21 we show an example for dialog action called *DA\_Search\_Course*. Here a UML communication diagram is used to specify the dialog elements serving as a trigger (here *Search\_Course* dialog element) or input (several dialog elements related through *ActionImpact* links) and output (dialog elements *Course* and *Course\_details* related through *ActionEffect* links) of the dialog action.

Figure 20: Layout model of the *BookAttendeeOnCourse* dialog

In our approach we focus on the modelling of dialogs with activity diagrams. UML state machines are a powerful language with a clear formal transition system to describe the behaviour of systems with respect to the states and transitions between them. Since the dialog modelling purpose is only to represent the dialog actions (behavioural part) and the dialog elements



**Figure 21:** Example of a dialog action *DA\_Search\_Course* and the related dialog elements

(structural part) and not the complete behaviour of the system, the appropriate solution are UML activity diagrams. In this thesis we focus on the systems' behaviour, which is described with the use case and application function concept (see previous subsections).

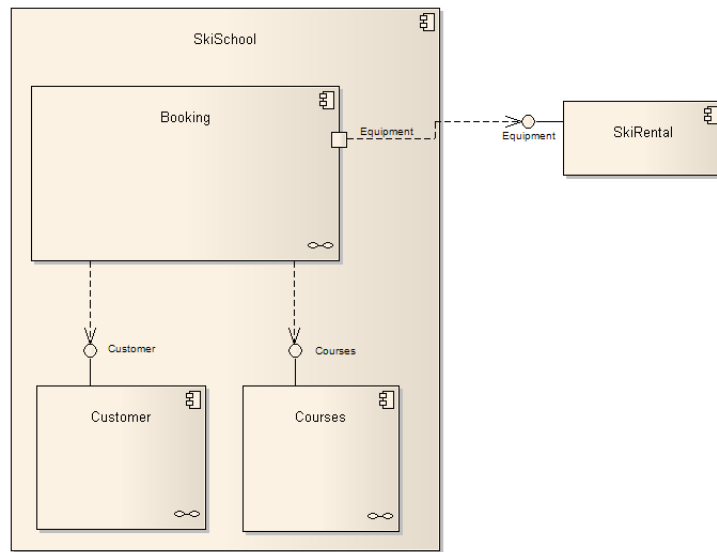
#### 2.4.10 Conceptual Components

##### *Definition*

"Conceptual components are the fundamental concept of structuring the software specification and grouping its artefacts. They are derived from the topics of the problem domain" [SSE09, p.130]. The problem domain is defined in the user requirements specification. The process of structuring the application is also called functional decomposition. The main idea is to use conceptual aspects to structure the application and therefore segment the business logic. The conceptual components are used to manage the (conceptual) complexity of the application. Conceptual components are also used in approaches as Mattsson et al. [MLLF09].

##### *Content*

The functional decomposition into conceptual components results from grouping behaviour artefacts like use cases and appli-



**Figure 22:** Example of a functional system decomposition into conceptual components

cation functions together with structure artefacts like the logical data model. As shown in Figure 22 the set of behaviour and structure artefacts fulfills a certain conceptual aspect or requirement like booking (*Booking* component), customer management (*Customer* component) or course management (*Courses* component). Additionally an external component *Ski Rental* was identified, which communicates with the system called *SkiSchool*. Each of the mentioned artefacts belongs to one and only one conceptual component.

The conceptual components can communicate with each other and also with external systems (see *SkiRental* in Figure 22). The communication is handled through logical interfaces<sup>3</sup>. The interfaces visualize the communication between use cases of conceptual components. Each interface is defined by an application function, which were introduced earlier.

There exist two types of interfaces: offered and required logical interfaces. The first ones define the behaviour and data elements offered to the environment of a conceptual component. The second ones defines the behaviour and data elements expected from

<sup>3</sup> Logical interfaces are not technical interfaces as the once used in the technical design or implementation models.

the environment. In Figure 22 the component *Customer* offers a logical interface called *Customer*.

### *Form*

The decomposition of the application into single conceptual components, which communicate through interfaces is modeled with the UML component diagram. Textual descriptions of components and interfaces motivate and refine the diagram. Those descriptions can be specified with structured text or table form.

#### 2.4.11 Artefact Meta-Model

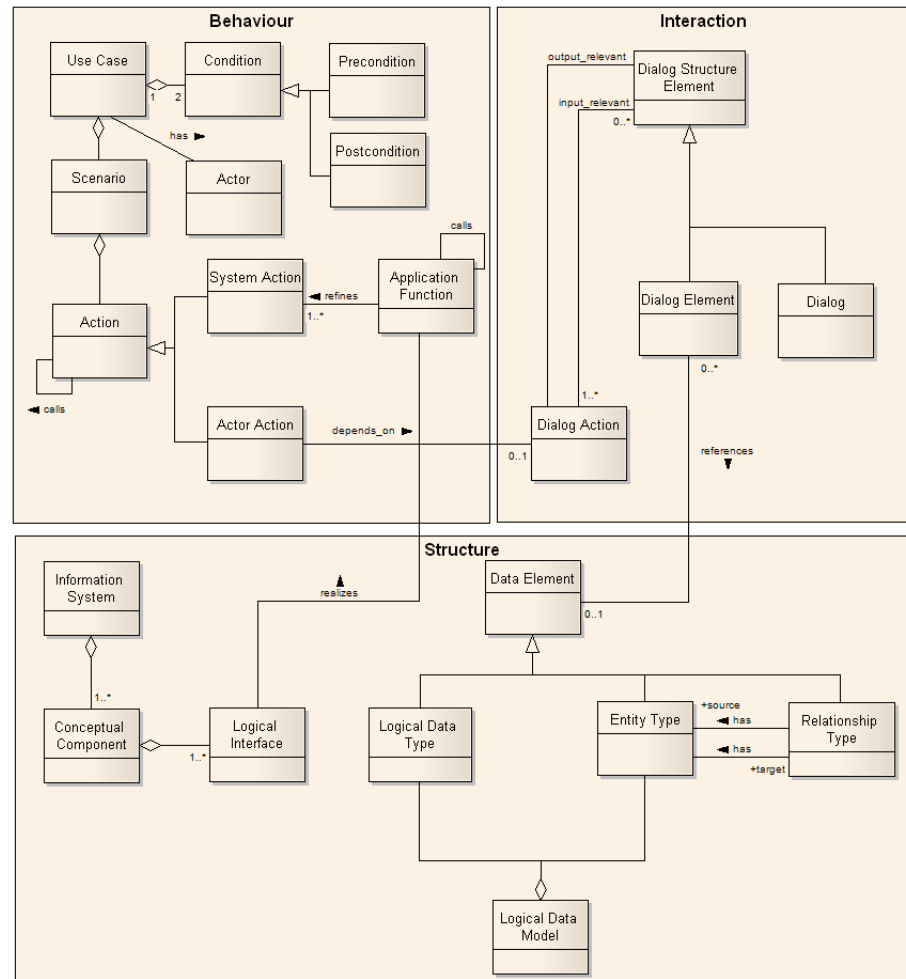
In this subsection, we summarize the introduction of several models in the last sections. We do it by introducing a meta-model for the artefacts part of the specification method relevant for test model generation. The literature about meta-modelling as [AK03] or [SK06], distinguishes between two types of meta-models: linguistic and ontology. The meta-model used in this thesis is a ontology meta-model and is depicted in Figure 23.

The use case model is depicted in the left upper corner. As mentioned in Subsection 2.4.5 each use case consists of one or more scenarios. In each scenario a primary actor performs several actions. Beside the actor actions there also exist system actions. Those are refined by application functions. An action always calls other actions, which are modeled with an edge in the UML activity diagram (see Figure 18).

The dialog model is depicted in the right upper corner. The main parts are the dialog actions and dialog elements. The composite pattern in Figure 23 depicts the hierarchical structure of dialogs. The dialog elements can act as input or output relevant for the execution of dialog actions as explained in Subsection 2.4.9.

The logical data model (Subsection 2.4.7) and logical data types (Subsection 2.4.8) are summarized in the right bottom corner. Different as the data types, the entity types can be related to each other by several relationship types.

The functional decomposition mentioned in Subsection 2.4.10 is depicted in the left bottom corner. The application scope is called information system. It consists of one or more conceptual com-



**Figure 23:** Relevant part of the artefact meta-model of the exemplary analysis modelling approach used in this thesis

ponents. Each conceptual component groups the use case, dialog and data model. Conceptual components have logical interfaces, which are realized by application functions.

Within the artefact meta-model, relations are defined by the UML concept of Association [Obj09, p.39]. There exist three very important relationships in this meta-model which are relevant for this phd thesis:

1. Relation *depends\_on* between the *actor actions* of the use case model and *dialog actions* of the dialog model.
2. Relations *input\_relevant* and *output\_relevant* between the *dialog actions* and *dialog elements* (both input and output relevant).
3. Relation *references* between the *dialog elements* and *logical data types*.

Those relations define a context, which can be used for extracting information needed to specify high-quality test cases. Since the relations are instantiated as model links, the holistic approach for model-based testing can navigate through the analysis model to extract relevant information. The importance of those relations for the holistic view in model-based testing will be explained in later chapters of this thesis.

## 2.5 MODEL TRANSFORMATIONS

The context of this phd thesis is dealing with the model-based testing scenario, where test models are automatically derived from test model. In the last sections we have introduced several meta-models for for both models. Having this knowledge, we can use the technique of model transformations widely-known from the model-driven development (MDD) and model-driven architecture (MDA) domain. Within MDA different models on several abstraction levels exist. The *model transformations* are a technique for transforming models between different abstraction levels [GPR06]. Since the models used in this thesis have different abstraction levels as described in Section 1.1, this technique is suitable for the automated derivation of test models from analysis models.

In this section we use the taxonomy for model transformations from Mens et al. [MVGVKo6] to categorize them. Based on the requirements derived from the research problems introduced in Chapter 1, we select the model transformation type and language to be used in this thesis.

### 2.5.1 Definitions

To provide a common understanding of model transformations, we provide some basic definitions here:

**Definition 16 Model transformation** - is "the automatic generation of a target model from a source model, according to a transformation definition" [MVGVKo6, p.5]

**Definition 17 Transformation definition** - is "a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language" [MVGVKo6, p.5]

**Definition 18 Transformation rule** - is "a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" [MVGVKo6, p.5]

### 2.5.2 Categorization

According to Gruhn et al. [GPRo6], we distinguish between the two general types of model transformations:

- **Model to Model (M2M)** - the input and output of a transformation is a model
- **Model to Text (M2T)** - the input of a transformation is a model, but the output is a textual description (for example text files with code)

In the context of MDA [Obj03] the definition of a *platform-independent model (PIM)* and *platform-specific model (PSM)* is used to differ the models with respect to the level of information about the platform in which the developed system is used. Through one or several M2M or M2T the PIM is transformed into a PSM. In this thesis the generated test cases have to be executed in a test envi-



ronment. The test model is a PIM model and the executable test cases a PSM.

A more detailed categorization is introduced in the model transformation taxonomy from Mens et al. The authors distinguish in [MVGVKo6] between:

- **Number of source and target models** - a transformation can have **one source** and **one target model**. Another type are multiple source models and/or multiple target models transformation. A combination between single and multiple models used as source or target are also possible.
- **Endogenous or exogenous** - depending on the meta-models used in the model transformation to express models the **endogenous** (same meta-model for source and target models) and **exogenous** (different meta-models) exist
- **Out-place or in-place** - the target model resulting from the execution of model transformations can be **out-place** (different source and target model instances) or **in-place** (one model instance for source and target)
- **Horizontal or vertical** - depending on the abstraction level, the model transformations can be **horizontal** (same abstraction level for source and target) or **vertical** (different abstraction levels)
- **Preservation** - the target model always preserves certain aspects of the source model. For example transformations, which refactor models preserve the behaviour of the source model

Based on the problem statement from Section 1.1, we identify the following type of model transformations to be used in this thesis:

*Transformations in this thesis*

- *M2M and M2T* since we aim to automatically derive test models from analysis models (M2M) and further derive test cases from test models (M2T)
- *One source / multiple target models* since we always use one source (like the analysis or test model) for transformations, which result in multiple target models (for example test and trace models)
- *Exogenous* since we use different meta-models for the models used in our transformations

- *Out-place* since our goal is to automatically derive a separate test model
- *Vertical* since the analysis and test model have different abstraction levels
- *Preserving behaviour* since logical test cases automatically selected from the analysis model preserve the behaviour of the use case as linear sequence of actions

The requirements mentioned above, will be refined in more detail while presenting the research approach of this phd thesis.

### 2.5.3 Traceability Issue

While there exist different types of model transformations, they all transform source into target models. This transformation step reveals important information, which can be used to trace the elements of the target back to the source model. Such traces are used to measure the coverage of the analysis model by the automatically generated test model. The model coverage measurement is one of the research problems (see Section 1.1). In this thesis we use the following definition of traceability.

**Definition 19** **Traceability** is "the ability to trace the connection between the artefacts of the testing life cycle or software life cycle or software life cycle; in particular, the ability to track the relationships between test cases and the model, between the model and the informal requirements, or between the test cases and the informal requirements" [UL07, p.407].

The model coverage measurement inspects the traceability between test cases and the model. Besides the test behaviour viewpoint of the UTP represented by test cases also the test architecture and test data viewpoints should be traceable back to the analysis model.

To use the traces between source and target models for model coverage measurement, they have to be made explicit. This is important, since we aim to develop algorithms which can measure the reached model coverage based on traceability information. This way the support for traceability is a further requirement for the selection of a model transformation language in the next subsection.

#### 2.5.4 Model Transformation Languages

In the last two subsections we have introduced the requirements for model transformations used in this thesis. Now, we will compare several model transformation languages and select one to be used in this document.

In general, two types of transformation languages exist:

- **Declarative** transformation languages define relations between source and target models; this way they focus on the *what* has to be transformed
- **Operational** transformation languages define the steps (and their order) to be performed to execute a transformation; this way they focus on *how* transform

Since the automated derivation of test models from analysis models requires several tasks (as test selection, navigation through models, etc.) additionally to model transformation, we need a hybrid transformation language which is both declarative and operational. The declarative aspect is needed for pure model transformations between the analysis and test model. The operational aspect is needed to perform transformation together with the mentioned additional tasks.

We have identified the following model transformation languages, which suits most of the requirements mentioned earlier:

- Query/View/Transformation (QVT) Language - standardized transformation language provided by the Object Management Group [Obj08a]
- Atlas Transformation Language (ATL) - transformation language provided by the Eclipse M2M project<sup>4</sup>
- Epsilon Transformation Language (ETL) - transformation language provided by the Eclipse Epsilon project<sup>5</sup>

This set of model transformation languages is based on a broad evaluation performed in three students thesis (see [Nie09, Hey10, Fic10]). Additionally, transformation frameworks as openArchi-

<sup>4</sup> <http://www.eclipse.org/atl/>

<sup>5</sup> <http://www.eclipse.org/gmt/epsilon/>

tectureWare<sup>6</sup>, MOFScript<sup>7</sup> and JET<sup>8</sup> were analyzed. All three languages can only be used for M2T.

In Table 4 we compared the three modelling languages. To perform M2M transformations (analysis to test model), all languages can be used. However, to additionally perform M2T transformations (logical to concrete test cases) only the ETL seems suitable. The M2T in ETL are possible through the Epsilon Generation Language (EGL)<sup>9</sup>. In EGL the target of the transformation is specified within a scripting language rather than a predefined meta-model.

**Table 4:** Comparison of suitable model transformation languages

Criteria	QVT	ATL	ETL
General	M2M	M2M	M2M/M2T
Models	multiple-source / multiple-target	multiple-source / multiple-target	multiple-source / multiple-target
Meta-models	exogenous	exogenous	exogenous
Place	in-place / out-place	in-place / out-place	in-place / out-place
Abstraction	vertical	vertical	vertical
Type	declarative / imperative*	declarative / imperative	declarative / imperative
Traceability	none*	built-in, not changeable**	built-in, changeable**

\* - this property depends on the concrete implementation of QVT

\*\* - changeable with customized transformation rules

As depicted in Table 4, there exist only slightly differences between the languages. First, the QVT language has no support for traceability in terms of an explicit trace model. The QVT specification mentions a trace model [Obj08a, p.145-146], but only one concrete implementation, namely mediniQVT<sup>10</sup> supports it. Unfortunately, no editable trace meta-model is given in QVT. The

<sup>6</sup> <http://www.openarchitectureware.com/>

<sup>7</sup> <http://www.eclipse.org/gmt/mofscript/>

<sup>8</sup> <http://www.eclipse.org/modeling/m2t/?project=jet>

<sup>9</sup> <http://www.eclipse.org/gmt/epsilon/doc/egl/>

<sup>10</sup> <http://projects.ikv.de/qvt/>

ATL supports the traceability with a so called Weaving Model. The problem here is that the adaption of the Weaving meta-model is very complicated from the technical point of view (see the evaluation in [Fic10]). To overcome this problem, we have customized the model transformation rules itself to generate an explicit trace model. For example in ETL the specification of pre- and post-blocks for each transformation rule is possible. An extension of ATL transformation rules is also possible and proposed by Jouault in [Jou05].

Based on the comparison described above, we have identified ETL as the model transformation language used in this thesis.

### *Feasibility studies*

The definition of requirements for a model transformation language and systematic selection of a concrete language is important for the definition of a holistic approach for model-based system testing. For each of the mentioned languages, we have performed a feasibility study. For that, we have used analysis models collected within the industry research project related to this thesis. We have observed that standard modelling tools used in the industry as Enterprise Architect<sup>11</sup> exports models in formats not compatible with the widely-used XMI [Obj07a] format. This format is used as input by all mentioned model transformation languages. That is why, the prototype implementation from [Nie09, Hey10, Fic10] used for evaluation purposes is based on standard programming and scripting languages. Still, the requirements described in this section are fulfilled in the prototype implementation. The implementation with pure ETL frameworks is part of the future work on this prototype.

## 2.6 SUMMARY

In this chapter we have introduced all preliminaries and definitions of this phd thesis. First, we have introduced the Fundamental Test Process, its artefacts and basic testing definitions. Then, the domain of model-based testing, its artefacts and process and the essential test selection techniques were presented.

<sup>11</sup> <http://www.sparxsystems.com>

The UML Testing Profile as the test modelling notation was also introduced. Further, we have presented the representative modelling approach for business analysis used in this thesis. Finally, the topic of model transformations was covered.

To underpin the research problems of this phd thesis, we present a detailed survey of the related work in the next chapter.

---

RELATED WORK

---

In this chapter, we will introduce the work related to the problems identified in the motivation of this thesis. The literature evaluation presented here has the goal to *identify MBT approaches which use several modelling viewpoints of analysis models for generating high-quality test models and test cases for system testing*. It was done in a systematic and structured way. First, we have analyzed the known surveys of model-based testing approaches as [DNSVT07] and [GECMT05]. Then, we supplemented the list of relevant approaches found in [DNSVT07] and [GECMT05] with our own survey. We searched with the research search engines as IEEE Digital Library<sup>1</sup>, ACM Library<sup>2</sup>, Springer Link<sup>3</sup> and Cite-seerx<sup>4</sup> for papers according to keywords related to the thesis contribution points as: *model-based testing, UML, business information systems, system testing, holistic, test model, modelling viewpoints and model relations*. After a qualitative selection (citation count, maturity level, publication form, etc.), we ended up with a list of almost 60 approaches.

In the following sections we will first explain the evaluation criteria and refer to their related work. Then, we show a comparison table for better literature overview. Finally, we briefly introduce the approaches related to the problems defined in Chapter 1.

---

CONTENTS

---

3.1	Evaluation criteria . . . . .	74
3.2	Identified related work . . . . .	84
3.3	Summary . . . . .	101

---



---

<sup>1</sup> <http://ieeexplore.ieee.org>

<sup>2</sup> <http://portal.acm.org/>

<sup>3</sup> <http://www.springerlink.com/>

<sup>4</sup> <http://citeseer.ist.psu.edu/>

### 3.1 EVALUATION CRITERIA

In order to compare the found approaches, we derived a list of 11 + 2 evaluation criteria based on the problem definition from Chapter 1. The additional "+2" criteria focus on the overall quality of the presented approach in terms of provided empirical evidence and tool support. For better understanding, we mapped the evaluation criteria in Figure 24 to the "before" part of the contribution figure from Section 1.2.

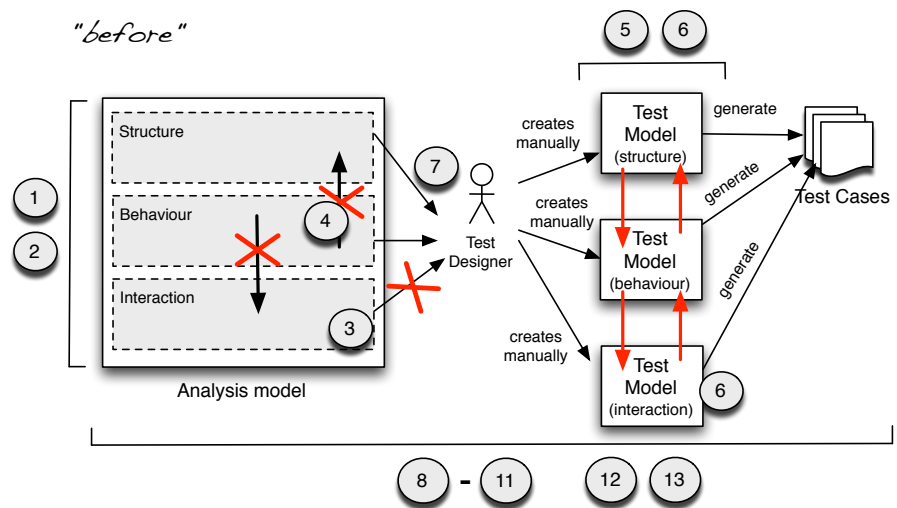


Figure 24: Evaluation criteria mapped to the research problem figure

The numbers from Figure 24 symbolize the following evaluation criteria:

#### MODEL USAGE

1. **UML for system modelling** - does the approach use UML for modelling the analysis model?
2. **Modelling viewpoints** - does the analysis modelling approach use the three viewpoints describing the structure, behaviour and interaction with the system?



3. **Integrated interaction viewpoint** - does the approach use the interaction modelling viewpoint (especially models of the GUI) integrated into the analysis model?
4. **Model Relation** - does the approach use model relations between the structure, behaviour and interaction modelling viewpoints for test generation?
5. **Test Model** - does the approach use a separate test model?
6. **UML for test modelling** - does the approach use UML for test modelling?
7. **Developer Model** - does the approach use the UML analysis model created by developers for test generation?

#### INTERNAL TEST QUALITY

8. **Understandability** - are the generated test cases understandable?
9. **Analysability** - are the generated test cases analysable?
10. **Completeness** - are the generated test cases complete w.r.t. the information needed for test execution?
11. **Traceability** - are the generated test cases traceable to the source models and/or requirements?

#### RESEARCH QUALITY

12. **Case study** - does the paper provide an evaluation of the approach by means of one or more case studies?
13. **Tool** - does the approach provide tool support?

The evaluation criteria have been selected according to the problem definition from Section 1.1. In this section we introduce each criterion and reference to the relevant literature.

##### 3.1.1 UML for system modelling

The specification of complex software as the Business Information Systems (BIS) requires modelling notations, which can deal with different abstraction levels and functionality types. The Unified Modeling Language seems to be the non plus ultra while modelling software systems [HMo8]. The UML is widely used

for modelling BIS. As mentioned by [BL02], there exists no standard UML modelling approach for BIS. We selected the one introduced by Salger et al. [SSE09], which was already introduced in Section 2.4.

The usage of UML for system modelling in the evaluated approaches is important, because we focus on modelling approaches similar to [SSE09]. Further, we focus on system testing (see Chapter 1) for which the abstraction level provided by analysis models is needed. Low-level models of the system's design or implementation are not focus of the model-based testing approach introduced in this thesis. This is based on the fact that within system testing, high-level workflows rather than technical details are considered for test analysis and generation purposes.

The system modelling with UML is also widely used in model-based testing. As shown by the survey from Dias Neto et al. [DNSVT07] the majority of MBT approaches identified since 1990 uses UML diagrams for test case generation. The notation used by several approaches are statecharts. For example [OA99, Wei09, HN04, PJJ<sup>+</sup>07, UL07] apply different state, transition and data-flow coverage criteria to statecharts for generating test cases. Similar or as mentioned by Utting et al. [UL07] the same coverage criteria can be applied for UML activity diagrams. Good examples for such approaches are [BL02, GECM<sup>+</sup>09, HVFR05, MXX06, VLH<sup>+</sup>06, DSWO04, Bin99]. The other kind of MBT approaches concentrates on UML sequence diagrams (as [FL02, NFTJ06]) or UML class and object diagrams, as [GMWE09] or [Bin99].

### 3.1.2 Modelling viewpoints

As motivated in Chapter 1 there is a strong need to use several modelling viewpoints (describing the structure, behaviour and interaction view of the system) for system level test generation. This need was also recognized by Deng et al. in [DSWO04] where a tightly-coupled model for software engineering was introduced. The authors claim that this kind of model (using several diagram types and abstraction levels) enables a more complete testing approach.

In the literature only few approaches use several diagram types of the different viewpoints for testing. Briand and Labiche show

in *"A UML-Based Approach to System Testing"* [BL02] how to use UML analysis artefacts as use cases, sequence and collaboration diagrams, class diagrams and Object Constraint Language (OCL) [Obj06b] expressions among those artefacts. Another approach was introduced by Vieira et al. in *"Automation of GUI Testing Using a Model-driven Approach"* [VLH<sup>+</sup>06]. The authors use UML use case, activity and class diagrams. The use case diagram is always refined by an activity diagram, which is extended by notes to connect it with the class diagram. The last approach *"UML-based integration testing"* was introduced by Hartmann et al. [HIM00]. The main focus is to generate test cases for integration testing by using UML statecharts refined with the CSP (Communicating Sequential Processes) language. The usage of CSP [Hoa85] can be compared with the usage of OCL refinement in [BL02].

The usage of several modelling viewpoints is an important requirement for our evaluation, because we aim to develop a holistic testing approach, which uses information from different modelling viewpoints.

### 3.1.3 Integrated interaction viewpoint

Graphical user interfaces (GUIs) are one of the most commonly used parts of today's software [XM08]. Therefore GUI testing is a very important task. To design test cases, which test the GUI the interaction modelling viewpoint of the analysis model can be used. The other possibility is to use separate models created exclusively for GUI testing.

In the past several model-based test approaches have been developed to automate GUI testing as shown in the recent survey from Memon and Nguyen in [MN10]. Early approaches as [SS97] used finite state machines to generate test cases. Later on models as the event-flow graph [MSP01] and event-interaction graph [XM08, MN10] were used. Recent approaches as [BM10] also use the control-natural language (CNL) to generate test cases for GUI testing. Except for [BM10] all GUI testing approaches propose to build a test model separated from the system model.

The usage of GUI models as the interaction modelling viewpoint is an important requirement for our evaluation, because we fo-

cus on BIS where the user interacts with the system through a GUI.

#### 3.1.4 Model relations

Analysis models as the one created in Salger et al. [SSE09] show that several elements as class and activity diagrams are not orthogonal, but strongly related to each other. Those relationships are often created between diagrams belonging to the different modelling viewpoints. Deng et al. in [DSWO04] states that there exist several relationships between model elements. Also Binder in [Bin99] discusses the topic of model relations. Both sources state that the relations can be effectively used for testing, as test cases validate the existence of relations during test execution.

The usage of model relations is an important requirement for our evaluation, because this way the context for gathering information from the analysis model for generating logical test cases can be identified.

#### 3.1.5 UML for test modelling

Besides modelling of the SUT behaviour and then generating test cases from those models, the UML can also be used for modelling tests as such. This was the main idea for creating the UML Testing Profile (UTP) [Obj07b] (see Section 2.3). There exist several approaches, which use UTP as test model representation. For example Baker et al. in [BRDG<sup>+</sup>08] introduces all UTP concepts and explains how a UML model of a bluetooth protocol can be used to create an UTP model. This work was based on several publications from Zhen Ru Dai as [DGNP04, Dai04, BRR<sup>+</sup>06] and finally her phd thesis in [Dai06]. Other approaches as [CYXZ05] and [LMdG<sup>+</sup>09] also use UTP as the test model notation.

The usage of UML for test modelling is an important requirement for our evaluation, because UML is a widely-known language for modelling BIS. While we already aim to use different modelling viewpoints and UML for business analysis the same modelling language should be applied for testing in order to guarantee process consistency and usability for the test designer.

### 3.1.6 Test Model

Next to the possibility of modelling tests with UTP, there are several publications which deal with the idea of a separate test model. In [PP05], Pretschner and Philipps introduce several scenarios for model-based testing. One of those uses two models: one for the development and one for testing. The authors state that this would be the optimal scenario and would combine the MBT and MDD advantages. The need for empirical studies on this topic is needed.

The test model is the external view of the system, rather than the internal in case of a system model as stated by Malik et al. in [MJV<sup>+</sup>10]. In our literature analysis we have found several approaches using test models for test case generation. Dai et al. in [DGNP04] and [BRDG<sup>+</sup>08] uses UTP for testing a bluetooth protocol. Gross et al. creates in [GSD05] a UTP test model for generating built-in tests. Rumpe shows in [Rum03] how several UML diagrams together with so called test patterns can be used to create a test model. A test model of a GUI as an event-flow graph is used by Memon in [MSP01]. Another MBT approach from Bertolini and Mota [BM10] generates test cases from a test model created by using the controlled natural language (CNL) [CS08]. A survey from Denger et al. in [DM03] shows several test design methods which are primary based on creating separate test models.

The usage of a test model is an important requirement for our evaluation, because test models are more abstract than analysis models and contain additional information (like concrete test data and expected results). In order to specify high quality test cases, this information has to be managed in a separate test model.

### 3.1.7 Developer Model

Test models can be created explicitly for test generation or automatically generated from existing development models. This has a direct impact on the effort while using MBT as shown by Guldali et al. in [GMS10]. The topic of automatic usage of developer models in the context of MBT approaches was introduced by the work of Dai in [DGNP04], [Dai04], [Dai06] and [BRDG<sup>+</sup>08].

Dai uses model transformation rules for creating a test model from a system model. Gross et al. [GSD05] also introduces several model transformation rules to create an older version of the UTP model. The usage of transformations for MBT in the context of the MDA was strongly encouraged by Torres et al. in [TECMGo9] and Gutiérrez et al. in [GECM<sup>+</sup>09]. A semiautomatic usage of developer models was introduced by Vieira et al. in [VLH<sup>+</sup>06] and Hartmann et al. in [HVFR05].

The usage of developer models is an important requirement for our evaluation, because parts of the analysis model are used during test modelling. To lower the test modelling effort automatic test model generation is needed.

### Test Case Quality Attributes

Independent of the source for test generation, the important factor is the quality of test cases<sup>5</sup> being generated. Zeiss et al. in [ZVS<sup>+</sup>07] shows a quality model for test specifications based on the ISO9126 quality standard [ISO04]. The quality model is divided into seven main characteristics, as test effectivity, reliability, usability, efficiency, maintainability, portability and reusability. Two quality attributes as test understandability and analysability are of interest for the research problem of the missing holistic view. Additionally, the completeness and traceability of test specifications (not mentioned in [ZVS<sup>+</sup>07]) have to be analyzed in this context. Like already mentioned in Section 1.1 if the holistic view on the analysis model is missing during the test generation process, then the test case quality with respect to the mentioned quality attributes decreases. Therefore, we consider those quality attributes during the literature survey. We first provide clear definitions for each of them:

**UNDERSTANDABILITY** describes the degree to which a test designer understands if a test specification is suitable for his needs.

**ANALYSABILITY** describes the degree to which a test specification can be diagnosed for deficiencies.

---

<sup>5</sup> By test cases we mean the logical test cases which can be part of a test model (like in UTP), and concrete test cases used for manual or automated test execution.

**COMPLETENESS** describes the degree of information each attribute of a test case (see Subsection 2.1.1) contains, which is needed to execute him.

**TRACEABILITY** describes the degree to which test cases are traced back to the test basis (i.e. elements of the analysis model or requirements specification).

The listed quality attributes have a hierarchical structure, which is depicted in Figure 25. The holistic view influences the *completeness* and *traceability* attributes. The attended extraction of information from several modelling viewpoints results in more complete test models. The traceability to all viewpoints is possible only during the application of the holistic view. Further, the completeness influences the *understandability* and *analysability* of logical test cases within the test model. Finally, the traceability influences the analysability, because each element of the test model can be traced back to the analysis model.

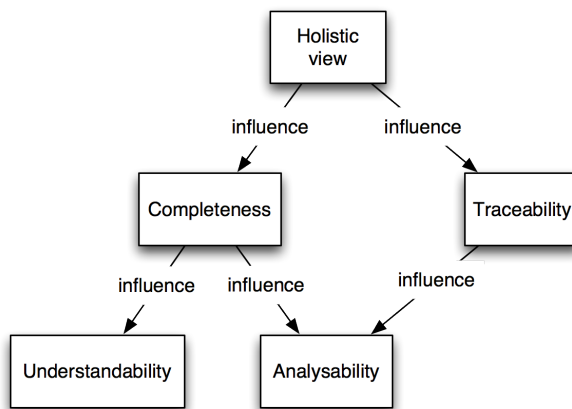


Figure 25: Dependencies between test quality attributes

### 3.1.8 Understandability

The first test quality attribute understandability deals with the sometimes subjective notion of how a test designer understands the purpose and description of a test case. The related quality attribute from Zeiss et. al. [ZVS<sup>+</sup>07] is also test correctness, which denotes the correctness of the test specification with respect to the system specification or the test purposes.

The understandability of test artefacts as logical test cases is important, since test cases have to be understood at every stage of the software development process. The same understandability should be given in new and maintenance testing projects. The completeness quality attribute resulting from the application of a holistic view has a direct impact on the understandability.

### 3.1.9 Analysability

Besides the understandability also the analysability of test specifications is important. Test cases automatically generated from models still have to be analysed for deficiencies. Zeiss et al. mention in [ZVS<sup>+</sup>07] this maintainability characteristic as important. They point out that especially the documentation and good structure of test cases have an influence on the analysability.

The analysability of logical test cases is important, since test designers spend great part of their efforts in understanding and analysing them. The evolving systems, which are build today require low maintainability of all artefacts of the software development process. Logical test cases, which are analyzable through good structure and relations to the underlying test architecture and data improve its maintenance.

### 3.1.10 Completeness

Another test quality attribute is the test completeness. Deng et al. [DSW04] mentions that software testing and maintenance has to be carried out in a more complete manner. The authors show how test cases for functional and regression testing can be generated by using a strongly intra-related model. The authors claim that because the model is more complete than the ones used in existing approaches, the generated test cases are also more complete. While Deng et al. by complete means test cases used for different test types (as functional and regression testing), we define the test case completeness as containing all information needed for test execution. Precisely, all attributes of a test case from Section 2.1.1 are filled with data.

The completeness is important, since it is the main quality attribute, which results from the application of the holistic view. It



has also a direct relation to the understandability and analysability of test artefacts. Since we are interested in the improvement of the internal test quality in this thesis, the completeness of test artefacts is a mandatory aspect.

#### 3.1.11 Traceability

Besides the mentioned test quality attributes, the traceability of test cases is essential for tasks as the coverage measurement or the impact analysis. Different kinds of traceability links used in software development were introduced by Ramesh and Jarke in [RJ01]. In the context of model-based testing some approaches deal explicitly with this topic. For example Naslavsky et al. in [NZR07] shows how to guarantee traceability between UML sequence diagrams, model-based control-flow graphs and test generation logs. The mentioned artefacts are generated sequentially and traceability links are incorporated into the generation process. Unfortunately, the authors do not discuss the relation between traceability, coverage measurement and test case completeness. Another example is the approach by Bouquet et al. in [BJL<sup>+</sup>05], where a model in the B language was annotated with requirement links. While generating test cases from the B models, traceability links to the requirements were incorporated. During the generation process a traceability matrix is created to visualize the reached coverage. As in [NZR07] the influence on other test case quality attributes was not discussed.

The usage of traceability is an important requirement for our evaluation, because each test case should be traceable to the elements of the test basis (analysis model or even single requirements). This way, the the basis from which the test case was derived (especially modelling viewpoints) can be identified and the test coverage be measured.

#### 3.1.12 Case study and tool support

Besides the criteria introduced above, it is also important to evaluate the quality of the analysed paper. For this, we define two further criteria, namely the existence of according case studies and tool support for each approach. Those two quality fac-

tors are widely-used in the research community as described by Mary Shaw in [Shao3]. The tool support criteria is important in the context of model-based testing research to provide empirical evidence and the proof-of-concept of the modelling and generation approach.

### 3.2 IDENTIFIED RELATED WORK

According to the evaluation criteria from the previous section, we have performed a detailed analysis of 33 approaches dating back from 1999 to 2010. The results are shown in Figure 26.

The detailed description of each approach from Figure 26 would be unnecessary long and would not provide the cognitions for this thesis. That is why we define the following seven characteristics to group and describe the found approaches:

1. Generation from system models
2. Generation from several modelling viewpoints
3. Generation using test models
4. Generation of test models from developer models
5. Generation using model relations
6. Generation from GUI models
7. Test case quality attributes

As for the evaluation criteria, we also mapped the different groups of approaches to the underlying research problems in Figure 27. For each characteristic, we briefly describe in the following paragraphs the according approaches and select one or more representatives. Those selected approaches will be introduced in detail and are highlighted in Figure 26.

#### 3.2.1 Generation from system models

The generation of test cases from models can have several scenarios as shown by Pretschner and Philipps in [PP05]. One possibility is to generate test cases from so called system models. We have found that several approaches as [VLH<sup>+</sup>06, PJJ<sup>+</sup>07, HNo4, HNo3, BLo2, GECM<sup>+</sup>09, DSWO04, NFTJ06, Bin99, UL07, FL02]

No.	Reference	Modelling Viewpoints										Internal Test Quality				System Testing	Case Study	Tool
		UML	UML Behaviour			UML Structure			UML Interaction	GUI Models	Model Relation	Test Model	UML Test	Developer Model	Understandability	Analysis-Completeness	Traceability	
		Activity	Sequence	State	Use Case	Class	Comp.	Object										
1	[BBW06]	(x)								x		x					(x)	x
2	[BCOL07]	(x)		(x)		x	x	(x)			(x)	(x)	x					(x)
3	[Bin99]	(x)	x	x	x	x	x	x			(x)							
4	[BL02]	x	x	x	x	x	x				(x)						x	x
5	[BM10]									x		x					x	x
6	[BRDG+08]	x	(x)	x		x	x					x	x	x			x	x
7	[CXYZ05]	x										x	x	x			x	
8	[Dai04]	x	x	x		x						x	x	x			x	(x)
9	[DM03]	(x)			(x)	(x)		(x)				x						
10	[DSWO04]	x	x	x	x	x					x							
11	[FL02]	x		x								x						x
12	[FS05]	x	x											(x)			x	x
13	[GEM+09]	x	x														x	x
14	[GMWE09]	x	x	x	x		x					x	x			x	x	x
15	[GSD05]	x										x						
16	[HGB08]	x	x	(x)	x	(x)					(x)						x	x
17	[HIM00]	x		(x)	x						(x)							x
18	[HVR05]	(x)	x		x	x					(x)	(x)					x	(x)
19	[ISBP07]																	
20	[LMdG+09]	x		x	x	x					(x)	x	x	x		x	x	(x)
21	[MJV+10]	(x)										x	x				(x)	
22	[MSP01]	(x)		(x)						x		x				x	x	x
23	[MXX06]	x	x															
24	[NFTJ06]	x		x	x													x
25	[NRP05]	x		x							x	(x)						x
26	[OA99]	x		x											(x)			x
27	[PJ+07]	(x)	x	x	x	x		x			(x)			x			x	x
28	[PPW+05]	(x)		(x)			(x)									x	x	x
29	[Rum03]	(x)		x	x	x						x		(x)				(x)
30	[UL07]	x																
31	[VLH+06]	x	x		x	x			x	(x)	(x)	(x)					x	x
32	[Wei09]	x		x		x		(x)				x	x	(x)			x	x
33	[XM09]	(x)		(x)						x					x			x

Figure 26: Evaluation Table

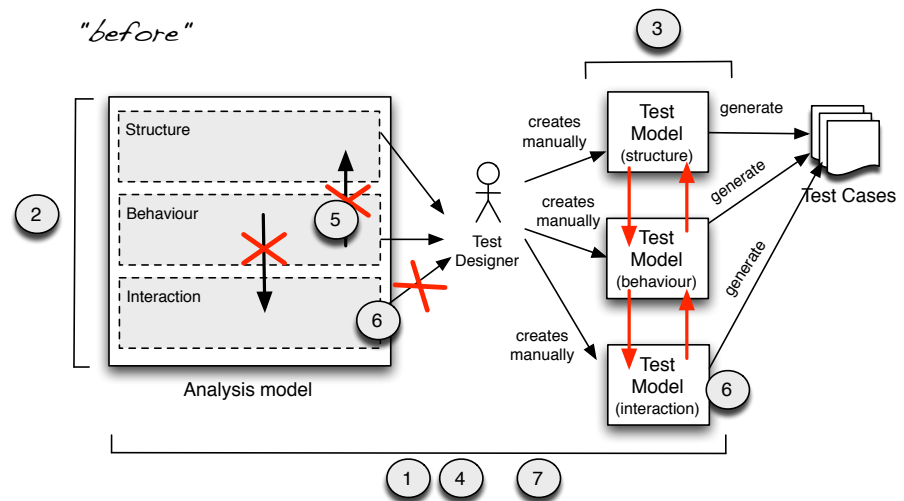


Figure 27: Approach characteristics mapped to the contribution figure

use models of the system behaviour to generate test cases. We assume that those models were not used for code generation as this would cause the problem mentioned by Pretschner and Philipps. If code and test cases are generated from the same model, then the tests are meaningless. Only the functionality of the code generation algorithm could be tested.

We have already mentioned that system models differ from test models in certain aspects, as abstraction level or modelling view. Therefore pure system models cannot be directly used for test generation. Instead additional information has to be added by test designers. In all found approaches the additional information has been added by using special annotation languages, modelling notations (for example OCL), contract-based modelling, etc.

To show how system models can be used for test generation, we selected a representative approach provided by the international research project called AGEDIS which took place from 2001 to 2004. Several publications as [HNo4, HNo3] or [PJJ<sup>+</sup>07] were published by the AGEDIS participants. The general idea was to automate the testing activity (design and execution of test cases) within software engineering projects for object-oriented and distributed systems. The approach uses a proprietary modelling language called AGEDIS modelling language (AML). This lan-

guage is very similar to the UML. The authors use *class*, *state* and *object diagrams* for test case generation. The generation process as described by Pickin et al. in [PJJ<sup>+</sup>07] consists of four steps:

1. Formal specification derivation - on the fly generation of a labelled transition system (LTS) for a UML specification (class together with the related state and object diagrams)
2. Formal objective derivation - definition of a LTS semantic for test objectives which are defined as UML state diagrams
3. Test synthesis - synthesis of the LTS created in 1. and 2. which results in test cases
4. UML test case derivation - a precise definition of a test case (similar to UML sequence diagrams) is derived from the result of 3.

As AGEDIS considers not only the derivation, but also execution and evaluation of the test cases there are more than four steps as introduced above. In the context of this phd thesis the algorithms performed within step 1 and 3 are interesting.

First, the initial UML system specification is transformed into a LTS in step 1. For test generation purposes the so called test objectives (partial state diagrams with regular expressions describing the state sequence to be reached) have to be modeled. The whole approach is based on the synthesis of the LTS with the test objectives. This reflects the observation that pure system models cannot be directly used for test generation.

Second, the usage of several diagram types relates to the idea of the *holistic view*. However the authors do not provide any explanation why those models were used. The assumption is that UML design models can be used for test purposes and the three selected diagram types describe an object-oriented system. The authors assume for example that for each main class a state diagram is given. Further, there exist an object diagram defining the initial state of the application.

As shown in Figure 26 the approach from Pickin et al. [PJJ<sup>+</sup>07] does not use any models of the GUI. The only model relations we could identify was the standard UML typization between the class and state diagrams. Also, no separate test model but the UML system specification together with the test objectives was used in this approach. Because the UML system specification

*Cognitions for this thesis*

is used for test purposes (through the generation of the LTS), there exist a certain usage of developer models. Unfortunately no information about the test case quality (understandability, analysability, completeness or traceability) is given in [HNo4], [HNo3] or [PJJ<sup>+</sup>07]. Several case studies within different industries were described in the AGEDIS final report [HNo4]. As one of the goals of the AGEDIS project was to create a tool framework, very good tool support exists.

The AGEDIS modelling and test generation approach is relevant for this thesis, since it shows that 'pure' generation from system models should not be the goal within model-based testing. Besides the system specification with AML additional LTS generation and synthesis with test objectives is needed. Different as in our approach, the usage of modelling viewpoints and relations between them was not considered.

### 3.2.2 Generation from several modelling viewpoints

In the last paragraph we have mentioned several approaches which use system models mostly described with one to two diagrams to generate test cases. In order to introduce the work related to the holistic problem from Chapter 1 we identified approaches, which use several diagrams from different modelling viewpoints for test case generation. As diagrams are used in modelling viewpoints, but the later are not explicitly mentioned in the found approaches, we analyze the used diagrams and relate them to its viewpoint.

A detailed characterization and comparison of almost 50 model-based testing approaches has been introduced by Dias Neto et al. in [DNSVT07]. The authors provide a list of all model types found in the analyzed approaches. As mentioned in Section 1.1, the approaches use UML statechart, class, sequence, use case or activity diagrams for test case generation. We have analyzed 15 UML-based approaches found in the survey (qualitative analysis). We have found that three approaches (namely Briand and Labiche [BL02], Hartmann et al. [HIM00], Vieira et al. [VLH<sup>+</sup>06]) use more than two UML diagrams to generate test cases.

Besides the three mentioned approaches, there exist four other, which fall into this category. Deng et al. in [DSWO04] uses class, activity and use case diagrams to generate test cases. Nebut et

al. uses in [NFTJ06] use case, sequence diagrams and further specification of contracts. The usage of state, class and object diagrams seems to be used by many approaches as [UL07, Wei09, OA99]. Some detailed thoughts of how each UML diagram can be used for testing was introduced by Robert Binder in [Bin99].

We have selected the approach from Briand and Labiche [BL02] as a representative, because of the very high citation rate and similar analysis model as the one used in this thesis [SSE09]. The authors show how to use UML for business analysis with diagrams as use case, sequence and collaboration diagrams, class diagrams and Object Constraint Language (OCL) expressions among those artefacts. The underlying methodology is called TOTEM (Testing Object-Oriented using the UML). To generate test cases the following seven steps have to be executed:

1. Check completeness, correctness, consistency of the Analysis model
2. Derive Use Case dependency sequences
3. Derive test requirements from system sequence diagrams
4. Derive test requirements from system class diagram
5. Derive variant sequences
6. Derive requirements for system testing
7. Derive test cases for system testing
8. Derive test oracles and harness

In [BL02] only steps 2,3 and 5 are introduced. Especially the checks performed in step 1 are interesting for the topic of test case completeness. Unfortunately, no further references on this topic in the context of the TOTEM methodology could be found.

Briand and Labiche define test requirements as logical test cases, which are supplemented with detailed detailed information for test execution (steps 7 and 8). The derivation of test requirements uses:

- UML activity diagrams to derive use case sequences
- UML sequence diagrams to derive test execution paths
- UML class diagrams to derive object sets to be tested (addressed as future work)

For each derivation some algorithms and techniques (like depth search, regular expressions, OCL refinements) are provided. The usage of several diagrams is motivated by the fact that analysis model is used as test basis and that this model is described by several interconnected diagrams. The usage of an analysis model is motivated by the fact that in a object-oriented UML development the system level is based on business analysis artefacts. The authors state that there is no well-accepted standard for UML analysis models [BL02, p.2]. The TOTEM methodology defines how an UML analysis model should look like.

Knowing that [BL02] uses several diagrams, the next question is how the model relations are used in this context? The model relations can be found in the meta-model of the TOTEM methodology [BL02, p.30] shown in Figure 28. As highlighted the *Analysis Document* provides the relation between the different diagrams (interaction and class) and further elements as *UCSequentialDependencies* and *Data Dictionary Description*, which are used in the approach.

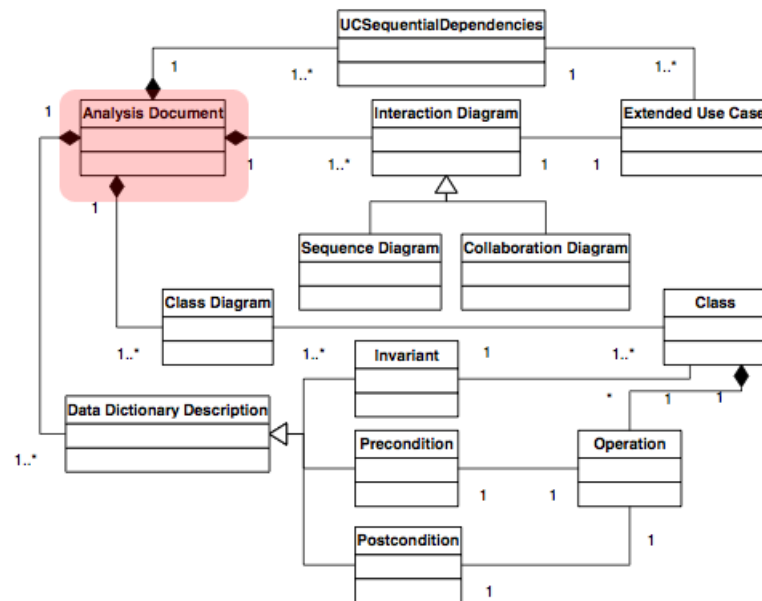


Figure 28: TOTEM meta-model from [BL02, p.30]

Similar to Pickin et al. [PJJ<sup>+</sup>07] they use system models together with some additional test information (here OCL expressions). Therefore no explicit test model is given. As no evaluation study



is provided, there is no information about the quality of the generated test cases. The modelling and generation approach from [BL02] concentrates more on the fault-detection rate and model coverage, rather than on the test case quality attributes. The authors mention some tool support for their approach.

Summarizing, we identified approaches as Briand and Labiche [BL02], which use several diagrams representing the structure and behaviour modelling viewpoint for test generation purposes. The underlying meta-model defines relations between them. Unfortunately the algorithms from steps 2-4 do not use the relations within a context (for example use case scenario and the related parts of the dialog and data model), but try to derive test requirements for the single modelling viewpoints. This observation is relevant for our approach, since we aim to use the relations at the meta-model level in a context relevant for test generation purposes.

*Cognitions for this thesis*

Further observation is that approaches using several diagrams consider the structure and behaviour viewpoint without considering the interaction viewpoint. Only Vieira et al. in [VLH<sup>+</sup>06] includes information about the GUI in their model, but incorporates it in the structure and behaviour modelling viewpoint. This way, no clear interaction viewpoint and its relation to the systems behaviour and structure can be identified.

Finally, approaches using several diagrams analyzed here were good candidates for applying the holistic view during the test generation process. Only two of the three modelling viewpoints are used. The context-based relations between them are not clearly defined by means of a meta-model. The aspect of the internal test quality was mentioned in [BL02], but not further analyzed.

### 3.2.3 Generation from test models

We have already mentioned that system models differ from test models in certain aspects (see [MJV<sup>+</sup>10] or [WL10]). Until now, we have introduced several approaches using system models together with additional information (like the test objectives in [PJJ<sup>+</sup>07] or test-related OCL refinements in [BL02]) to generate test cases. Assuming that test models are created by different roles (test designers rather than business analysts) and contain

test-related information, we have identified several test model based approaches.

One group of the approaches as [VLH<sup>+</sup>06, OA99, MXX06, ISBP07, Wei09, FLo2, GMWE09] uses standard UML diagrams together with additional test information. The additional test information is often specified with the OCL as shown by Bouquet et al. in [BGL<sup>+</sup>07]. Especially for testing GUIs test models like the event-interaction graph in [MSP01] or CNL-specifications in [BM10] are used. The UML Testing Profile is also widely used as a notation for a test model as shown in [DGNP04, Daio4, Daio6, BRDG<sup>+</sup>08, CPT<sup>+</sup>06] and [GSD05]. Some proprietary test models which derivation is based on use case descriptions was referenced in the survey from Denger et al. in [DM03]. The use case models used as test models can be also reengineered by analyzing existing test cases as shown by Hasling et al. in [HGB08]. The last group of approaches uses test patterns as an additional way to model tests with UML. The exemplary publications are [Rum03] from Bernhard Rumpe or Robert Binders book [Bin99].

We have selected the publication "Model-Based System Testing using Visual Contracts" by Güldali et al. as a representative approach. In [GMWE09] the test model consists of so called visual contracts (VC) [Loh06] which enable the visual representation of the Design by Contract idea. For each system use case the pre- and postconditions are specified by two object structures which are modeled with UML object diagrams. The test case generation process is based on classical techniques like equivalence and boundary value analysis. For each specified object of the precondition one or more instances are generated and afterwards linked to the postcondition object. This way a set of test input data together with the expected result can be generated.

In order to execute the test cases, some model transformation rules based on an implementation model are provided. This way the completeness of the generated test cases according to implementation details is guaranteed. The VC are created by test designers and therefore act as a separate test model. Both the system and the test model are described with UML. In this approach no GUI models (interaction modelling viewpoint) are used for test case generation. The relations between models and automated usage of developer models are not considered. The completeness and traceability of test cases are mentioned as important in this approach. The authors mention some tool sup-

port for generating and transforming the logical into concrete test cases.

The observation from the analysis of approaches like [GMWE09] is that test models differ from analysis models by incorporating test-related information. Test models are used by test designers and not business analysts. Further observation is that no automated usage of analysis models for generating test models is used. In the cited approaches the interaction modelling viewpoint is not used within the test modelling. Models like the VC act as test models and through the use of model transformations the internal quality of generated test cases is increased.

*Cognitions for this thesis*

#### 3.2.4 Generation of test models from developer models

Several publications as [PP05, MJV<sup>+</sup>10, WL10] mention that system and test models differ from each other, but also contain similar information. Like shown by Pretschner and Philipps in [PP05] there exist several scenarios how to use system and test models for testing. One possibility is to derive the test model from the system specification. In the case of the model-driven development process, this task can be partially or even fully automated.

The topic of automatic usage of developer models for the creation of test models was investigated in several publications. For example Torres et al. presents in [TECMG09] a survey of approaches using MDA techniques (like model transformations) for testing. Guitierrez et al. in [GECM<sup>+</sup>09] uses model transformation techniques to create a test model containing of test scenarios and operational variables. In the context of the UML Testing Profile several publications deal with the (semi-)automatic derivation of such test models. To mention is the work of Zhen Ru Dai ([Dai04, DGNP04, Dai06]), Baker et al. in [BRDG<sup>+</sup>08], Lamanha et al. [LMdG<sup>+</sup>09] and Chen et al. [CPT<sup>+</sup>06]. As the work of Dai is very representative for the the category described in this subsection, we will describe it now in detail.

The issue of automatically reusing system models for test modelling was investigated by Zhen Ru Dai in her dissertation called "An Approach to Model-Driven Testing" [Dai06]. Dai uses model transformations to transform several UML diagrams into a test model. This model is described with the UML Testing Profile

(UTP) [Obj07b]. The standardization of UTP was strongly influenced by the work of Dai (see [Dai04], [DGNP04] or [BRDG<sup>+</sup>08]). The UTP was already introduced in Section 2.2.

The system model is introduced by an example of a roaming approach for bluetooth devices. Dai uses several UML 2.0 diagrams like: class, package, composite structure, activity, sequence and state diagrams to describe the system. Although this is a model for an embedded system, it can be seen as an analysis model in the early phase of the system development process. On the other side Dai derives a test model for almost all test levels like acceptance, system, integration and also unit testing. In our opinion the analysis model has an abstraction level which can be used only for system testing. This way the system model used for Dai would be more a design model according to the V-Model [Boe79].

For the transformation the Query/View/Transformation (QVT) [Obj08a] is used. The transformation process itself incorporates the so called Test Requirements which define how system model elements should be used for test purposes. For example the choice of the SUT component or configuration of the test system. This way Dai appends test information during the transformation process.

First observation is that transformation rules introduced in [Dai06] consider only the single elements of the system model. The relations between for example the use case and the data model are not investigated. This can be influenced by the fact that the system model has no meta-model other than the one of the UML. For example in Salger et al. [SSE09] the meta-model enables the clear definition of the model relations used by Business Analysts.

The usage of model transformations for automatic usage of developer models is very interesting to solve the problem from Section 1.1. However, during test design the important task is the selection of test cases from models. The examples of the behaviour modelling viewpoint introduced in [Dai06] as in [DGNP04] or [Dai04] are mostly sequence diagrams with sequential messages being send among lifelines. This way, the transformations use those sequence diagrams as test cases in the UTP model and no test selection is performed. In the book from Baker et al. [BRDG<sup>+</sup>08], which deals with the UTP and the transformation done by Dai, also other diagram types like activity diagrams

with decision nodes are considered. But also there no clear test selection method is presented. The point is that analysis models for Business Information Systems consist of behaviour models with different scenarios and the test selection is strongly needed.

Approaches similar to Dai et al. fulfills several of our evaluation criteria. They use several UML diagrams for test generation and test modelling purposes. Test models described with the UTP are automatically created by reusing system models with model transformations. The approaches are focused on the system testing level. Still, no model relations or the interaction modelling viewpoint were used. All publications provide examples of industry case studies together with tool support. Unfortunately, the evaluation criteria regarding internal test quality were not met here. Therefore the aspect of a holistic view on several modelling viewpoints is missing in such approaches.

### 3.2.5 Generation using model relations

While in this phd thesis we investigate the usage of a holistic view on several related models, it is important to identify approaches already using this modelling feature. Back in 1999 Robert Binder already investigated the role and different types of model relations in [Bin99]. Especially the relation between behavioural models like use cases and structure models like class diagrams was investigated by approaches as Kösters et al. [KSW01]. Based on concept of activity graphs from [Win99], the early version of the UML model was refined with the appropriate coupling between use cases and class diagrams. This way, the the quality of the generated tests and the requirements specification could be supported through validation and verification measures.

The importance of model relations has motivated the work of Deng et al. in [DSWO04]. The authors propose a so called Semantic Software Development Model (SSDM) which defines single models for requirements, design, implementation, testing and maintenance. The single models of SSDM are described with different UML diagrams like class, use case, sequence, activity, statechart, collaboration diagrams. The authors define several relationships between models at different levels. For example relations between class diagrams or use case diagrams and a set

of dynamic UML diagrams: "A use case is illustrated by several dynamic UML diagrams" or "The objects in a Sequence/Collaboration diagram and the objects described by a statechart diagram are the instances of the classes in the corresponding Class diagram" [DSWO04, p. 3]. The SSDM is used to generate test cases.

While describing the approach from Briand and Labiche (see Subsection 3.2.2), we have mentioned the different kind of relations used. Other examples of approaches using model relations are [HVFR05, HIM00, VLH<sup>+</sup>06]. The last one can be seen as a representative approach for this category and will be described in detail.

An interesting approach for generating test cases from UML models using model relations was introduced by Vieira et al. in "Automation of GUI Testing Using a Model-driven Approach" [VLH<sup>+</sup>06]. The authors use UML use case, activity (behaviour viewpoint) and class diagrams (structure viewpoint) as a test model for test case generation. The use case diagram is always refined by an activity diagram, which is extended by notes. The notes or annotations as the authors call them, are used to link each activity with the elements of the class diagram being used. The task of annotating the model as well as creating the models is done manually. In the extended version of this paper by Hartmann et al. [HVFR05] the authors mention a semiautomatic usage of the mentioned UML diagrams which are created by the development team.

The generation of test cases is based on a control-flow (all-path-criterion) together with data-flow (category-partition method) test selection. This way a very high number of test cases can be generated. This is also a current problem of this approach, because in [VLH<sup>+</sup>06] for a simple example of six sequentially executed use cases, over 41000 test cases are generated. This high test case number is not affordable to use in practice as the authors state.

Vieira et al. shows how to use UML models for test generation. The motivation for the usage of those three UML diagrams is that those are widely accepted and as describing workflows are well suited for creating automate test drivers. As the extended approach from Hartmann et al. [HVFR05] semiautomatically uses the developer models, the assumption is that those

UML diagrams are also used by business analysts. The idea of annotating the activity diagram by test designers sounds interesting, but model links created by developers are not mentioned. Also the topic of test case quality reached by the approach is not mentioned. The authors concentrate on the model coverage reached by their approach. Both publications [VLH<sup>+</sup>06] and [HVFR05] provide small case studies and tool support for their approach.

The aspect of model relations created manually by test designers is interesting for this thesis. In our motivation we mentioned the different modelling viewpoints and relations between them. Different as in [VLH<sup>+</sup>06] we aim to use linkage between models (instance of model relations at the meta-model level) created already by business analysts. Vieira et al. also does not analyze the different kinds of relations between models and does not provide a clear meta-model as we do.

### 3.2.6 Generation from GUI models

An important group of approaches relevant for this phd thesis are the GUI testing approaches. A recent overview of such approaches has been introduced by Memon and Nguyen in [MN10]. First, it is important to say that all GUI-based approaches use separate test models for test generation. Some early work on this field used finite state machines for generating GUI test cases as [SS97]. In the last subsection, we introduced the approach from Vieira et al. [VLH<sup>+</sup>06], where a separate class diagram described the GUI structure and was used in the annotations of the activity diagram. Another possibility is to use a textual specification language like the CNL in [BM10], which includes terms related to the GUI.

Most of the approaches as [MSP01, Xie06, XMo8, QJ09] use models of the GUI events. The generated test cases are sequences of events to be executed. Special kind of models, which include the positive and negative aspects of GUI's behaviour as the undesirable malfunctions is applied in Belli et al. [BBH05, BBW06]. The authors use the term holistic view in this context as referring to the complementary modelling of undesirable behaviour of the GUI.



As the work of Memon and Xie had a strong influence on this type of testing, we select their publication [XM08] as the representative one and describe it in detail.

In "Using a pilot study to derive a GUI model for automated testing" [XM08] Xie and Memon introduce a technique, which uses an event-interaction-graph (EIG) to generate test cases. The authors mention the typical problems on this field like the enormous input event interaction space, maximization of fault detection while targeting a subspace and high test case length. Those problems were not solved by current approaches on this field or non clear empirical evidence has been shown. That is why the authors searched for new techniques dealing with the mentioned problems.

In the previous work of Memon and Xie (see [MSP01] or [Xie06]) the authors defined several hypothesis about the typical defect types within GUI testing and how to identify them. For example only few events of a GUI in a certain context (e.g. typing a very long text into a dialog field) lead to a crash. Based on this intuitive ideas, the authors defined new test coverage criteria like the minimal effective event-context (MMEC). The MMEC is defined as the shortest sequence of preceding events needed to detect the fault. The validation of this test coverage criteria and the EIG as a test model has been conducted by performing several case studies. Namely they used the universities open-source office suite and four open-source projects.

The chosen approach is based on the computation of the MMEC. The MMEC is computed while executing test cases which are generated from an EIG by using the coverage criteria introduced in [MSP01]. The SUT is seeded with faults using the mutation technology. Each time a mutant is killed (by finding a defect) the MMEC is calculated. While the MMEC searches for the shortest sequence of events in a test case, the length can be reduced. The authors conducted this process (creating an EIG, seeding faults, executing test cases and computing the MMEC) on the five mentioned open-source projects. They compared the found defects with a bug tracking database and identified four regular expressions to describe the MMEC in each case. This way several bugs found in the previous versions and new bugs in the mentioned open-source software could be find.



Our representative approach for generating test cases from GUI models like the other mentioned approaches does not use UML. Since in this thesis a consistent UML modelling approach is required, we have identified the need for UML-based GUI modelling. The notation of an EIG is similar to the UML state diagrams and describes only the behaviour part of the GUI. We could not identify approaches which automatically uses system models to derive a GUI test model. In one publication from Memon et al. [MBN03] the EFG was reverse engineered from the existing code. The test cases generated from the EIG were clearly traceable to the test model and could be analyzed after the defect detection during the test execution. The authors provide several case studies and good tool support as mentioned in [XMo8].

*Cognitions for this thesis*

### 3.2.7 Test case quality attributes

Out of all approaches analysed during the literature evaluation, only very few consider the topic of test case quality attributes like understandability, analysability, completeness and traceability. Compared with that almost all publications concentrate on the defect-detection rate or reached test coverage in terms of code coverage. Looking at Figure 26, we could identify the importance of traceability in several approaches as [MSP01, BRDG<sup>+</sup>08, BM10, FSo5, Wei09, PPW<sup>+</sup>05, LMdG<sup>+</sup>09, HGB08] and [XMo8]. The analysability issue of the generated test cases was considered by Xie and Memon in [XMo8] while computing the MMEC after the test execution. The completeness of test cases with respect to implementation details was only considered in Güldali et al. [GMWE09]. Finally, the understandability of generated test cases was only partially considered by Offutt and Abdurazik in [OA99]. Since the internal quality of generated test cases is important as explained earlier in this chapter and stated by [ZVS<sup>+</sup>07] and [PWGW08], we identify a need for further research on this topic in this thesis.

*Cognitions for this thesis*

The only publication identified in our survey, which provides empirical evidence for analysing the internal test case quality is the one from Zeiss et al. in [ZVS<sup>+</sup>07]. The authors introduce a quality model for test specifications, which was derived from the ISO 9126 software quality standard [ISO04]. For a subset of the

quality attributes from the derived quality model some metrics were defined using the Goal Question Metric approach [BCR94]. For example, the analysability quality attribute was measured by the complexity violation metric. The complexity could be measured for example with the McCabe's cyclomatic number [McC76].

The subset of 10 quality attributes was applied to four test specifications described with the standardized TTCN-3 specification language [TTC05]. For each quality attribute one or more metrics were applied. By defining boundaries for each metric, violations and possible quality improvements of the test specifications have been shown. Unfortunately only one (namely analysability) of the four quality attributes relevant for this thesis was considered in this evaluation. Further the specification of tests with the UML rather than TTCN-3 is of interest in our context.

#### Excursion: Term "holistic view" in model-based testing

In this thesis, we use the term "holistic view" and "holistic model-based system testing" to describe the research problem of applying a special view on several intra-related modelling viewpoints in a certain model-based testing scenario (using analysis models to generate test model). In fact, the term "holistic view" is not new and has been introduced by Fevzi Belli in a conference paper from 2001 [Bel01] and a technical report called "A Holistic View for Finite-State Modeling and Testing of User Interactions" back in 2003 [Bel03]. The author uses this term to describe a test modelling and generation approach considering the positive (desired) and negative (undesired malfunctions) aspects of the SUT. This way, the approach presented in this thesis and the one introduced by Fevzi Belli both concern modelling aspects. The difference is that Belli concentrates only on modelling the GUI (interaction modelling viewpoint) of test model, rather than the whole model landscape of analysis models.

### 3.3 SUMMARY

Until now we have introduced several groups of approaches and compared them according to predefined evaluation criteria in Figure 26. For each group we have chosen a representative approach and described it in detail. There are some general results of the literature evaluation, which can be summarized as follows:

- Few approaches use several intra-related modelling viewpoints for test generation (missing holistic view)
- The interaction modelling viewpoint is not considered as an integral part of the analysis model
- GUI models are mostly separated from analysis models and created exclusively for test purposes
- Model relations are not described within underlying meta-models and their usage in a context is missing
- Test models with additional and independent test information are needed
- Automated use of analysis models is possible through model transformations
- Internal test quality was considered by very few approaches

Our literature survey provides deeper insight into the problems introduced in Chapter 1. We investigated several approaches according to evaluation criteria (see Section 3.1) derived from the problem statement of this thesis. By showing that only few approaches use several diagrams and very few approaches deal with the interaction viewpoint integrated into the analysis model, we argue for the missing holistic view problem. Further the missing usage of model relations is related to the same problem. Influenced by the work of Dai et al. [DGNP04], we identify a strong need for automatic usage of developer models and a possible solution through the usage of model transformations.

A very important conclusion is that the quality of test cases with respect to understandability, analysability and completeness was investigated only by four approaches. The missing holistic view and conceptual and empirical evidence for its influence on the internal test quality leads to a new research opportunity in the field of model-based testing.



## Part II

### APPROACH AND EVALUATION



---

## META-MODEL ALGEBRA

---

The solution developed for the research problems of this thesis is based on several meta-models (analysis and test meta-model). We discovered the need for a meta language, which would allow us to describe the properties of meta-models and the behaviour of methods of which the solution consists. This language should support the method engineer (in the case of this phd thesis its author) to define operations performed on meta-models in a systematic and understandable way. Like in mathematics, the algebra allows to define operators between sorts. Our language called meta-model algebra, allows us to define meta-model operations like test coverage measurement between meta-models.

This chapter provides a brief introduction into the meta-model algebra. We start with a short motivation and basic definitions. Then, we introduce the main concepts, namely the algebra operations and the related meta-model properties. Afterwards, we introduce the specification language and the instantiation concept of the algebra. At the end, we discuss its applicability and summarize the chapter.

### CONTENTS

4.1	Motivation . . . . .	106
4.2	Definitions . . . . .	108
4.3	Algebra Meta-Model . . . . .	109
4.4	Related work . . . . .	110
4.5	Meta-Model Properties . . . . .	112
4.6	Algebra Operations . . . . .	116
4.7	Algebra Specification Language . . . . .	120
4.8	Algebra Instantiation . . . . .	122
4.9	Applicability discussion . . . . .	123
4.10	Summary . . . . .	124

---

## 4.1 MOTIVATION

*Need for a  
meta-model algebra*

The development of methods for model-driven software engineering uses meta-models to describe the structure of artefacts on which they work on. The according terms known from the literature are method engineering [Bri96] and meta-modelling as for example the OMG Software & Systems Process Engineering Meta-Model (SPEM) [Obj08b]. Another example is the Meta-Object Facility (MOF) [Obj06a], which can be used to define languages. In all mentioned domains the meta-modelling of structural aspects (for example definition of meta-models according to the MOF-based languages) rather than behavioural (for example transformations of meta-models) aspects are focused. Additional techniques as constraints modelling (for example OCL [Obj06b]) or declarative model transformations (for example QVT [Obj08a]) could be used to specify the behaviour, but they miss the specification of the required meta-model properties as traversability or structural mapping.

*General idea*

The general idea of the meta-model algebra is to support the specification of the behaviour of operations on meta-models and the required properties of meta-models within method engineering. In Figure 29, we depicted the idea by using views on the structure, behaviour and roles in method engineering. The main artefacts are meta-models, which describe the structure of the data the method operates on. First, the behaviour of the method is described by the algebra operations (which represent the *signature* of a method). Second, algorithms (which represent the *semantic* of a method) refine the algebra operation. Both artefacts are created by a method engineer and are based on meta-models. Finally, the algorithms are implemented into tools and executed by software engineers in a project on meta-model instances.

*Thesis context*

In the case of this thesis, we use method engineering to develop a holistic model-based testing approach, which consists of several meta-model operations. Each operation is based on the analysis and / or test meta-model (see Chapter 2). The method engineer from Figure 29 is the thesis' author. The implementation of algorithms is part of the prototype used in the evaluation chapter. The execution of our approach can be done by software engineers in model-driven development projects.



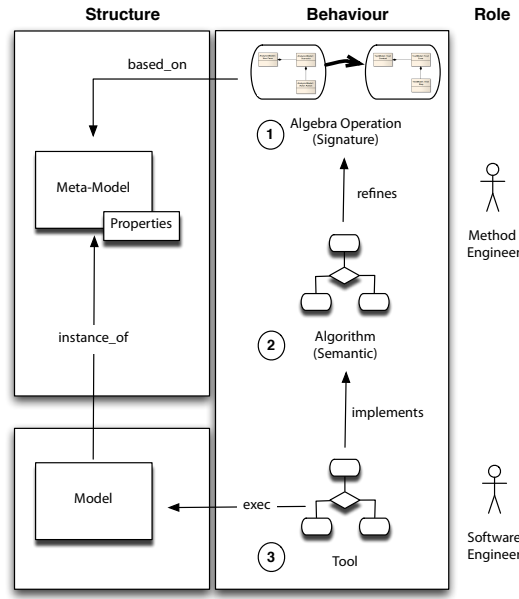


Figure 29: General idea of the meta-modelling with the meta-model algebra

To motivate the usage of algebras for meta-models, we provide an exemplary definition of an algebra  $A$ :

$$A = (M, \quad (4.1)$$

$$\text{select} : M \rightarrow M, \quad (4.2)$$

$$\text{transform} : M \rightarrow M, \quad (4.3)$$

$$\text{cover} : M, M \rightarrow M) \quad (4.4)$$

Exemplary terms specified with algebra  $A$ :

$$\text{select}(m) \quad (4.5)$$

$$\text{transform}(\text{select}(m)) \quad (4.6)$$

In the example provided above, we define an algebra  $A$ . This algebra is defined over the universe of all meta-models  $M$ . The algebra consists of two unary operators *select* (selection of test cases), *transform* (model transformation). This way, the *signature* of meta-model operators is specified (see step 1 in Figure 29). Since the mentioned operations cannot work on any type of meta-model (for example to select test cases the traversability

of meta-models is needed), the meta-models  $M$  have to fulfill certain properties. Then, terms like 4.5 (selection of logical test cases based on the analysis meta-model structure) or 4.6 (transformation of the test selection results) with the variable  $m \in M$  can be constructed. But, to develop a holistic method in this thesis, also the *semantic* of the operations has to be defined. This is done by defining algorithms. The specification of algebra operations with meta-model properties together with algorithms is the goal of the meta-model algebra.

*Customization* Since the definition of an complete algebra, which includes all possible operations on meta-models would require a separate phd thesis, we concentrate on a set of operations relevant for the holistic model-based testing. We apply the set of operations to concrete meta-models (see analysis and test meta-models from Chapter 2) in the next chapter. In the context of model-driven development, further possible operations could be identified. Since the meta-model algebra is not the main part of this thesis, we will give an outlook of possible extensions as future work.

## 4.2 DEFINITIONS

We provide the following definitions for the purpose of this thesis:

**META-MODEL ALGEBRA** consists of one or more operations and their related properties, which are performed on single or multiple meta-models.

**ALGEBRA OPERATION** specifies an operation, which is performed on elements of one or several meta-models. Each operation has a signature, which consists of an input and an output set of meta-model elements. Both sets can consist of elements of the same or different meta-models. The concrete specification of an algebra operation is done by a method engineer within an algorithm. We assume that algorithms are implemented in a tool and executed automatically.

**META-MODEL PROPERTY** defines an abstract property, which is fulfilled between elements of single or multiple meta-models. Properties are defined over an abstract and not concrete meta-model (for example analysis or test meta-model).

An algebra operation is always based on one or several meta-model properties and can be only applied on meta-models for which the property holds.

**META-MODEL ALGEBRA INSTANTIATION** takes place together with the instantiation of the meta-models used in the meta-model algebra operations. The operations are refined as concrete algorithms and implemented (instantiated) for automated execution.

### 4.3 ALGEBRA META-MODEL

In Figure 30, the meta-model of the algebra is shown. The algebra consists of several operations which can be performed on meta-models. The algebra operations and the properties are the main concepts of the meta-model algebra. Each operation has one property on which it is based on. Further, each operation uses one or several meta-models to define the input and output sets. A property is described over an abstract meta-model (highlighted in Figure 30), since it defines an abstract property which can hold on any concrete meta-model.

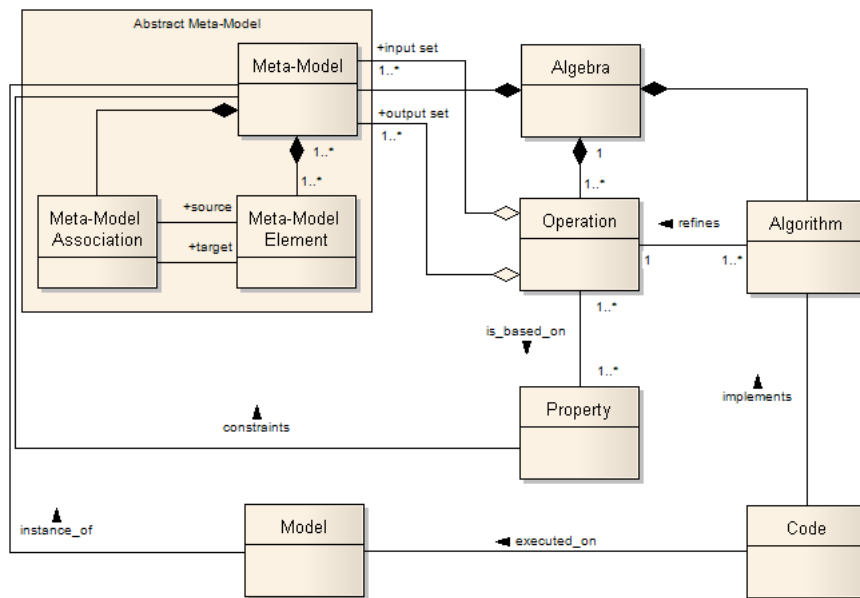


Figure 30: Meta-Model of the Meta-Model Algebra

*Algebra operation  
vs. algorithm*

Each algebra operation is refined by one or more algorithms. Let us consider the distinction between algebra operations and algorithms in the meta-model from Figure 30. The algebra operations represent the high-level view (what / signature), while the algorithms represent the concretization of each algebra operation (how / semantic). We have chosen this distinction because of the fact that there can be several concrete algorithms, which can be mapped to one algebra operation. Each algorithm can use a different technique (for example sequential, parallel or recursive algorithms) to produce the output set based on the input set of meta-model elements. This distinction can be also compared to programming languages: the algebra operations are the signatures and algorithms the body of methods.

Together with the instantiation of the meta-models, the implementation of the algorithms into source code takes place. The implemented code is executed on models, which are based on meta-models used as the input and output set of algebra operations. More detailed description of the instantiation is provided in Section 4.8.

We have chosen this meta-model structure because of the simplicity and usability of the algebra for the method engineer. Besides the operation and property elements, further ones formalizing the semantics by detailed constraints of each operation or property would be possible. For the purpose of this thesis a language, which supports the specification of the syntax and rigorous semantics of operations is relevant. The specification of more precise formal semantics of operations (for example with algebraic specification [EM90]) is not part of this thesis.

## 4.4 RELATED WORK

In the literature the topic of algebra is known from the mathematics. Algebra is the branch of mathematics concerning the study of the rules of operations and relations, and the constructions and concepts arising from them, including terms, polynomials, equations and algebraic structures [BS81]. There are different types of algebras. For the purpose of this thesis, the so called elementary algebra, which consists of variables, operations and equations is relevant. In its simplest meaning in mathematics

and logic, an operation is an action or procedure, which produces a new value from one or more input values. There are two common types of operations: unary and binary. The meta-model algebra operations used in this thesis are both unary (single meta-model operations) and binary (multiple meta-model operations).

In the field of computer science the domain of algebraic specification is a formal process of refining specifications to systematically develop more efficient programs. Ehrig and Mahr introduced in [EM90] the concept of signature algebras, which are defined over a set of sorts (data structure) and operations. Semantics in this context is defined by equations over the mentioned sorts by using operations. This fundamental concept is applied within the meta-model algebra by replacing sorts with meta-model definitions and operations with algebra operations.

In the context of method engineering the general concept of specifying methods for software engineering was shown by Engels and Sauer in [ES10]. The authors provide a good overview to method engineering, meta-modelling and language engineering. The goal of the approach presented in [ES10] is to provide a meta-method for systematically developing methods for software engineering. To support the method engineer the authors introduce a fundamental process of the meta-method, which consists of six steps. The result of this process is a software engineering method. The meta-method integrates the structural (for example the meta-model structure) and behavioural (for example transformations on the meta-model) meta-modelling aspects. The behavioural aspects of meta-modelling presented in [ES10] influenced the definition of the algebra operations.

Good examples of meta-modelling approaches for models of software development methods are SPEM [Obj08b] and the ISO/IEC 24744 Software Engineering - Metamodel for Development Methodologies [ISO07]. Both examples introduce the structural aspects of meta-modelling in form of meta-models. Unfortunately, none of them supports the behavioural aspects which are relevant in this thesis.

## 4.5 META-MODEL PROPERTIES

Before we introduce the concept of algebra operations, we need to introduce the meta-model properties, since each algebra operation is based on one property. Properties of meta-models are constraints, which have to be fulfilled for one or several meta-models. For example the existence of model relations between meta-model elements is a property. Meta-models without the possibility to model relations can not be used to define algebra operations based on the mentioned property. We define the following meta-model properties for the purpose of this thesis:

- traceability
- modelling viewpoints
- model relation
- structural mapping
- traversability

*Identification of  
properties*

We have chosen the five meta-model properties because of their relevance for the development of a holistic model-based testing approach. The *modelling viewpoints* and *model relation* property is the essential property needed to use a holistic view on all three viewpoints of the analysis model (see Chapter 1). To use analysis models for test purposes (see Section 1.1), the *structural mapping* property is needed. The *traceability* property is needed to measure the holistic model coverage (see Section 1.1). Finally, the *traversability* property is needed to perform the test selection, which is essential in model-based testing (see Subsection 2.2.4). Besides the mentioned properties, further ones can be defined and used within the meta-model algebra.

In the following we briefly describe each meta-model property.

### 4.5.1 Traceability

Traceability was already defined in Subsection 2.5.3 as the ability to trace the connection between the artefacts of the testing life cycle or software life cycle [UL07]. In the context of model-driven development we redefine it as the ability to trace the connection between elements of models based on their meta-model structure. For example: a test case (element of the test meta-model) is

traceable to a use case (element of the analysis meta-model). The identification of the trace between the mentioned meta-model elements can be done at the model instance level by executing algorithms (refinements of algebra operations).

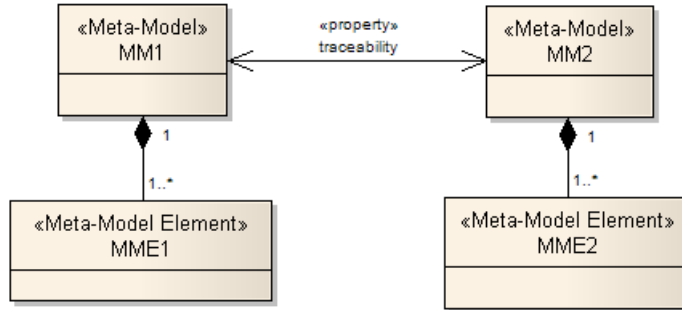


Figure 31: Meta-model property *traceability*

In Figure 31 we have depicted the traceability property on the meta-model level. The element MME2 (part of the meta-model M2) is traceable to the element MME1 (part of the meta-model M1). The traceability property is bidirectional, since MME2 can be traced back to MME1 and MME1 can be traced forward to MME2. The bidirectional traceability can be used in model-driven development for impact analysis and especially in model-based software testing for model coverage measurement.

#### 4.5.2 Modelling viewpoints

The essential contribution of this phd thesis is the usage of different modelling viewpoints for model-based system testing. Based on an exemplary analysis modelling approach from Salger et al. [SSE09], we identified viewpoints as structure, behaviour and interaction. This viewpoints are defined by the modelling approach itself and supported by the chosen modelling language (here UML).

To define operations on meta-models, which use the holistic view the property of existing *modelling viewpoints* has to be fulfilled. It means that the modelling approach and the modelling language used to create the meta-model have to support the concept of viewpoints.

### 4.5.3 Model relation

The relations between elements of the analysis model has a direct impact on the quality of the holistic model-based testing approach (see Section 1.1). Model relations are syntactical associations between elements of a meta-model. For example the *Association* concept of the UML meta-model [Obj09, p.39] fulfills this meta-model property. The relations are defined for single elements of the meta-model, which can be part of different modelling viewpoints. This kind of model relations enables the usage of the holistic view in our approach.

The topic of the semantic of model relations has been covered by Jan Hausmann in [Hau05]. Within algebra operations, we concentrate on the syntax and not semantic of model relations.

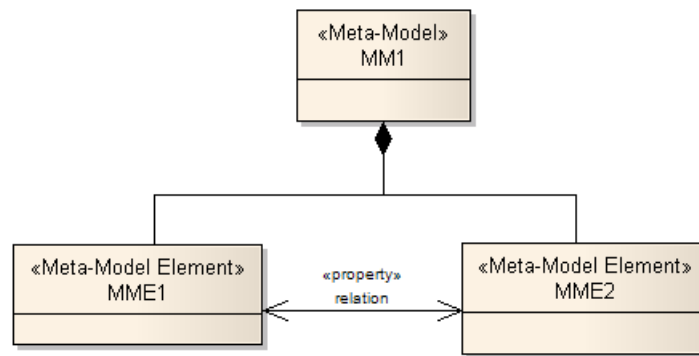


Figure 32: Meta-model property *model relation*

In Figure 32 the model relation relation property is visible between MME1 and MME2, which both are parts of the meta-model M1. Similar to the *traceability* property, also model relations are bidirectional.

### 4.5.4 Structural mapping

The intended use of analysis models for test purposes is based on the idea of a structural mapping between the analysis and test model (see Section 1.1). The meta-model property *structural mapping* is defined as the structural mapping between elements of two or more meta-models. The mapping takes place when an algorithm (refinement of an algebra operation) is executed. This



property is visualized in Figure 33. Information contained in the element MME1 is mapped to the element MME2. This property is not bidirectional, since we assume that MME2 was created by using the information of MME1 and not otherwise.

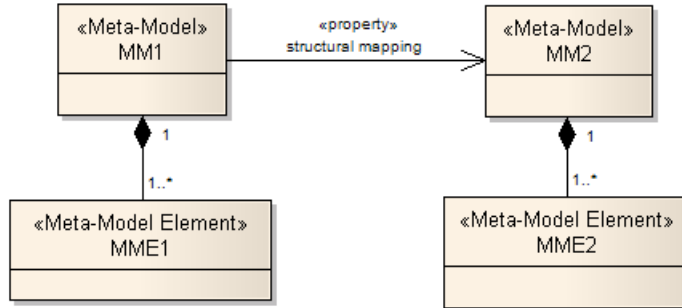


Figure 33: Meta-model property *structural mapping*

#### 4.5.5 Traversability

The last meta-model property is the traversability of behavioural parts of a meta-model. The topic of graph traversal for test selection was introduced in Subsection 2.2.4. Based on this introduction, we define traversability as the property of traversing meta-model elements by using its transition system. For example traversing meta-models defined in a graph-like structure (for example sequences of actions within use case scenarios from the analysis meta-model in Subsection 2.4.11) to select traces (also called paths) with sophisticated algorithms. In this case the transition system allows to navigate tokens from the initial to the final action of the use case scenario. To allow only valid transitions the operational semantics of the transition system has to be defined.

The problem of infinite loops in such a transition system is handled within the algorithms as the refinement of algebra operations, which are based on the *traversability* property. To execute operations based on this property the assumption of meta-models, which can be traversed from the initial to the final node has to be fulfilled.

## 4.6 ALGEBRA OPERATIONS

The purpose of the algebra operations is to specify operations performed on meta-models. The algebra operations specify the purpose and data structure needed to operate (signature), but not the concrete steps (semantics) for its execution. The concretization of an algebra operation is an algorithm specified by a method engineer. Let us consider the visualization from Figure 34. In a model-driven development process two meta-models MM1 and MM2 are used to support activities like business analysis, design, code generation, testing, etc. Both meta-models consist of several meta-model elements. An operation can be performed on a single meta-model (see OP1 in Figure 34) or several meta-models (see OP2 in Figure 34). The specification of an operation uses meta-model elements as input and output sets. The input set for OP1 and OP2 are the elements of MM1. In the case of OP2 the elements of MM2 are the output set. The output set of operation OP1 consists of elements of MM1.

*Different operation types*

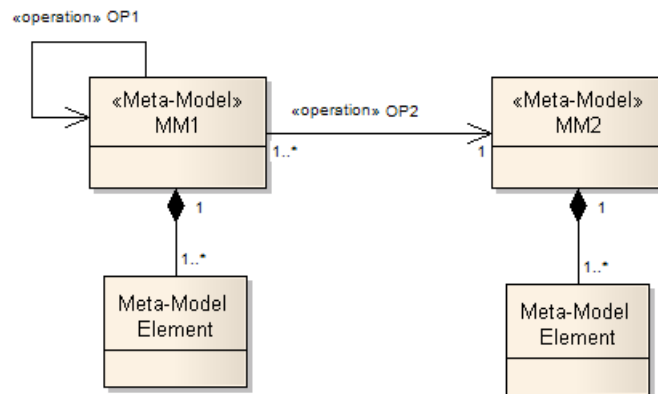


Figure 34: Meta-model algebra operation

Based on the description provided above, we define the following types of algebra operations:

**SINGLE META-MODEL OPERATION** is an operation which input and output sets are based on the same meta-model.

**MULTIPLE META-MODEL OPERATION** is an operation which input and output sets are based on different meta-models.

The distinction between single and multiple model operation types was chosen because of the operations identified in the

literature. Good examples of single meta-model operations are test selection algorithms (an overview was introduced in Subsection 2.2.4), which are based on meta-models used to describe the systems behaviour. A widely known example for multiple meta-model operations are M2M transformations (an overview was introduced in Section 2.5).

For the purpose of this thesis, we concentrate on single and multiple meta-model operations, which use a maximum of two different meta-models for the specification of input and/or output sets. This restriction was chosen because none of the operations defined in the model-based testing approach from Chapter 5 uses more than two different meta-models. The specification of algebra operations based on more than two meta-models is possible and can be evaluated in the future work.

To define an operation of the meta-model algebra the following requirements have to be fulfilled:

- REQ1 The structure of each meta-model has to be defined
- REQ2 The input and output element sets based on their meta-models have to be defined for each operation
- REQ3 Each operation has to be based on a meta-model property (see Section 4.5)
- REQ4 Meta-models used in the operation have to fulfill the meta-model property

Based on the introduction provided so far, we define the following attributes of an algebra operation:

- Name (one word name of the algebra operation)
- Goal (short description of the operations purpose)
- Type (type of the algebra operation)
- Input meta-model
- Input set (list of meta-model elements based on one or several meta-models serving as the input of the operation)
- Output meta-model
- Output set (list of meta-model elements based on one or several meta-models serving as the output of the operation)
- Property (name of the abstract meta-model property which is used by the algebra operation)

Based on this template the algebra operation can be formalized as follows:

$$\text{operation} : M_i \rightarrow M_o \quad (4.7)$$

where:

- operation stands for the algebra operation
- $M_i$  stands for the input meta-model(s)
- $M_o$  stands for the output meta-model(s)

For the purpose of this thesis, we define the following operations which can be specified for one or more meta-models and performed on their instances:

- transform (model transformation)
- select (test selection)
- extract (model information extraction)
- cover (model coverage)

In the following we will define each mentioned algebra operation by using the attributes introduced in this section.

#### 4.6.1 transform

Let us take the *transform* operation and specify it with the mentioned template:

- Name = transform
- Goal = elements of meta-model  $M_1$  are transformed into elements of meta-model  $M_2$
- Type = multiple meta-model operation
- Input meta-model= meta-model  $M_1$
- Input set = elements of meta-model  $M_1$
- Output meta-model = meta-model  $M_2$
- Output set = elements of meta-model  $M_2$
- Property = structural mapping

## 4.6.2 select

Let us take the *select* operation and specify it with the mentioned template:

- Name = select
- Goal = transition systems of behavioural elements of meta-model  $M_1$  are traversed to identify paths
- Type = single meta-model operation
- Input meta-model= meta-model  $M_1$
- Input set = behavioural elements of meta-model  $M_1$
- Output meta-model= meta-model  $M_1$
- Output set = sequence of behavioural elements of meta-model  $M_1$
- Property = traversability

## 4.6.3 extract

Let us take the *extract* operation and specify it with the mentioned template:

- Name = extract
- Goal = elements of meta-model  $M_1$  are analysed to identify context-related information by using relations between its meta-model elements
- Type = single meta-model operation
- Input meta-model= meta-model  $M_1$
- Input set = elements of meta-model  $M_1$
- Output meta-model= meta-model  $M_1$
- Output set = elements of meta-model  $M_1$
- Property = modelling viewpoints, model relation

## 4.6.4 cover

Let us take the *cover* operation and specify it with the mentioned template:

- Name = cover
- Goal = elements of meta-model M2 are analysed for their coverage of elements of meta-model M1 by using traces between M1 and M2
- Type = multiple meta-model operation
- Input meta-model= meta-model M2
- Input set = elements of meta-model M2
- Output meta-model= meta-model M1
- Output set = elements of meta-model M1
- Property = traceability

## 4.7 ALGEBRA SPECIFICATION LANGUAGE

In the last section, we have specified the different algebra operations textually. Since the specification of input and output sets for each operation is mandatory and depends on the meta-models used, also a graphical visualization is needed. In this section, we introduce the visual specification language for algebra operations.

### *Language syntax*

The visual specification language is based on the Unified modelling Language [Obj09] and is similar (in terms of visualization and some properties) to the concept of the Visual Contracts (VC) introduced by Lohmann in [Loh06]. The main idea is to visualize each operation with UML diagrams. The input and output set of each operation are separate typed graphs. This does not mean that both graphs have to be based on different meta-models. The input set graph is transformed to the output set graph by applying the algebra operation. Different as in VC the transformation is applied on meta-models (depicted as class diagrams) and not objects (depicted as object diagrams).

In Figure 35 the structure of the algebra specification language is shown. The core elements of the visualization are the input and output sets depicted as class diagrams. The sets consist of the meta-model elements and their relations relevant for the operation. This way, each set represents a graph typed over one or multiple meta-models. The edge between the mentioned sets symbolizes the algebra operation performed.

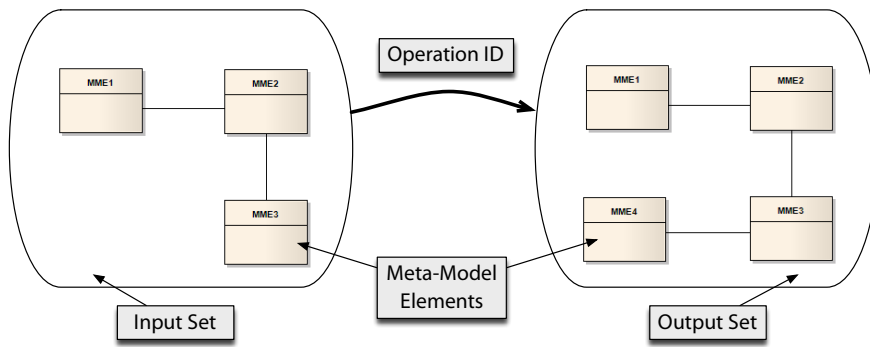


Figure 35: Structure of the algebra specification language

The Figure 36 visualizes the example for the *transform* algebra operation in the algebra specification language. There are three elements (*Use Case*, *Scenario* and *Actor Action*) of the analysis meta-model (Section 2.4.11) in the input set. The output set consists of three elements (*Test Context*, *Test Case* and *Test Step*) of the test meta-model (Section 2.3). The *transform* algebra operation transforms the elements of the analysis model into elements of the test model. The specification of the semantic (concrete steps needed for the execution of this operation) is done within an algorithm.

Example  
visualization

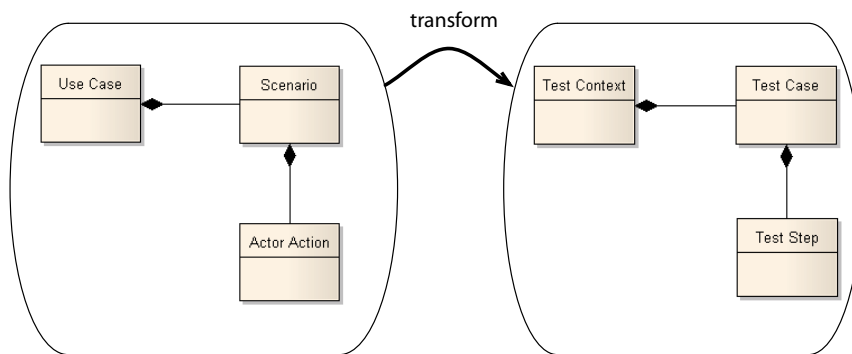


Figure 36: Example for the *transform* operation in the algebra specification language

## 4.8 ALGEBRA INSTANTIATION

In the last sections, we often mentioned the instantiation of the meta-model algebra. The purpose of this section is to describe the instantiation process and its integration in the method engineering process as a part of this phd thesis.

The algebra instantiation is needed to provide a proof-of-concept for each algebra operation and their refinement in form of algorithms. The meta specification of operations supports the method engineer in developing model-driven methods. The automatic execution of algorithms (which are part the method created by the method engineer) by tool implementations supports the software engineers in performing their tasks.

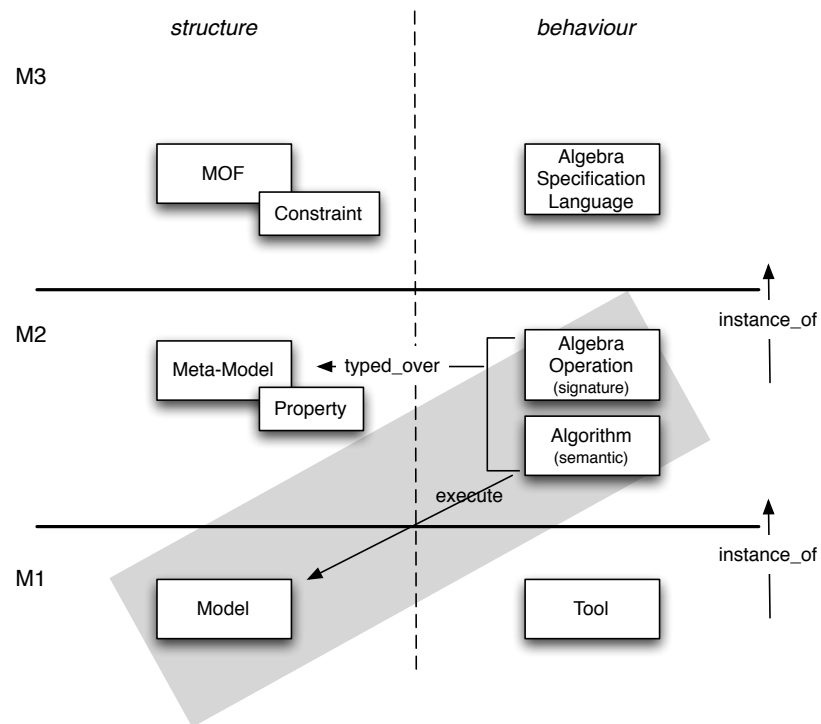


Figure 37: Visualization of the instantiation process from the method engineering perspective

### Instantiation process

In Figure 37 we depicted the instantiation process of the meta-model algebra. We distinguish between the meta-model levels M3, M2, M1 as introduced by MOF [Obj06a] and additionally the structure and behaviour level (depicted horizontally). The



categorization is derived from the method engineering perspective on meta-levels introduced in Engels and Sauer [ES10]. In the so called software engineering perspective the meta-model, algebra operation and algorithm are placed on the  $M_3$  level. This way, the instantiation of methods from method engineering to software engineering results in a shift within the MOF levels.

The method engineer is responsible for the specification of algebra operations according to the algebra specification language presented in this chapter. Based on this specification he refines each algebra operation into an algorithm. Those two artefacts are instantiated by implementing them into tools. The algorithms (implemented in tools) are executed only on models, which are instances of the meta-models fulfilling the meta-model properties on which the algebra operations are based. This consistency is needed, since the algebra operations and their related algorithms are specified for certain meta-model types. Further, the meta-model properties are instances of constraints from the  $M_3$  level. In our approach the constraints on the  $M_2$  level are described textually because no formal semantics is needed.

## 4.9 APPLICABILITY DISCUSSION

The concept of the meta-model algebra tackles the problem of missing behavioural specification in the domain of meta-modelling and method engineering. The algebra is based on the specification of algebra operations (signature over meta-models and related properties) and algorithms (semantics specified with UML activity diagrams). This way, we support method engineers in their task of specifying operations on meta-models.

We have mentioned, that the meta-model algebra is an elementary algebra, which uses unary and binary operations (see Section 4.4). Further properties of algebras known from mathematics as commutative, associative, inverse operations or order of operations were not analyzed in this chapter. The specification of algebra operations used to develop the holistic model-based system testing approach does not require the analysis of such properties. Future work on the meta-model algebra can investigate further algebra operations with respect to the mentioned properties.

As mentioned at the beginning of this chapter, for the specification of behavioural aspects in meta-modelling techniques like model transformations or constraints modelling can not be used. This problem could be solved by extending model transformation languages as QVT [Obj08a] with general assertions, which can be checked on any meta-model type. For this, techniques known from the area of model checking could be used. There exist no straightforward solution for the modelling of properties in constraint modelling languages as OCL [Obj06b]. Since OCL can be only applied on the M2 level, general constraints for meta-models cannot be specified. Further research on both topics can lead to more precise language for the meta-model algebra in the future.

#### 4.10 SUMMARY

Until now we have introduced the high-level concept of the meta-model algebra. By using a visual specification language, we are able to specify operations on meta-models, which are based on certain abstract meta-model properties. We will use the meta-model algebra in the next chapter to introduce the holistic model-based testing approach, which aims to solve the problems mentioned in the introduction of this thesis.

---

## MODEL-BASED TEST SPECIFICATION PROCESS

---

Holistic view in model-based testing is needed! This is what we have shown in the last chapters. Now, we want to introduce the research approach developed to solve the problems of missing holistic view, while using analysis models for test generation and holistic coverage measurement. We call it the *model-based test specification process*.

In this chapter, we first summarize the requirements for a holistic model-based testing approach. Then, we will describe the process and its artefacts at a general level. Afterwards, we will provide detailed descriptions for all process steps.

### CONTENTS

5.1	Requirements . . . . .	125
5.2	Approach overview . . . . .	126
5.3	Step 1. Analyze and annotate the Analysis Model . . . .	130
5.4	Step 2. Generate Basic Test Model . . . . .	138
5.5	Step 3. Extend the Basic Test Model . . . . .	179
5.6	Step 4. Generate Concrete Test Cases . . . . .	186
5.7	Summary . . . . .	194

---

### 5.1 REQUIREMENTS

During the development of the model-based test specification process, the following requirements were important as their fulfillment is strongly related with the presented research problems and the contribution of this phd thesis:

- REQ<sub>1</sub> UML notation should be used for the analysis and test model
- REQ<sub>2</sub> The analysis model should be automatically used to create a test model
- REQ<sub>3</sub> Model relations within the analysis model should be used
- REQ<sub>4</sub> Model elements of the structure, behaviour and interaction modelling viewpoints within the analysis model should be used
- REQ<sub>5</sub> The coverage of several modelling viewpoints of the analysis model by the test model has to be automatically measured
- REQ<sub>6</sub> The test model should be extended with independent test information
- REQ<sub>7</sub> Concrete test cases should be automatically generated from the test model
- REQ<sub>8</sub> The approach should guarantee high understandability, analysability, completeness and traceability of test cases

The requirements listed above were defined according to the contribution points from Section 1.2 and the evaluation criteria from Section 3.1. In Figure 38 we have mapped the requirements to the contribution figure from Section 1.2.

## 5.2 APPROACH OVERVIEW

In this section, we first briefly introduce the process and afterwards the artefacts used during the process execution.

### 5.2.1 Process

The model-based test specification process consists of the following four steps:

- 1. ANALYZE AND ANNOTATE THE TEST BASIS** In this step, the analysis model regarded as the test basis is manually analyzed by test designers for testability deficiencies. Additionally, test designers prioritize use cases by annotating the relevant parts of the models.



Subsection 2.2.3 and Section 2.3), we automatically select them. The holistic view is applied by navigating through model relations and collecting information from the interaction and structure modelling viewpoint. The holistic model coverage measurement is performed with a dedicated algorithm. All algorithms operate on meta-models and are specified with the meta-model algebra introduced in the last chapter.

3. **EXTEND BASIC TEST MODEL** In this step, the automatically generated basic test model is manually reviewed and extended with test data and expected results.

**Rationale:** The usage of analysis models has always to incorporate independent test information as stated in Section 1.1. The manual review and extension step guarantees the quality assurance of the generated test model. It also aims to reveal faults in the analysis model.

4. **GENERATE CONCRETE TEST CASES** In this step, concrete test cases are automatically generated from the test model and external test data sources.

**Rationale:** To execute tests, concrete test cases with test data and platform-specific information are needed. In this last process step, the holistic view is applied also on the extended test model to generate concrete test cases. Because of the complexity of manually created test data sets, an external source for test data management is used.

Two of the steps (step 2 and 4) are fully automated and the other two have to be done manually as depicted in Figure 39. As shown in this figure the process is classified in the test phases analysis, design and implementation of the fundamental test process from [SL05]. Since the three mentioned phases were already defined as test specification in Section 2.1.1, we use the name *model-based test specification process*.

*Loops in the process*

As shown in Figure 39, two loops are possible in our process. The first loop between step 3 and 1 is taken when the automatically generated basic test model is incomplete or the reached model coverage is insufficient. In this case the analysis model has to be manually refined. The second loop between step 4 and 3 is taken when the automatically generated concrete test cases are incomplete. In this case the test model has to be further refined.

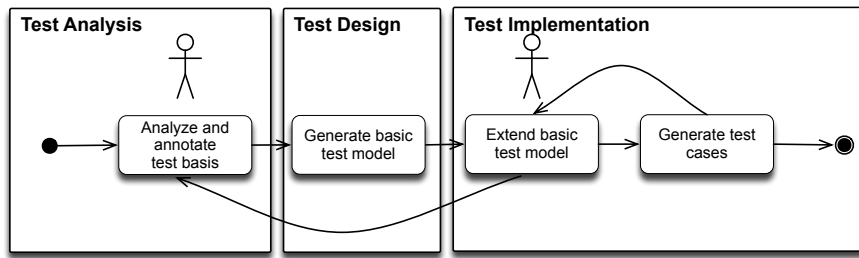


Figure 39: Model-Based Test Specification Process

### 5.2.2 Artefacts

The model-based test specification process uses the following artefacts:

- Step1: Analysis model (with annotated model elements)
- Step2-3: Test model (basic and extended)
- Step2: Trace Model
- Step2: Coverage report
- Step4: Concrete test cases

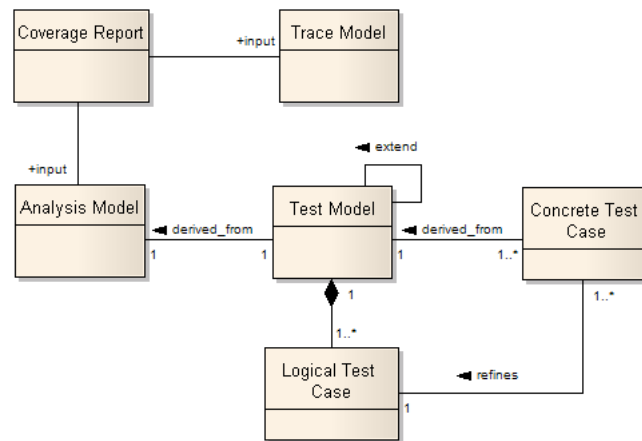
Additionally we have depicted the artefact meta-model in Figure 40. This allows us to show the relations between the mentioned artefacts.

In the first step, the analysis model created by business analysts is extended with annotation information. The structure of the analysis model was described in Section 2.4. The analysis model is taken as the input for step 2.

The result of step 2 is a test model described with the UML testing profile (UTP). UTP was introduced in Subsection 2.3. We differentiate between the *basic test model*, which is the result of the automatic generation process and the *extended test model*, which results from the manual extension process. In the nomenclature used in Roßner et al. [RBGW10], the respective terms *logical test specification model* and *concrete test specification model* are used. Both models have the same structure, but differ in the information level.

*Different  
nomenclature for  
test models*

Additionally to the test model in step 2 a trace model described with the UML is generated. The syntax and semantic of the trace



**Figure 40:** Artefacts of the Model-Based Test Specification Process

model will be introduced later in Subsection 5.4.4. We measure the coverage reached by the automatically generated test model. The result of the model coverage measurement is the coverage report. The input for the coverage report generation are the analysis model together with the trace model.

The final result of the model-based test specification process are concrete test cases. Those are automatically generated from the manually extended test model and refine the logical test cases it contains.

To understand the holistic approach for model-based testing, we now provide a detailed description for each of the mentioned steps.

### 5.3 STEP 1. ANALYZE AND ANNOTATE THE ANALYSIS MODEL

The first step in our model-based test specification process is the manual testability check and prioritization of the input model as shown in Figure 41. The business analysts created the analysis model based on the requirement specification. Like the manual task of writing specification documents, also the modeling task is error-prone. Besides the typical modelling problems (for example modeling or naming conventions for specifying use cases

*Need for testability  
checks*



with activity diagrams could be violated), the testability with respect to the generation of a test model has to be checked. There are several context-related requirements (like behaviour completeness of use cases) for reaching high model quality in terms of testability as introduced by Voigt et al. in [VGEo8]. If such testability requirements are not met, then the automatic test model generation in step 2 can fail or result in a low quality test model. That is why the manual testability check is very important for the quality of the overall model-based test specification process.

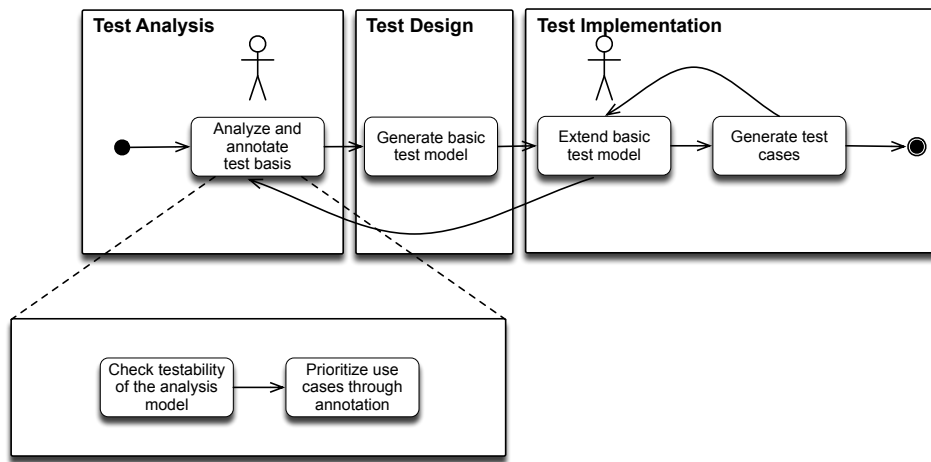


Figure 41: Refinement of the first step within the model-based test specification approach

Besides the mentioned challenge of testability assessment, another one arises here. The usage of sophisticated test selection algorithms in model-based testing can result in a very huge number of test cases and insufficient coverage of critical parts of the SUT. To prevent this, each test strategy should apply risk-based testing (introduction was provided in Subsection 2.1.4). The manual annotation task of the analysis model can drive the test selection algorithm to select test suites for the most important parts of the model.

*Risk-based testing*

### 5.3.1 Manual testability checks

In order to perform the testability checks the knowledge about the specification method and its meta-model from Section 2.4 is needed. The role responsible for this task is the test designer. In case of testability lacks in the analysis model, the test designer communicates with the business analysts to ensure higher quality. The modification of the analysis model should be performed only by business analysts and not test designer.

*Need for manual  
task execution*

The reason for the manual execution of this step lies in the test independency problem described in Section 1.1. Although there exist the possibility to automate several verification and validation checks, manual analysis of the analysis model still reveals testability lacks like missing consistency, completeness or correctness. Within the industry research project performed in this phd thesis, we have performed two assessments in large-scale projects at Capgemini Technology Services in Germany. The empirical results collected there, motivated the need for manual testability checks.

The goal of the testability check is not the verification of the analysis model with respect to its meta-model. With the check we aim to assure the testability of the analysis model, which helps us to generate high-quality test models. The testability checks are performed by the test designer on the following model elements with quality attributes mentioned in Voigt et al. [VGEo8]:

1. Use cases
  - Behavioural completeness
  - Pre- and postconditions
  - Dead actor or system actions
  - Deterministic transitions
  - Expressive action names and descriptions
  - Consistent guards
2. Dialogs
  - Trigger for each dialog action
  - Expressive dialog element names
3. Logical data type model
  - Enumeration types

The concrete testability checks can be generalized to any type of analysis model. Depending on the meta-model used, the list provided here can be extended or changed.

### *Testability of Use Cases*

Within this check, the testability of use cases of each conceptual component have to be investigated by the test designer. First, the *behavioural completeness* of each use case is checked. *Behavioural completeness* is defined as the existence of a behaviour model (in the case of the specification method used in this thesis, the UML activity diagram) for each use case. This is important, because otherwise no automated test selection can be performed. Also, the specification of detailed *pre- and post-conditions* is of importance for the holistic test generation. If the conditions are missing or are incomplete, the same case is for the pre-/postconditions of the generated test cases. This way the creation of the test case' initial state and the evaluation of its results is negatively affected.

Another quality attribute to be checked are *dead actor or system actions* within the behavioural model (here UML activity diagram). Since the test selection searches for paths from the initial to the final node, no dead nodes for the actor or system action should exist. This is important, because dead nodes can result in inappropriate test selection results. Also the behavioural model has to be checked for *deterministic transitions*, which means that each actor and system action should have only one outgoing transition. We do not restrict the number of incoming transitions for the mentioned actions, since loops in the behavioural model are allowed.

To guarantee the understandability of logical test cases, *expressive actions names and descriptions* should be checked. The expressiveness is assessed by the test designer who has the knowledge about the domain of the SUT. This is important, since we select test cases from use cases. The names and descriptions of use case actions (actor and system) are transformed to test case steps. The topic of model transformations will be described later in detail.

Finally, the use case' activity diagram should have *consistent guards*. Each decision node of such a diagram is described by a guard. The consistency of guards to the underlying data model has to be checked. This is important, since the test selection re-

sults in path which differ with respect to the guard value taken in the activity diagram. Inconsistent guards can results in test cases which are not understandable for test designers. In this thesis we use activity diagrams to refine use case descriptions. In other approaches different modelling languages for specifying the behaviour of use cases can be used. The concept of guards can be found in several of those languages. Therefore this testability check can be applied on other meta-models as the one used as an example in this thesis.

### *Testability of Dialogs*

The modelling approach used as an example in this thesis (see Section 2.4) distinguishes between dialog actions and dialog elements. The execution of each dialog action is triggered by a dialog element. Since this trigger is important for the execution of test case steps, the test designer checks if a *trigger for each dialog action* exists in the analysis model. Further, it is important that the dialog model has *expressive dialog element names*. Those names are used in the test case steps to support the test designer during the test execution. To execute a test case step is has to be clear what dialog elements should be used (for example fill with input data).

### *Testability of the Logical Data Type Model*

The last check performed by the test designer are the existence of *enumeration types* of the logical data type model. This is important, since the generated test model contains the test data structure, which is derived from the logical data types. The enumeration types deal later for the identification of test data partitions according to the equivalence class analysis method.

Several of the steps introduced above are simple checks, which can be performed by using a checklist. The other possibility is to automate the check of naming conventions, existence of model relations and attributes. Such syntactical checks can be easily automated and executed on the model level. The algorithms describing the constraints to be checked will always be customized according to the structure of the concrete analysis meta-model. That is why we do not provide an automation of this step. Instead, tools from the area of model checking can be used for that

purpose. Especially for automated testing of model consistency approaches as Engels et al. [EHRSo2] can be used.

### 5.3.2 Test prioritization through model annotation

One of the common problems with automatic test case generation is the test case explosion problem [AFGC03]. Since test cases are automatically selected by an algorithm, every possible coverage level can be reached. Very high coverage level results in a huge number of test cases. If test cases are executed manually, then their high number can make the execution impossible due to lack of resources. Even if a great part or even all the test cases can be automatically executed, the maintenance effort for test evaluation is not practicable<sup>1</sup> [UL07]. That is why intelligent model-based testing approaches integrating the risk-based testing concept are needed.

*Test case explosion  
problem*

To speak about the integration of risk-based testing in the holistic model-based testing approach, the topic of priority management has to be considered. The priority is defined in the ISTQB glossary [IST, p.32] as "the level of (business) importance assigned to an item, e.g. defect". In the context of use-case based test generation, the priority is assigned to single use cases. We refine this definition to a prioritization level, where the (business) importance is assigned to single use case steps, which are represented as action nodes of the activity diagram.

Our approach implements the risk-based testing idea with an annotation language which we introduced in Mlynarski et al. [MGSE09]. The purpose of our annotation language is to visually mark the model elements of the analysis model, which have high priority for testing. The annotation process reflects the natural test designers' method of operating. He combines his knowledge about the business functionality with the risks defined by the customer to identify the test-relevant parts of each use case. We use the term *test idea* here to describe useful indications as to

<sup>1</sup> For example if 10 000 test cases are generated and during the automatic test execution 2000 test cases failed, then each of the 2000 test logs has to be manually analyzed. A test case which has failed during its execution does not necessarily mean that a fault has been found. Also the test case itself can be incorrect. Assuming that the analysis of one test log takes 5 minutes, then for each test execution  $2000 \times 5 \text{ min} = 10000 \text{ min} = 166 \text{ hours}$  would be needed. This high maintenance effort lowers the ROI of the automatic test case generation.

what is to be tested. Test ideas are specified using an annotation language by test designers.

The prioritization strategy with test ideas aims at the use case by identifying the annotated use case steps. It means that a use case is marked with "high-priority" if one or more action nodes of the activity diagram are annotated by the test designer. We do not sum and normalize the number of annotated actions to provide a sophisticated priority classification. As analysis models of large-scale business information systems can contain several up to hundred use cases, we want to control the test generation process by selecting only the high-prioritized use cases. This way the restriction of the use cases taken as the test basis for automatic test selection and the restriction of the selection space within each preselected use case is performed.

We base our annotation language only on the behavioural model viewpoint. As we assume that model relations are used, the other viewpoints are automatically affected. Also we want to limit the test selection space, which means that the model elements used for the selection should be annotated. In our approach the selection is based on use cases. The following elements of the UML activity diagram refining a use case can be annotated:

- action nodes
- edges
- decision nodes

Besides the question what has to be annotated, also the how is very important. In the industry test case selection mostly relies on the experience of testers. They first identify the most important parts of the test basis by reviews or in discussions with the customer. Then, these parts are selected for test design and used for deriving test cases. In this process test designers develop ideas about what parts of the test basis have to be tested, because they are important from the business point of view. We call it test ideas and use this observation for the definition of our annotation language. The main target is to support the test case selection process with model-based techniques, while using the test designers' method of operating.

The visualization of the annotation language is done by using a UML stereotype called *TestIdea* and the green colour. Some

modeling tools (as Enterprise Architect<sup>2</sup>) support the automatic colouring for model elements having a predefined stereotype. This way the test designer only has to assign the stereotype *TestIdea* to action nodes, edges or decision nodes of an activity diagram.

An example of activity diagrams with annotated action nodes is shown in Figure 42. Our example shows three different annotations which lead to different test cases numbers. In the left activity diagram all action nodes were annotated, which means that the whole use case is very important and all nodes should be covered by test cases. The middle activity diagram contains four annotated action nodes which have to be covered. The last activity diagram on the right-hand side contains only two annotated action nodes. The resulting test case number differs because of the test selection criteria used later.

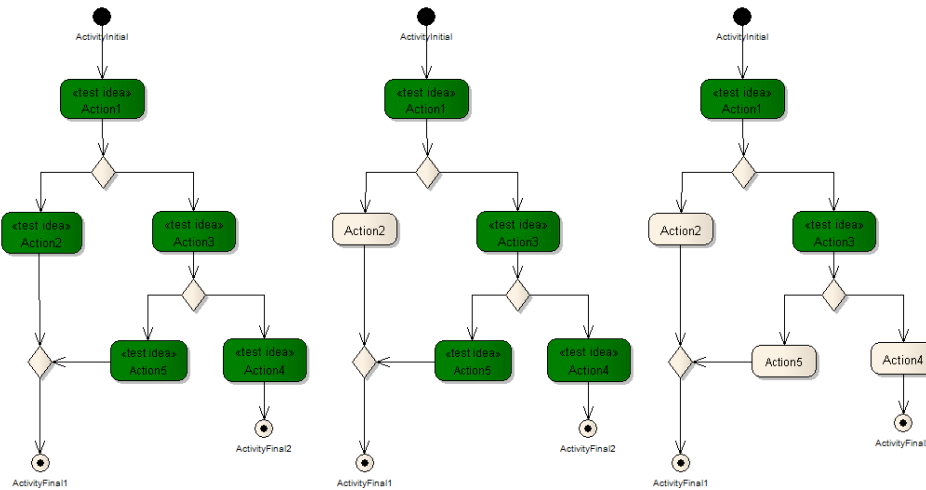


Figure 42: Example of annotated action nodes in activity diagrams.

Our test selection criteria has the goal to cover all paths within the activity diagram, which contain the annotated action nodes. We call it *AllPathsAnnotation*. We assume that between the annotated action nodes (or edges or decision nodes) within a path through the activity diagram the logical operator AND is used. Our test selection criteria is satisfied also in the case if several paths for the same tuple of annotated actions nodes exists. For example in the right-hand side activity diagram two paths could

<sup>2</sup> <http://www.sparxsystems.com>

possibly satisfy the test selection criteria. First path: *Action1*, *Action3*, *Action4*. Second path: *Action1*, *Action3*, *Action5*. In this case only the first or the second path already satisfies the *AllPathsAnnotation*.

The annotation language as described above can be applied for action nodes and edges identically. While annotating decision nodes a small difference exists. The test selection criteria for annotated decision nodes has the goal to cover all paths within an activity diagram which contain both outgoing edges from an annotated decision node. In this case we sharpen the definition of the logical AND operator. The generated paths should contain all combinations of outgoing edges from the annotated decision node. For the right-hand side activity diagram only both paths (*Action1*, *Action3*, *Action4* and *Action1*, *Action3*, *Action5*) satisfy the *AllPathsAnnotation*.

The quality of our annotation language in terms of number of generated test cases strongly depends on the number of annotated elements. If all action nodes, edges or decision nodes were annotated then no difference to classical test selection criteria (for example *AllPathsOneLoop* from Subsection 5.4.1) exists.

Within the first step the analysis model has been manually analyzed and annotated by the test designer. The next step is fully automated and results in a basic test model.

## 5.4 STEP 2. GENERATE BASIC TEST MODEL

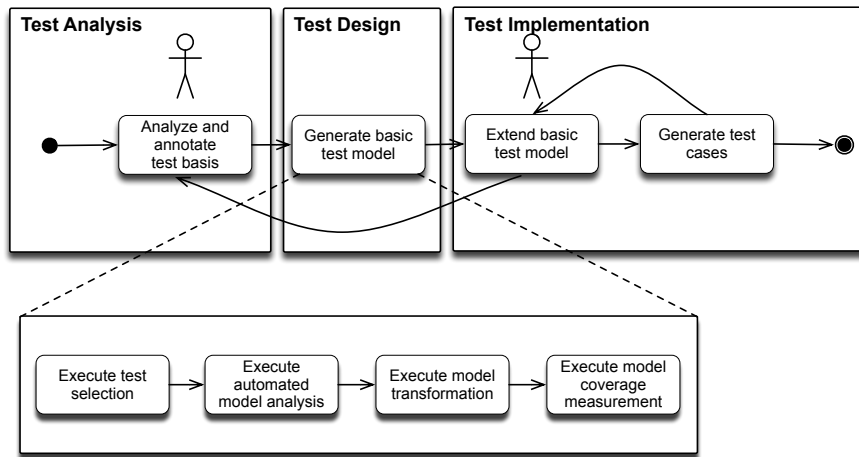
The automatic generation of the basic test model is the most important step of the model-based test specification process. The logical test cases are automatically selected and all information needed to complete the test case description is gathered from inter-related models. Through model transformations the test model is automatically generated. Finally, the reached model coverage of the analysis model is measured. For all mentioned tasks we define algorithms and specify them with the meta-model algebra. The tasks are depicted in Figure 43.

As mentioned in earlier chapters of this thesis, the test quality of the overall testing process depends strongly on the quality of the test specification document, namely the test cases. The test

*Influence on test  
case quality*



case quality depends strongly on the test selection method used (which influences the reached test coverage) and the completeness of test cases. The first aspect is related to the test selection algorithm and the second to the used model transformation.



**Figure 43:** Refinement of the second step within the model-based test specification approach

Model transformations can only be performed after the test selection is done. That is why we first describe the test selection algorithm and afterwards the model transformation used.

#### 5.4.1 Test Case Selection

In Section 2.2.4 we introduced the different types of test selection criteria and basic definitions. Each criteria depends on the behavioural model type used as basis for test selection. The criteria drive the reached model coverage. In order to use a test selection criteria a formal algorithm has to be defined.

In the case of the analysis model used in this thesis (see Section 2.4), we focus on use case models which are described by UML use case and activity diagrams. Test cases can be selected only from the activity diagrams. The use case diagrams describe only the behaviour decomposition in actor, use cases and the relations between them. As mentioned in Section 2.2.4 there are several transition-based test selection criteria which can be applied for activity diagrams.

*Different test selection criteria*

Our approach does not depend on a certain test selection criteria. Since the application of our holistic model-based testing approach should be possible for different types of analysis models (assuming meta-models fulfilling the modelling viewpoints and model relations property), several test selection criteria can be applied. However, for evaluation purposes we have to define one or more which will be used. We select the following test selection criteria to show the difference with respect to the number of generated test cases:

1. *AllPathAnnotation* (low number of test cases)
2. *All-Actions* (medium number of test cases)
3. *AllPathOneLoop* (high number of test cases)

All mentioned test selection criteria try to find paths (see definition in Subsection 2.2.4) from the initial to the final node. The difference lies in the coverage goal (actions, paths and annotated actions). While the *All-Actions* algorithm searches for a minimum set of paths which cover all actions, the *AllPathOneLoop* algorithm searches for all possible paths in the activity diagram. Since the number of paths generated by *All-Actions* is less<sup>3</sup> than the one found by *AllPathsOneLoop*, the number of resulting logical test cases differ. In the case of the *AllPathAnnotation* algorithm, the number of paths depends on the number of annotated action nodes. We assume that the number of annotated actions is less than the overall number of actions. That is why this algorithm generates a smaller set of test cases than the *All-Actions* algorithm. Similar observations on test selection criteria with respect to the generated test case number can be found in Liggesmeyer [Lig09] or Utting and Legeard [UL07].

Before we introduce the concrete test selection algorithm, we first specify the according algebra operation. In Chapter 4, we have introduced the purpose and language needed to specify algebra operations. The test selection algorithm presented here refines the operation called *select*. In the following we specify the operation according to the template from Section 4.6. Additionally in Figure 44 we depict this operation based on the relevant part of the UML meta-model needed for test selection.

- Name = select

<sup>3</sup> Depending on the model complexity and loop types (forward or backward) the number of paths generated by *All-Actions* is less or equal to *AllPathOneLoop*.

- Goal = behavioural elements of the UML meta-model, namely the UML activity diagram are traversed to identify paths (sequences of action nodes)
- Type = Single meta-model operation
- Input meta-model = UML meta-model
- Input set = elements of the behaviour package of the UML meta-model representing activity diagrams
- Output meta-model = UML meta-model
- Output set = sequence of actions from the initial to the activity final node
- Property = traversability
- Visualization = see Figure 44

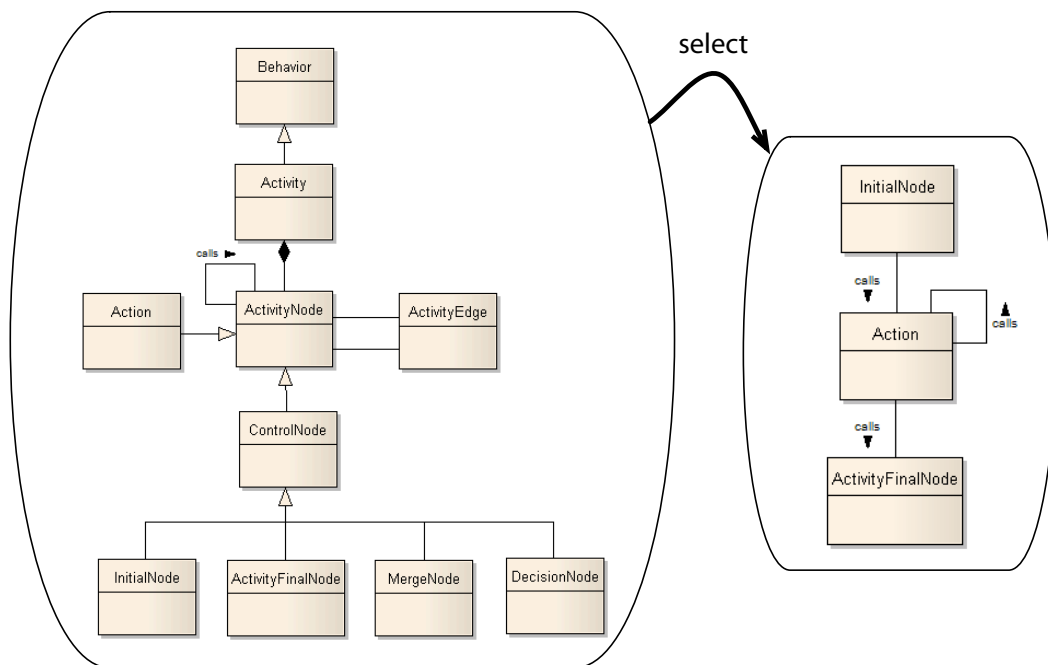


Figure 44: Meta-model algebra operation for the test selection

The operation *select* traverses the activity diagram (input set), which always consists of an initial, final and several other nodes (like decision, or merge) connected by activity edges. For the purpose of this thesis we do not use fork and join nodes, since the parallelism of workflows is a separate problem during test selection (see [RBGW10] for detailed analysis).

The output set of the operation are several paths through the activity diagram. The definition of a path was already provided in Subsection 2.2.4. Since there can exist an infinite number of paths in an activity diagram, the algorithm refining the algebra operation has to work on certain assumptions (like a path is visited only once). We will now provide the description of the test selection algorithm for the *select* algebra operation.

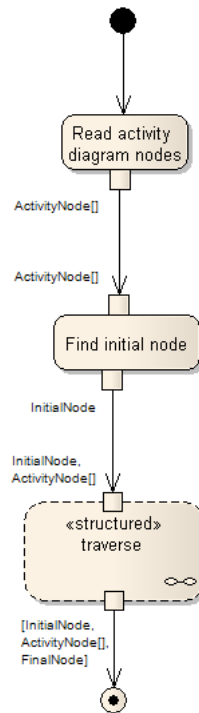


Figure 45: Main algorithm for the test selection

As the concrete test selection algorithms differ only slightly, we provide the algorithm for *AllPathAnnotation* only. Our selection is based on a heuristic and recursive algorithm which traverses each node of the activity diagram. This algorithm is depicted in Figures 45, 46 and 47 and uses the elements of the UML meta-model as defined in the *select* algebra operation. The diagram from Figure 45 is a high level workflow of the test selection algorithm. The selection is performed by the activity "traverse", which recursively traverses the activity diagram. The original version of the test selection algorithm was introduced by Gutierrez et al. in [GEMTo6]. We have extended the algorithm to re-

strict the loop visiting and deal with annotated nodes. We will now briefly describe the selection algorithm.

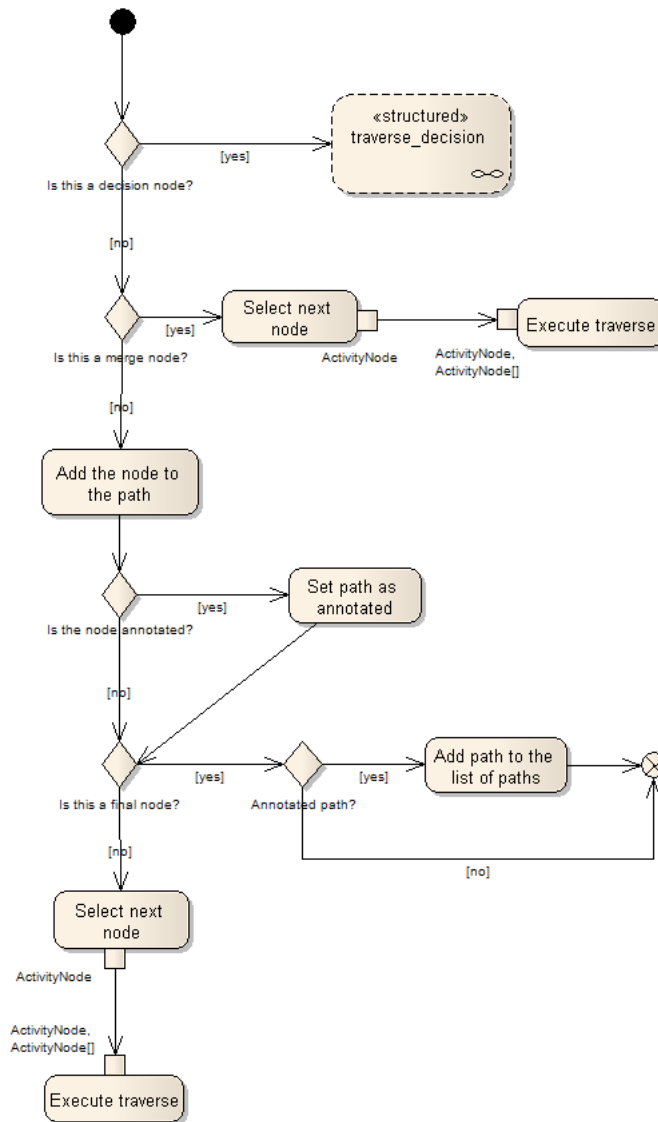


Figure 46: Recursive algorithm which traverses the activity diagram

Our test selection algorithm is invoked only on annotated use cases. It starts with reading the activity diagram and searching for the initial node. In step 3 within Figure 45 the algorithm starts to traverse the activity diagram beginning with the *InitialNode*. In Figure 46 the algorithm acts differently for decision (*DecisionNode*), merge (*MergeNode*) and final nodes (*ActivityFinalNode*). If none of the mentioned cases is given, the succes-

sor node is selected and the algorithm is executed recursively. The algorithm terminates each time a final node is found. Since we want to cover only paths with annotated action nodes, we mark each path as annotated if one or more annotated nodes were found during the test selection. A node is annotated only when it is stereotyped as *testidea*. This stereotype is used by the test annotation language from Subsection 5.3.2. In this case the path created by storing the visited action nodes is stored within a global path list. Otherwise, the current recursion of the algorithm is terminated.

In the case of visiting a merge node the algorithm simply selects the successor node and is executed recursively.

More complex is the case of visiting a decision node. In Figure 47 a subalgorithm for decision nodes is shown. First, all outgoing edges, which we call alternatives are selected. For each alternative the connected node *ANode* is selected. The *ANode* can be of the type *InitialNode*, *ActivityFinalNode*, *DecisionNode* or *MergeNode*. If the node was not visited yet and it is not a merge node, then it will be added to the path created so far. Then, the successor node is selected and the traverse algorithm is executed recursively. In the case the node was already visited (node is contained in the path list) the next alternative from the decision node is selected. This way, a loop is visited only once during the test selection. For the connected node *ANode'* the traverse algorithm is executed. The algorithm from Figure 47 is executed until all alternatives were selected.

The final result of the test selection algorithm is a list with paths through the activity diagram. A path contains of the *InitialNode*, several *ActivityNodes* and the *FinalNode*. The algorithm always terminates since loops can be visited only once (see Figure 47) and each time reaching the final node terminates the recursive execution of the algorithm.

To visualize the test selection algorithm, we provide a small example. As basis for the test selection we use an activity diagram depicted in Figure 48, which describes the use case *Book Attendee on Course* from the example introduced in Subsection 2.4.4. The diagram consists of 7 action nodes, 5 decision nodes and 2 merge nodes. The execution of the *AllPathAnnotation* algorithm resulted in 18 paths. Two exemplary paths are depicted in Figure 48 with the red and green color. Since the first node is a decision, the

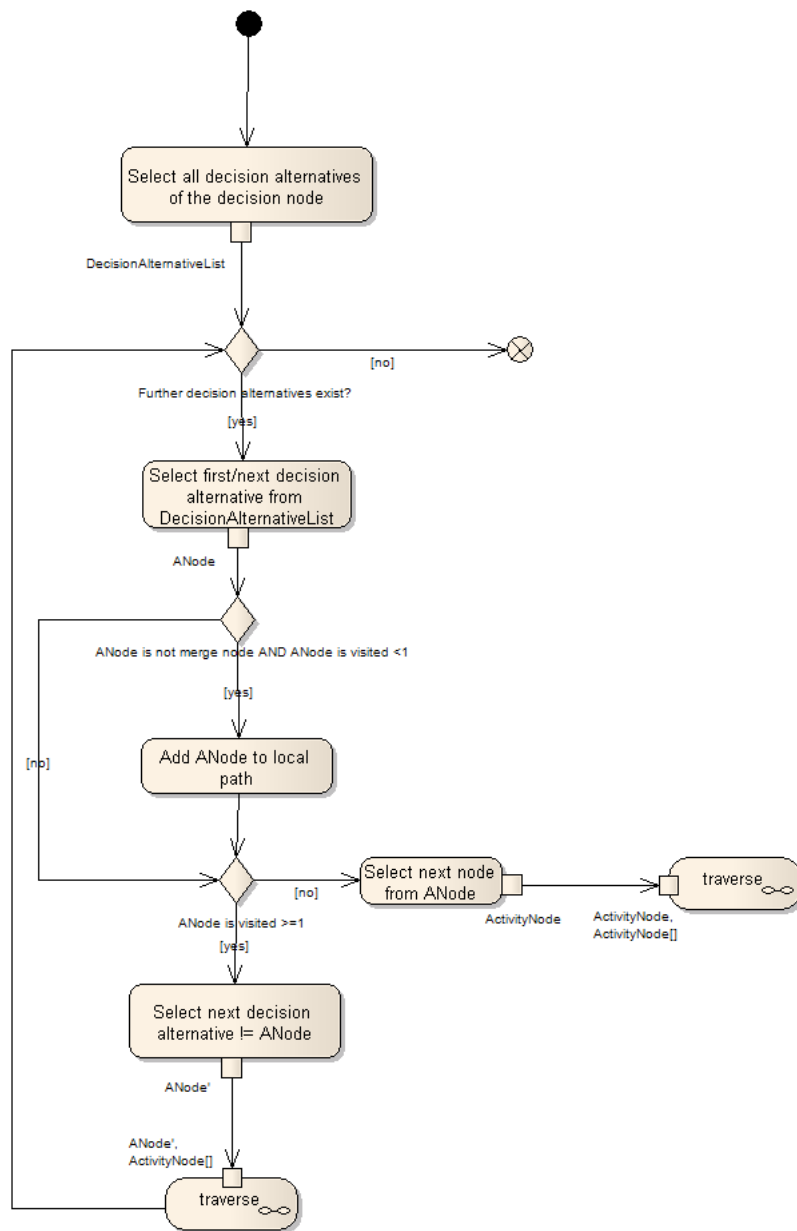


Figure 47: Subalgorithm which traverses the decision nodes of the activity diagram

algorithm will recursively traverse each outgoing edge of this node. The decision node called *"course\_data"* results in a backward loop to the *"Search\_Course"* action node. This loop is executed only once by the red path. This is also the case in all other loops which are covered by all 18 paths.

The results of the test selection algorithm are paths of the activity diagram, which represent the logical test cases. As mentioned at the beginning of this section further information is needed to complete the test case description. A selected path contains only action nodes of the activity diagram from the initial to the final node. This way, all test case steps are specified. A logical test case has to further contain pre- and postconditions, names of dialog elements and logical data types for each test case step. The goal is to fill all test case attributes from Section 2.1.1 with data.

To collect the data needed for the specification of logical test cases, we use model analysis algorithms together with model transformation rules. Both techniques will be described in the next subsection.

#### 5.4.2 Automated Model Analysis

At the end of the test selection process several paths through the activity diagram are created. The activity diagram provides the following information for each node:

- Name
- Description
- Incoming edge (in the case of a predecessor decision node with the according guard)
- Outgoing edge
- Call to another use case or application function
- Swimlanes<sup>4</sup>, which reference conceptual components and actors involved

Further the use case model, which the activity diagram belongs to provides the following information:

---

<sup>4</sup> Swimlanes are used in the exemplary UML modeling approach in this thesis. This modelling construct is not necessarily a part of other modelling languages.



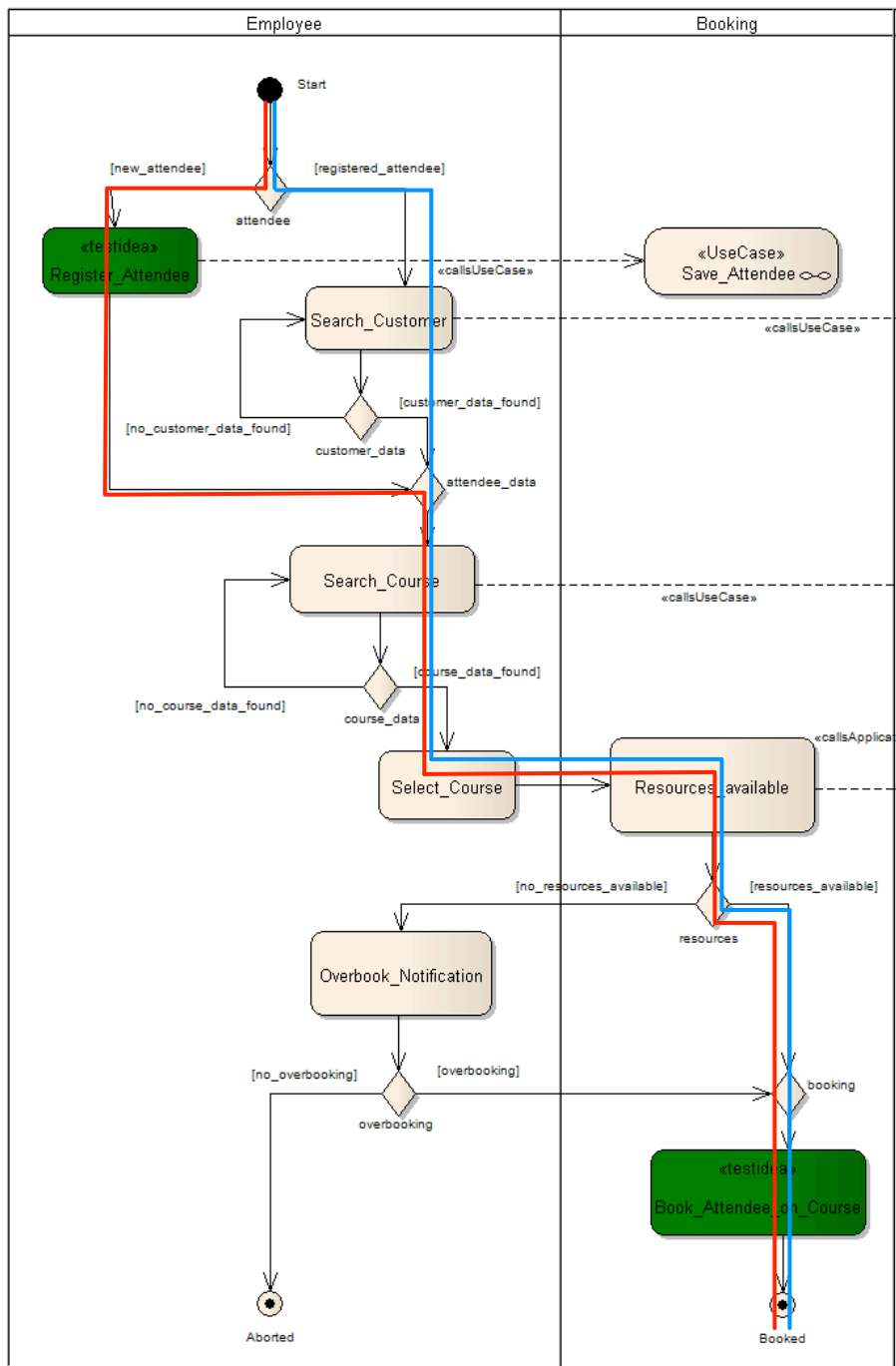


Figure 48: Example of two paths selected according to the *All-PathAnnotation* test selection algorithm

- Title
- Description
- Trigger
- Precondition
- Results
- Actors

The mentioned information can be used to partially specify a logical test case in a test model. This kind of test case is not complete in terms of the test execution. Important information as the used user interface elements or input and output data is missing here. To fully specify a logical test case, further information is needed. Where to search for this information and how to extract it is described in the following.

First, the information about the dialog elements and dialog actions used in each use case action is needed. The test cases automatically selected by the test selection algorithm have to be executed. Independent if executed manually or automatically the dialog information is needed to interact with the system during the tests.

Second, the information about the data model used in each use case action is needed. Test cases without test data are worthless. The derivation of concrete test data sets is possible if logical data types are provided in the description of the logical test case.

Both sources (dialog and logical data model) are not orthogonal, but inter-related to the use case in the analysis model. The extraction of the related dialog actions, its dialog elements and logical data types is possible by using model relations and model analysis algorithms.

The model relations between the elements of the analysis model were already introduced in Section 2.4.11. To define a concrete model analysis algorithm, the model relations shown in the excerpt of the analysis meta-model in Figure 49 are important. Especially the trace between use case actor action  $\rightarrow$  dialog action  $\rightarrow$  dialog element  $\rightarrow$  logical data type is needed to collect the information related to the logical test case.

Knowing that the model relations can be used to extract further information for test case specification, we now have to define an algorithm, which navigates through the relations and collects the

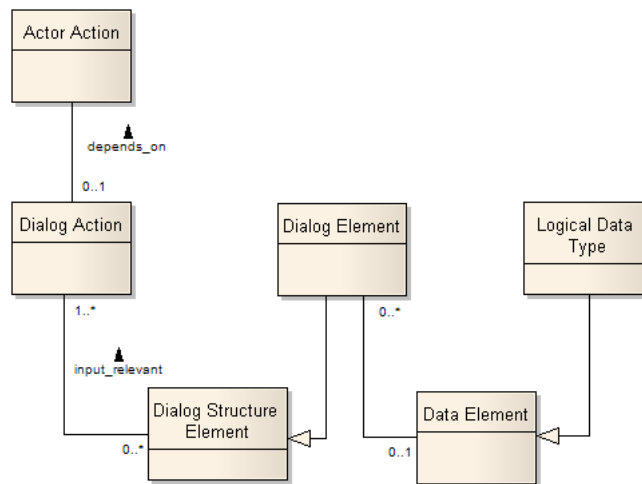


Figure 49: Model relations within the analysis meta-model important for model analysis

needed information. The algorithm is working on the instance of the meta-model introduced in Subsection 2.4.11.

The model analysis algorithm refines the meta-model algebra operation called *extract*. In the following we specify the operation and additionally in Figure 50 we depict this operation based on the relevant part of the analysis meta-model needed for model analysis.

Operation *extract* according to the template from Section 4.6:

- Name = extract
- Goal = elements of the analysis meta-model are analysed to identify and extract context-related information
- Type = Single meta-model operation
- Input meta-model = analysis meta-model
- Input set = all elements of the analysis meta-model
- Output meta-model = analysis meta-model
- Output set = elements of the analysis meta-model associated through context-related model relations
- Property = modelling viewpoints, model relation
- Visualization = see Figure 50

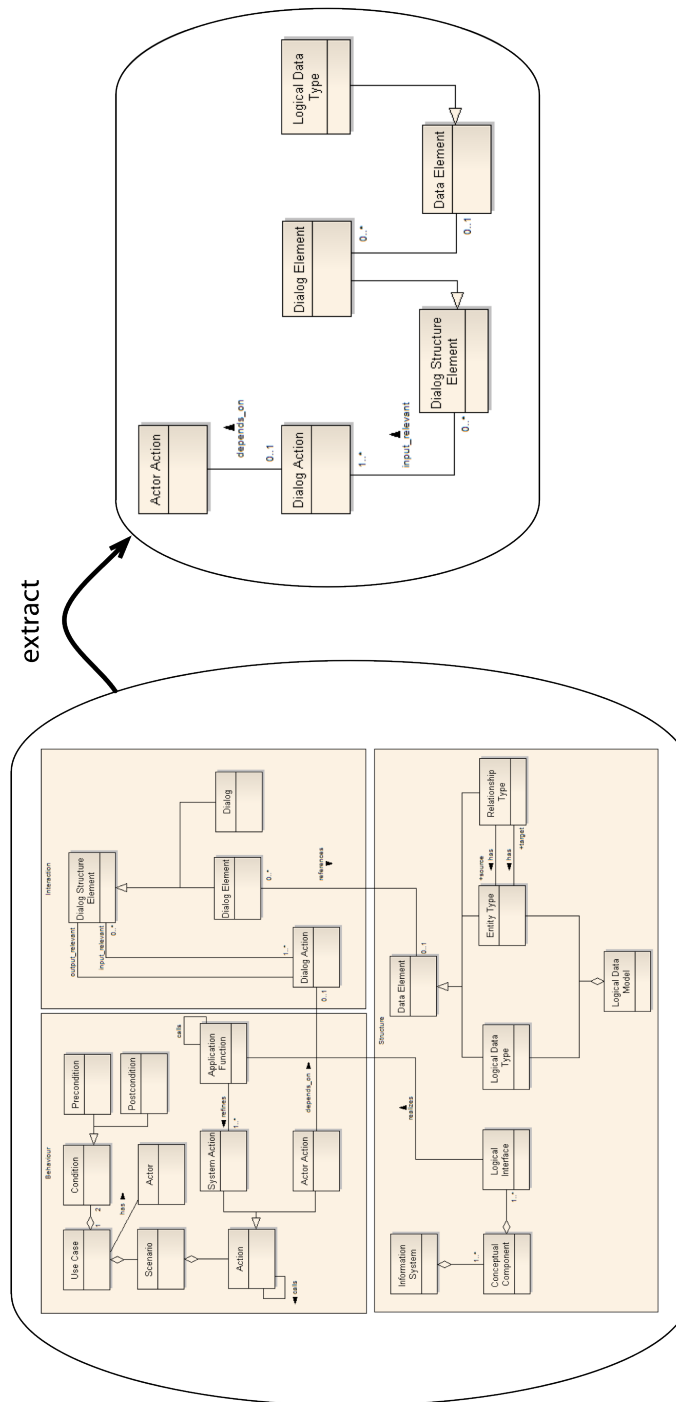


Figure 50: Meta-model algebra operation for the model analysis

The operation *extract* uses the whole analysis meta-model as the input set. The output set is a subset of this meta-model, which consists of elements with context-related model relations. The context is defined by the method engineer, who understands the semantic of modelling viewpoints and model relations between the elements of the output set. In this thesis the context is given by using the relations from Figure 49. The model analysis algorithm which navigates through the analysis model according to the *extract* algebra operation will be now described.

Our model analysis algorithm is depicted in Figure 51. The goal of this algorithm is to extract information about dialog actions, dialog elements and logical data types related to the selected path from the use case' activity diagram. The modelling approach for analysis models used as an example in this thesis is based on the UML. The UML meta-model fulfills the meta-model property *model relation* (see Section 4.5) by using the concept of *Association*. The UML specification defines an association as "An association describes a set of tuples whose values refer to typed instances. An instance of an association is called a link" [Obj09, p.39]. This way UML meta-model elements of the type *Class* can be associated with each other and the navigation between them at model level is possible by using links. By using the model relations from Figure 49 as instances of the UML class *Association*, the analysis algorithm can navigate through the analysis model.

*UML concept of associations used to navigate through the analysis model*

First, all actor actions of a use case are listed. This is done by listing all action nodes, which do not represent a system action. Then our algorithm finds a related dialog action for each actor action. This is done by using a relation<sup>5</sup> mentioned above. For the found dialog action all related dialog elements are listed. Finally, for each dialog element found the algorithm searches for the related logical data types. The result of our algorithm are the dialog actions, dialog elements and logical data types related to the actor actions of a use case given as input. This tripel set of model elements identified by the algorithm has a context important while testing a actor action. It supports the stimulation of the SUT with input data (list of *LogicalDataType*) through ele-

<sup>5</sup> In the model instance on which the algorithm operates the relations are instantiated by model links. The links are manually created by business analysts during the modelling task.

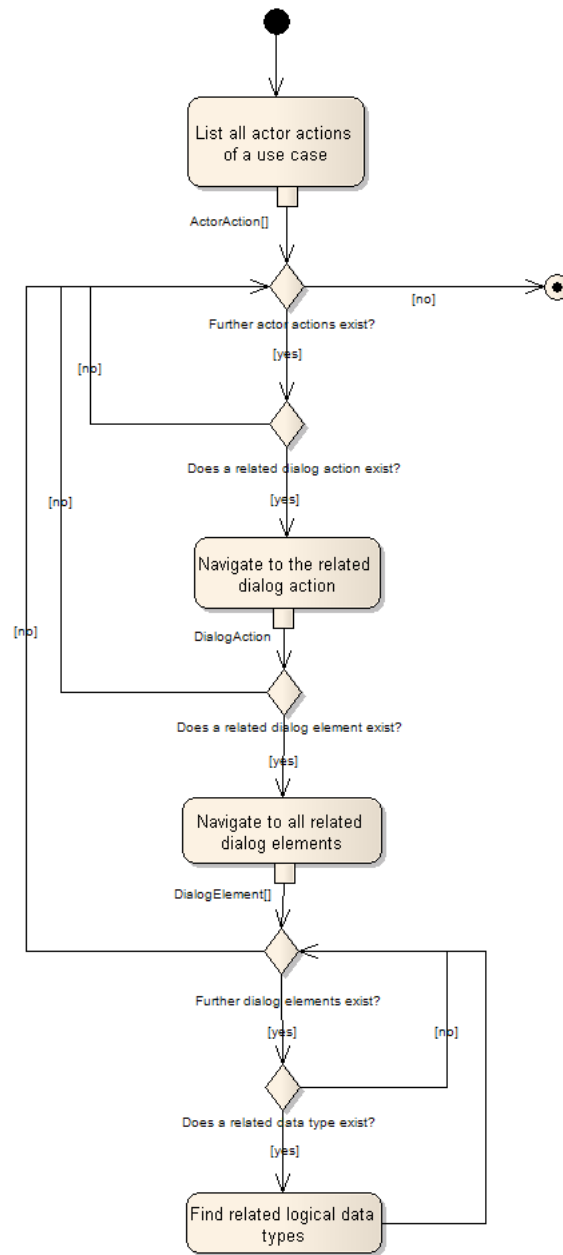


Figure 51: Algorithm for Automated Model Analysis

ments of the user interface (list of *DialogElement*) within a state, namely the context of an *ActorAction*.

In Figure 51 the following cases are crucial for the extraction of information from the different modelling viewpoints:

- There exist no related dialog action for an actor action
- There exist no related dialog element for a dialog action
- There exist no related logical data type for a dialog element

Without providing empirical evidence at this point, we can state that: If the first case occurs, then the automated model analysis which is an essential part of the holistic view has the lowest impact on the reached internal test quality as well as the reached model coverage. The three cases reflect the situation where none or few of the model relations specified in the meta-model were transcribed by business analysts while creating the analysis model. This way no linkage between the behaviour, interaction and structure modelling viewpoint in the model instance exists. The automated model analysis algorithm presented here cannot navigate from the behaviour to the interaction and further to the structure models. The essential context-related information about the user interface and input data is missing. Based on this fact, the following correlation can be formulated:

*Correlation between  
model relations and  
holistic view*

$$\text{holistic} \sim \text{rel} \quad (5.1)$$

The variable *holistic* stands for the application of the holistic view on analysis models in model-based testing. With the *rel* variable we symbolize the level of model relations instantiated according to the analysis meta-model by business analysts. In particular it is the degree of transcription of model relations at the model instance level. When the number of model relations grows, then the application of the holistic view is growing. This way, the holistic model analysis is directly proportional to the model relations.

Our model analysis algorithm navigates through the model instance assuming a certain structure of the models. In this thesis the structure is given by the meta-model of the specification method from Section 2.4 and the meta-model of the UML [Obj09] as an example. However, the automated model analysis

*Application on other  
meta-models*

presented here can be applied on every meta-model structure supporting semantical relations between different modelling viewpoints. According to the concept of the meta-model algebra, those meta-models fulfill the property *model relation*. In this case a customization of the algorithm presented in this subsection according to other meta-models is needed. Those meta-models are further important for the execution of model transformation rules, which will be described in the next subsection.

### 5.4.3 Model Transformations

Within this section we assume the basic knowledge about model transformations from Section 2.5.

In order to create a transformation definition, we first have to define a clear mapping between the source (analysis model) and the target (test model) of the transformation. We do it by introducing an *artefact mapping table* and a *UML mapping table*.

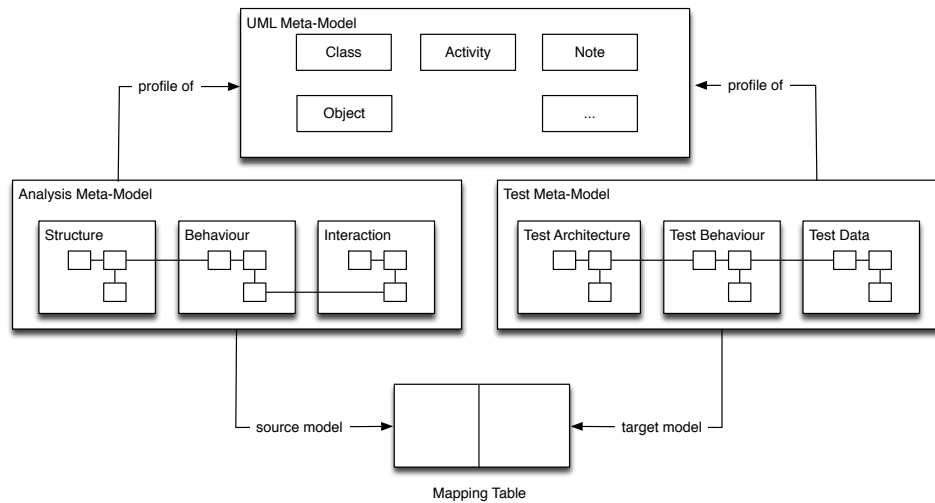
**ARTEFACT MAPPING TABLE** defines the mapping between elements of the analysis meta-model and the test meta-model. Example of such a mapping: *ActorAction* (analysis meta-model) to *TestStep* (test meta-model).

**UML MAPPING TABLE** defines the mapping between UML elements used in the analysis meta-model and the UML elements used in test meta-model. The UML mapping table can be derived from the artefact mapping table, since each element of the analysis and test meta-models is projected to an element of the UML meta-model. Example of such a mapping: *Package* (e.g. package containing a *UseCase* in the analysis meta-model) to *Class* (e.g. class describing a *TestContext* in the test meta-model).

In Figure 52, we depict the artefact mapping table and the different meta-models. To define the mapping table information about the analysis meta-model is needed. We define it as the source model for the transformation process. The target model is the test meta-model. Both meta-models are placed on the M2 level of the meta-level introduced by MOF [Obj06a]. The meta-model can be seen as profiles of the Unified Modeling Language. That is why in Figure 52 the UML meta-model is used as the highest level of abstraction.



The structure of the analysis model and the according UML diagrams were already introduced in Section 2.4. The test model and especially the UML testing profile were introduced in Subsection 2.3. Having that knowledge, we can now introduce the artefact mapping table used in our approach.



**Figure 52:** Relation between different meta-models and the mapping table

Table 5 contains the definition of all mappings between the analysis and test model. Each row results in a single model transformation rule, which is executed while generating the test model. The mapping from Table 5 were created based on the knowledge about the semantics of both meta-models. In Figure 53 we have depicted the high-level mapping between the different meta-model viewpoints. The structure viewpoint of the analysis model can be mapped to the test architecture viewpoint since both viewpoints describe the structure characteristics. Additionally, the test structure viewpoint can be mapped to the test data viewpoint since it described the test data structure (see logical data type model in Section 2.4). The behaviour viewpoint of the analysis model can be directly mapped to the test behaviour viewpoint. Additionally, parts of the behaviour viewpoint can be mapped to the test architecture since some behaviour meta-model elements (for example use case) create the context of test architecture meta-model elements (for example test context). Finally, the interaction viewpoint of the analysis model can be mapped to the test behaviour viewpoint. This mapping results

from the model relations mentioned in Subsection 5.4.2. Knowing the motivation for the mapping, we will now describe the concrete mapping from Table 5.

**Table 5:** Mapping table between the analysis and test meta-model

Modelling viewpoint	Model element	To test model
Behaviour	Use Case	Test Context
	Precondition	Precondition
	Postcondition	Postcondition
	Actor	Test Component
	Scenario	Test Case
	Actor Action	Test Step
	System Action	Check Step
Interaction	Dialog, Dialog Element	DialogNote
	Dialog Action	TriggerNote
Structure	Conceptual Component	Test Component, SUT, Data Pool
	Logical Data Type	DataNote, Data Pool

*Mapping between  
meta-model  
viewpoints*

In our example the use case is mapped to a test context. Use case elements as actor, scenario, actor and system action are mapped to the according test model elements. We define also a mapping between the use case' pre- and postconditions and the according conditions in the test model. The conceptual components are mapped to the test architecture elements (test component and SUT) and to the data pool. The logical data types identified through the model analysis algorithm from Subsection 5.4.2 are mapped to data notes and attributes of the data pool. In total we define 14 mappings on this granularity level. For simplicity we did not define too fine mapping rules. For example, a use case references the conceptual components through swimlanes within the activity diagram. The swimlanes, decision nodes, etc. of an activity diagram could be also defined as single mappings. This details related to the UML subset used within the specification method are implemented within model transformation rules.

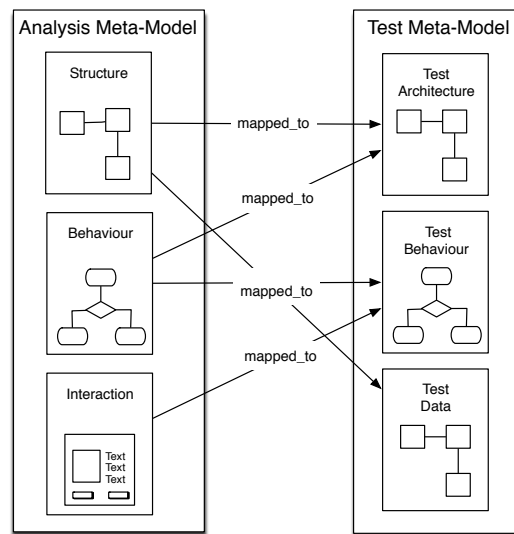


Figure 53: Relation between different meta-model viewpoints

The single model transformation rules are specified based on the mapping table shown in Table 5. They are implemented directly in the code of the model-based test specification tool support. As shown in Section 2.5 there exist different model transformation languages. To introduce some examples of single model transformation rules here, we use the Epsilon Transformation Language (ETL)<sup>6</sup>. Examples of concrete model transformation rules help to understand the details of single mappings. Especially the assignment of attributes of meta-model elements is interesting here. Two exemplary mappings from Table 5 are shown as ETL rules in Listing 5.1.

The two model transformation rules *UseCase2TestContext* and *UCConditions2TCCConditions* work directly on the meta-models *AnalysisModel* and *TestModel*. The operator *transform* searches for matches within the model instance of the analysis model while the *to* operator creates the model elements in the instance of the test model.

The single model transformation rules shown above are instances (in terms of implementation) of the mapping table from Table 5. In Figure 54 we classify the model transformation in the MOF levels introduced in [Obj06a]. The model transformation rules together with the model instances of the analysis and test model

<sup>6</sup> <http://www.eclipse.org/gmt/epsilon/>

Listing 5.1: Single model transformation rules in ETL

```

rule UseCase2TestContext
  transform ucp : AnalysisModel!UseCase
  to tc : TestModel!TestContext
  {
    tc.name = ucp.name;
  }

rule UCConditions2TCConditions
  transform ucpre : AnalysisModel!PreCondition,
            ucpost : AnalysisModel!PostCondition
  to tcpre : TestModel!PreConditionNote,
      tcpost : TestModel!PostConditionNote {

    ucpre.description = tcpre.description;
    ucpost.description = tcpost.description;
  }

```

belong to the M<sub>1</sub> level. The mapping table together with the meta-models and the UML belong to the M<sub>2</sub> level. The highest level of abstraction is the MOF framework on M<sub>3</sub>. The mapping table uses the analysis and test meta-models to define a mapping. This mapping is instantiated by the model transformation rules which expect the analysis model as the input and the test model as the output of the transformation.

Within the MOF classification of model transformations, also the specification of the according meta-model algebra operation (see Chapter 4) is important. The specification of the algebra operation belongs to the M<sub>2</sub> level, where the analysis and test meta-models are defined. Also the algorithm (refinement of the algebra operation) belongs to the M<sub>2</sub> level. Both meta-artefacts (algebra operation and algorithm) provide the specification needed to implement model transformations within our approach.

Operation *transform* according to the template from Section 4.6:

- Name = transform
- Goal = elements of the analysis meta-model are transformed into elements of the test meta-model
- Type = Multiple meta-model operation
- Input meta-model = analysis meta-model
- Input set = elements of the analysis meta-model

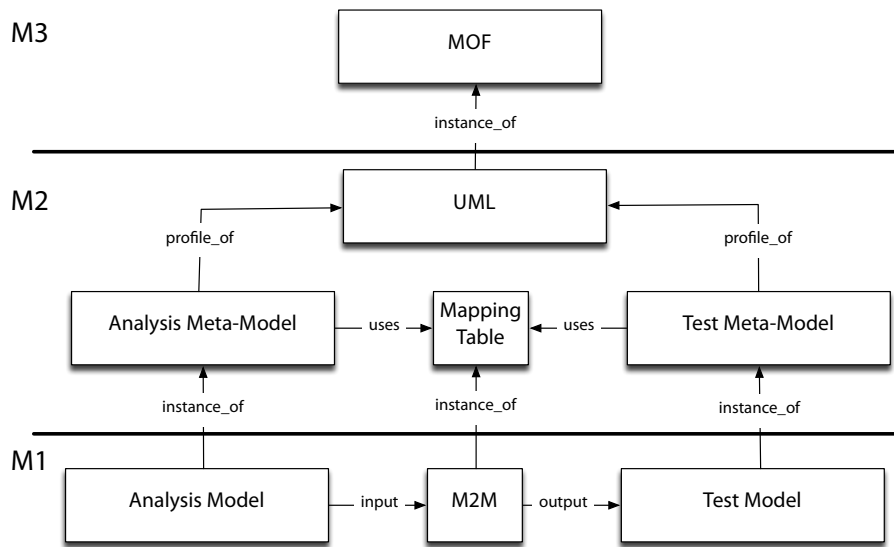


Figure 54: Overview of the solution according to MOF levels

- Output meta-model = test meta-model
- Output set = elements of test meta-model
- Property = structural mapping
- Visualization = see Figure 55

The operation *transform* uses the whole analysis meta-model as the input set. The output set is the whole test meta-model. The mapping between both meta-models is defined by the method engineer, who understands the semantic of the elements of both meta-models.

The whole model transformation process is depicted in Figure 56. First the meta-model definitions are read by the algorithm. Then the instance of the analysis model is read according to its meta-model. The conformance to the meta-model is important, because otherwise the model transformation rules would fail. In the third step an empty instance of the test model is created. Then the list of all transformation rules is read and executed sequentially in a loop. For each model transformation rule the model elements in the analysis model are matched and the according model elements within the test model are created. The algorithm terminates after executing all transformation rules.

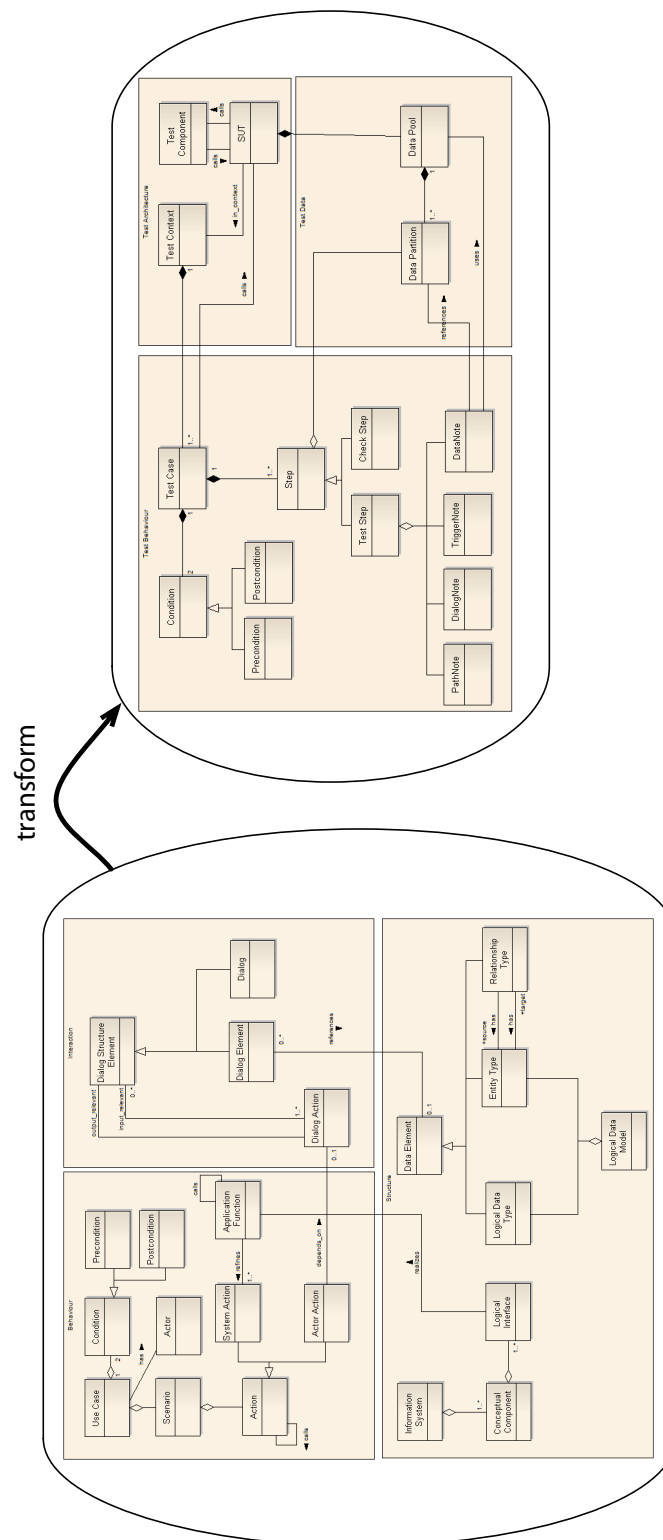


Figure 55: Meta-model algebra operation for the model transformations

The sequential execution order of the transformation rules is needed, because several test model elements have dependencies as introduced in its meta-model (see Subsection 2.3). For example the use case package has to be created before the activity diagram for a test case, since each test case (and its activity diagram) is placed within the use case package. Another example is the test context class which operations present the names of all test cases generated for a use case. The test context class has to be created before the transformation rules for test cases are executed.

*Order of model  
transformation rules*

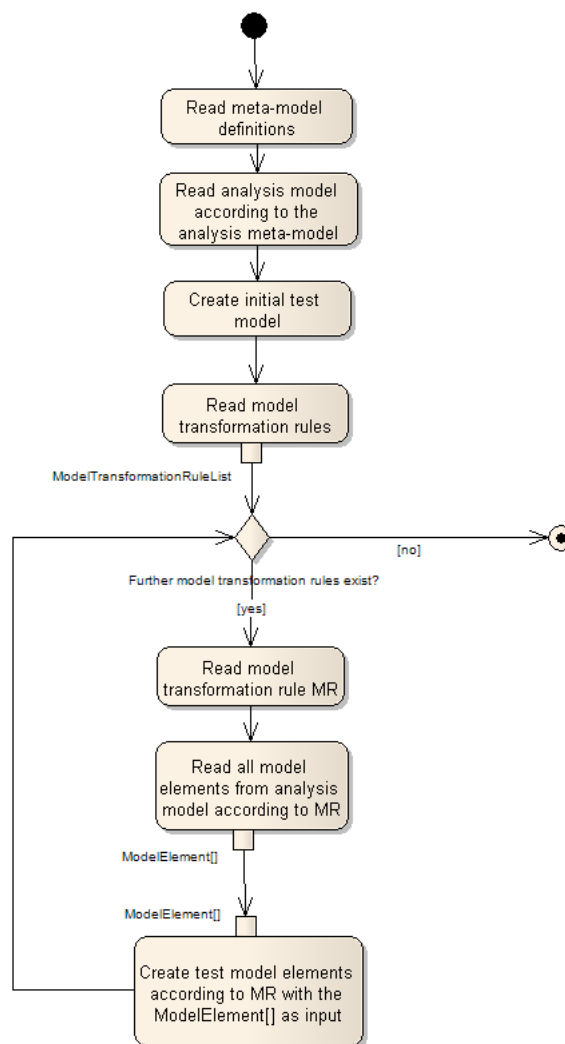


Figure 56: Algorithm for Model Transformation

While executing the transformation rules as shown in Figure 56 a problem arises. Since we select paths and perform model analysis to search for related information, some model transformation rules should work only on the gathered information and not the complete analysis model. That is why we differ between transformation rules executed before the test selection takes place and after it.

The following model transformation rules are executed before the test selection takes place:

- UCPackage2TestModelPackages
- UCPackage2DataPoolClass

The following model transformation rules are executed after the test selection takes place:

- UCActorAction2TCStep
- UCSystemAction2TCheckAction
- UCConditions2TCConditions
- Dialog2TCStepDialog
- DialogTrigger2TCStepTrigger
- LogicalDataType2TCStepData
- LogicalDataType2DataPool
- Swimlane2TCSwimlane
- Swimlane2TestComponent
- ConceptualComponent2SUT

Earlier in this subsection, we have introduced the high-level mapping table between the analysis and test meta-model (see Table 5). The pre- and post-selection transformation rules mentioned here extend the high-level table. The full specification of all mappings and their according model transformation rules does not provide further cognitions as the ones described until now.

*Exemplary  
transformation  
results*

To visualize the model transformations described in this subsection, we provide an excerpt of a use case (see Figure 57) and a automatically generated logical test case (see Figure 58). By using the test selection algorithm from Subsection 5.4.1 the red marked path in Figure 57 was selected. For both actions *Register\_Attendee* and *Search\_Course* the model analysis algorithm from Subsection 5.4.2 collects the related dialog actions, dialog elements and log-



ical data types. Through the execution of several model transformation rules described in this subsection logical test cases like the one from Figure 58 are created.

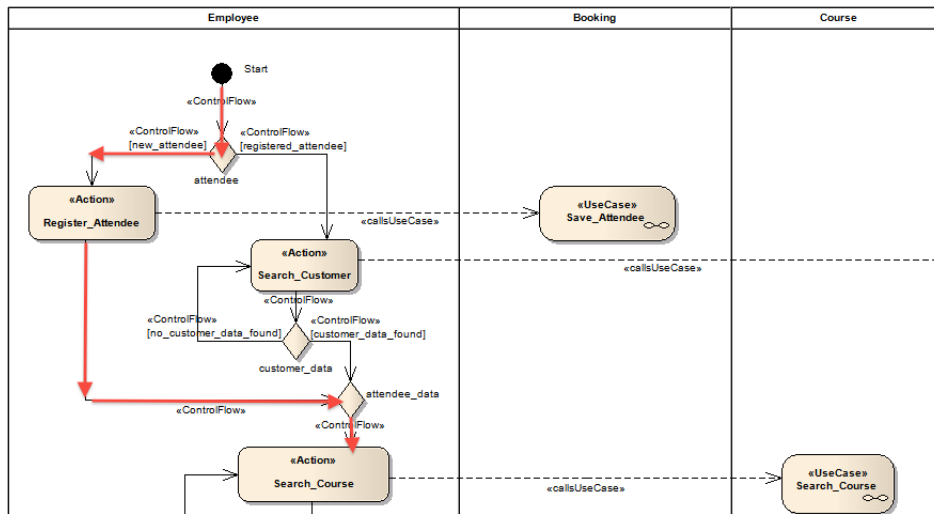


Figure 57: Path within a use case taken as the input for the transformation

Beside the logical test cases, also the test architecture model elements like the test context, test component and SUT classes are created. This way, the model elements needed to specify a test model according to the UML Testing Profile (see Subsection 2.3) are created. In Figure 59 we introduce the mentioned elements of the test model for the use case *Book\_Attendee\_on\_Course* (see running example from Subsection 2.4.4). The test context class groups all logical test cases selected with the test selection algorithm. There exist two naming conventions for test cases: sequence of integer numbers from 1 to n (where n is the number of generated test cases) or path names (which consist of guard names from the decision alternatives takes during the selection in an activity diagram). Through model analysis algorithms the SUT and test components as the associations between them could be identified. By executing the model transformation rules all mentioned elements of the test model were automatically created.

The model transformation technique enables the automation of a great part of the test specification process. It also supports the traceability of the test model elements, since the source of each of them is specified within the model transformation rules. The

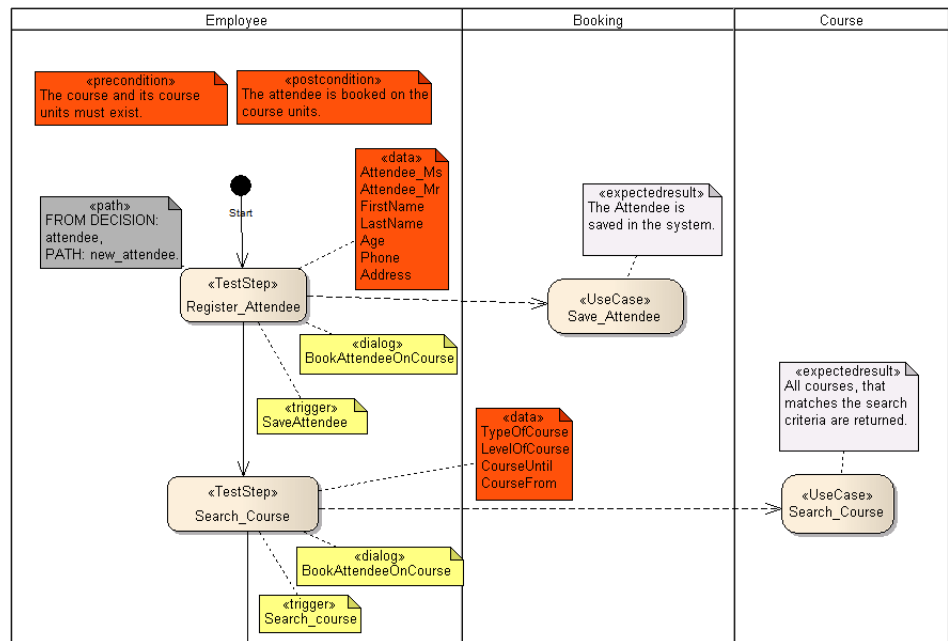


Figure 58: Logical test case created by the test selection, model analysis and model transformation algorithms

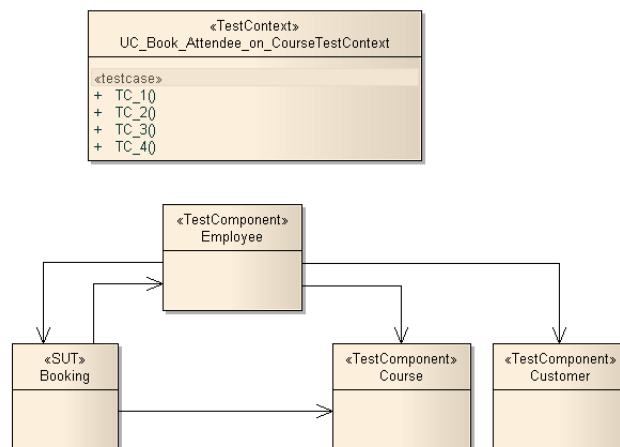


Figure 59: Test architecture package with the test context, several test component and one SUT class created by the model analysis and model transformation algorithms

traceability topic is strongly connected with the model coverage measurement and will be introduced in the next subsection.

#### 5.4.4 Model Coverage Measurement

After executing the test selection algorithm and the model transformation rules a basic version of the test model is created. Before the test model is extended with further information, the reached quality in terms of model coverage has to be measured. Model coverage in the current literature as shown in the surveys from Andrews et al. [AFGC03] or Mc Quillan and Power [MQP05] concentrate on the coverage reached by the test selection. In our case the coverage reached by the test selection algorithm together with model analysis and transformation algorithms is relevant. In Subsection 2.2.4, we have already introduced the definition of the holistic model coverage.

The holistic model coverage should be performed on the results of the mentioned algorithms. The goal is to identify in what extent the generated test model covers the analysis model. This can be done by using the traceability information between the single elements of the test model and the analysis model. We have already introduced the topic of traceability back in Subsection 2.5.3.

*Traceability for  
model coverage  
measurement*

The best source of the traceability information are the model transformation rules itself. In each rule the source and target model elements are specified. Knowing this the following question arises: Why do we have to measure the reached coverage rather than analyse the model transformation rules, since all source and target elements are known? First, there can be elements of the analysis model for which no model transformation rules were specified. The model transformation rules can be changed at any time during testing, which also influences the reached coverage. Finally, we distinguish between rules executed before and after the test selection.

Especially the later ones depend on the adherence of model relations from the analysis meta-model. If some model links are missing, the model transformation rules will not result in new elements of the test model. Let us consider the example from Figure 60. In the analysis model the use case is linked to the dialog, but the dialog is not linked to the logical data type model. After

*Importance of model  
relations*

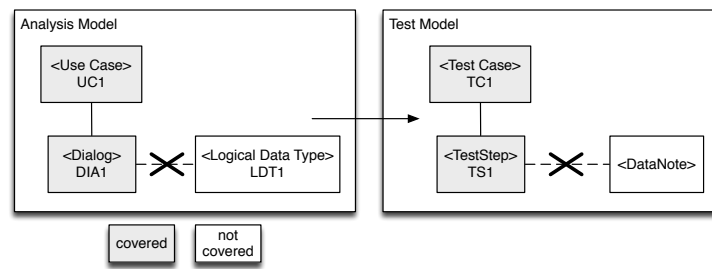


Figure 60: Coverage problem resulting from missing model linkage

executing the algorithms mentioned so far, a test model consisting among other of a test case with several steps is generated. Since the link between dialogs and the logical data type model is missing, no data note for a test case step is created. This way only the use case and dialog model elements are covered (depicted by gray background). This missing traceability information cannot be extracted only by analyzing the model transformation rules without executing them. That is why the traceability information of the model transformation process has to be made explicit and compared with the analysis model.

#### Trace meta-model

The main artefact used during the model coverage measurement process is the trace model. In Figure 61 the trace meta-model is shown. It consists of multiple trace links. Each trace link represents the sources (elements of the analysis model) and targets (elements of the test model) used during the model transformation process. Since our approach uses multiple source and multiple target transformations, the cardinality is always 1 to \*.

The trace model consists of several trace links. In the case the generation of the test model failed, the trace model is empty. The trace links consist of a collection of *source* and *target* references. We have chosen this structure, because of the relation with model transformation rules. The execution of one or more model transformation rule results in a trace link. Since an element of the test model can be transformed by using several elements of the analysis model, we specify the 1..\* cardinality between the trace link and source and target elements.

Each source and target element is described by an *id*, *name*, *modelID* (id of the model element in the analysis model), *modelName* and *modelType* (for example use case, dialog, etc.). Additionally

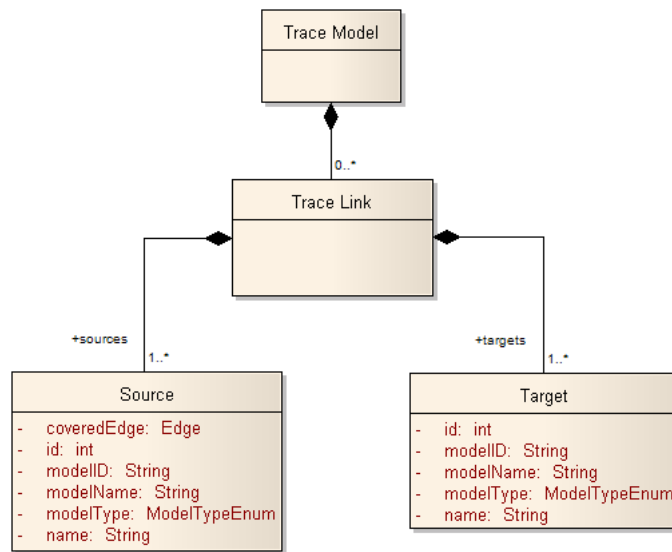


Figure 61: Trace meta-model

in Source the attribute *coveredEdge* is used. In the case of activity diagrams describing a use case, the outgoing edges with guards of decision nodes play an important role within the test model.

The model coverage measurement compares the *source* references from the trace model with the model elements of the analysis model by using the *modelID* attribute. Each model element for which no *source* reference could be found is automatically not covered. The relation between sources and targets in a *Trace Link* enables the identification and verification of covered elements. Besides the coverage measurement purpose, each element of the test model can be traced back to its source and this way its comprehensibility is improved.

The specification of the model coverage measurement process begins with the definition of the algebra operation (Chapter 4). The algebra operation relevant for the coverage measurement is named *cover*.

Operation *cover* according to the template from Section 4.6:

- Name = cover
- Goal = elements of the trace meta-model are analysed for their coverage of elements of the analysis meta-model
- Type = Multiple meta-model operation

- Input meta-model = analysis and trace meta-model
- Input set = all elements of the trace and analysis meta-model
- Output meta-model = analysis meta-model
- Output set = covered elements of the analysis meta-model
- Property = traceability
- Visualization = see Figure 62

The operation *transform* uses the analysis meta-model together with the trace meta-model as the input set. The output set is the subset of the analysis meta-model which is covered. The coverage is determined by the trace model which represents the traces between the test and analysis meta-model. Based on this abstract definition, we will now provide some exemplary algebra terms and the refinement of the *cover* operation as a concrete algorithm.

Exemplary  
meta-model algebra  
terms

Since the algebra operation described above uses the models resulting from the execution of other operations as *select*, *extract* and *transform*, we can specify algebra terms similar to the example introduced in Section 4.1.

$$\text{select}(m) \quad (5.2)$$

$$\text{extract}(\text{select}(m)) \quad (5.3)$$

$$\text{transform}(\text{select}(m) \cup \text{extract}(\text{select}(m))) \quad (5.4)$$

$$\text{cover}(\text{transform}(n), t) \quad (5.5)$$

where  $n, m \in AM$  ( $n$  and  $m$  are typed over the analysis meta-model) and  $t \in \text{Trace}$  ( $t$  is typed over the trace meta-model).

In 5.2 the *select* operation is defined. It is used as input in 5.3 for the *extract* algebra operation. Then, in 5.4 the selected test cases together with the extracted information is transformed into the test model with the *transform* operation. Finally, the operation *cover* uses the generated test model and the trace model to measure the model coverage in 5.5.

This simple algebra terms provide an additional specification of the input/output relation between algebra operation in our approach. Further, the order of algebra operations and therefore algorithms can be easily specified. Though this meta specification



Meta-mod  
surement

in the first place, the corresponding specification of algorithms is supported.

To measure the reached model coverage we define an algorithm, which is depicted in Figure 63. The activity diagram contains four swimlanes, which represent the different algorithms used in the overall approach. This is needed, because of the dependence of the coverage measurement on model transformation rules. We gain traceability information during the execution of the rules. The four steps beginning and ending with the execution of model transformation rules represent the algorithms mentioned in the last two subsections. Since the model transformations are executed before and after the test selection, we introduce a new artefact called trace model, which is generated after the execution of each model transformation execution.

In Figure 63 we abstract from the details of the general approach as manual model analysis, test annotation, test model extension, etc. We also use the analysis model as input for the model coverage measurement without modelling where and how it was created.

*Model coverage  
measurement  
algorithm*

The output of the steps executed at the beginning of the coverage measurement process from Figure 63 are the test model and a trace model. Both serve as input for measuring the coverage level. First the coverage of single model elements (like instances of use cases) is calculated. Afterwards, the coverage for certain model types (as use cases, dialogs, logical data types, etc.) is calculated. Then the coverage of the overall analysis model is calculated. The calculations for model types and the overall analysis model is done by averaging over the coverage calculated for single model elements.

Besides the calculation of percentage coverage for model elements, the identification of test cases and the coverage reached by them is important. Typical coverage reports in the field of requirements-based testing contain information like "for use case X the following test cases were created: TC<sub>1</sub>, TC<sub>2</sub>, TC<sub>3</sub>, etc.". This kind of visualization is also important in our model coverage measurement algorithm.

At the end all calculated information is visualized within a coverage report. This report contains all metrics for the different calculations in a hierarchical form. The purpose of this artefact is to support the test manager by checking the fulfillment of test





strategy by the automatic test generation. If certain goals are met, but result in too many test cases he can adjust the strategy and advise test designers to use other test selection criteria.

Within our measurement process, several metrics are used to calculate the reached coverage level. Based on the assumed structure of the analysis meta-model from Subsection 2.4.11 and the Goal-Question-Metric approach [BCR94], we have define the following goals for the model coverage measurement:

**Table 6:** Model coverage measurement goals

Goal dimension	Value
Object of study	Analyze the <b>analysis and trace model</b>
Purpose	for the purpose of <b>coverage measurement</b>
Quality focus	with respect to the <b>global, use case, dialog and logical data type model coverage</b>
Viewpoint	from the viewpoint of the test designer and test manager
Context	in the context of the test design phase, the purpose of <b>generating test models</b> from analysis models

In the goal definition from Table 6 we differentiate between the global coverage of the analysis model, the coverage of single use case (behavioural modelling viewpoint), dialog (interaction modelling viewpoint) and logical data type models (structure modelling viewpoint). Additionally the coverage of each model type (for example use case and dialog) can be measured.

#### Coverage metrics

Based on this goal definition, we define the several metrics for the mentioned elements of the analysis model. Each metrics is specified with a basic mathematical equation, which divides the covered elements by the overall number of elements. For this specification we use standard mathematical operators as addition  $a + b$ , division  $\frac{a}{b}$  and multiplication  $a * b$ . Additionally, we use the sum symbol  $\sum$  and the intersection symbol  $\cap$ . The following metrics are defined for the exemplary modelling approach used in this thesis:

$$\text{cov\_global}(\text{AM}) = \frac{\text{cov\_uc} + \text{cov\_dia} + \text{cov\_dt}}{3} \quad (5.6)$$

$$\text{cov\_uc}(\text{AM}) = \frac{\sum_{m \in \text{UC}} \text{cov}(m)}{|\text{UC}|} * 100 \quad (5.7)$$

$$\text{cov\_ad}(\text{uc} \in \text{UC}) = \frac{|\text{an} \in (\text{ad} \cap \text{path})|}{|\text{an}|} * 100 \quad (5.8)$$

$$\text{cov\_de}(\text{dia} \in \text{DIA}) = \frac{|\text{de} \in (\text{DIA} \cap \text{path})|}{|\text{de}|} * 100 \quad (5.9)$$

$$\text{cov\_da}(\text{dia} \in \text{DIA}) = \frac{|\text{da} \in (\text{DIA} \cap \text{path})|}{|\text{da}|} * 100 \quad (5.10)$$

$$\text{cov\_dia}(\text{dia} \in \text{DIA}) = \frac{\text{cov\_de} + \text{cov\_da}}{2} \quad (5.11)$$

$$\text{cov\_dt}(\text{dt} \in \text{DT}) = \frac{|\text{dt} \in (\text{DT} \cap \text{path})|}{|\text{dt}|} * 100 \quad (5.12)$$

The following abbreviations are used in our metrics definition:

- AM = analysis model
- UC = all use cases
- uc = use case
- ad = activity diagram
- an = action node
- de = dialog element
- da = dialog action
- dia = dialog
- dt = logical data type
- path = selected paths within the activity diagram

The first metric *cov\_global* describes the global coverage of the analysis model. We calculate it as the coverage sum of the coverage of different model types (like use case, dialog or data type model) divided by the overall number of model types (here three). For each type the metrics *cov\_uc*, *cov\_dia* and *cov\_dt* are defined.

The metric *cov\_ad* describes the coverage of the activity diagram, which refines each use case. This metric calculates the coverage of action nodes within the activity diagram. We define it as the number of action nodes within a path divided by the overall number of actions. Depending on the test selection criteria (see Subsection 5.4.1) two other metrics for edges and decision nodes could be also used here.

For the dialog coverage the metrics *cov\_de* (percentage of covered dialog elements) and *cov\_da* (percentage of covered dialog actions) which are used within *cov\_dia*. We define the coverage of dialogs as the mean value between the coverage of dialog elements and coverage of dialog actions.

Finally, the coverage of the logical data type model is defined with the *cov\_dt* metric. All logical data types referenced from the activity diagram through the dialog actions and dialog elements are defined as covered. The mentioned reference is used within the model analysis algorithm in Subsection 5.4.2.

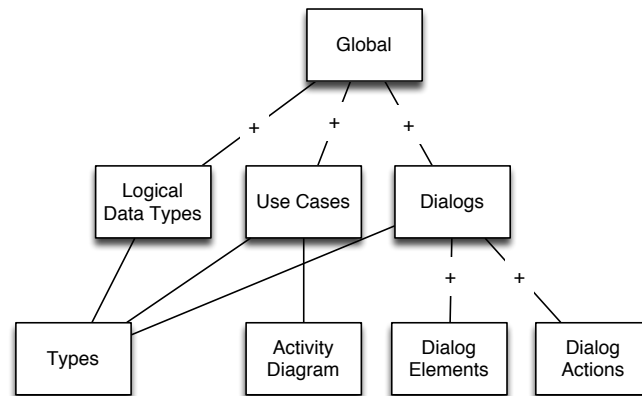


Figure 64: Hierarchy of the model coverage metrics

Our metric set has a hierarchical structure as shown in Figure 64. The main metric *cov\_global* uses the submetrics like *cov\_uc*, *cov\_dia* and *cov\_dt* to calculate the overall coverage. It uses the mentioned submetrics, but does not sum them. The submetrics also depend on other metrics. For example the coverage of use cases uses *cov\_ad* to calculate the coverage of each use case.

The visualization of the coverage measurement is the coverage report. It contains the calculations for all mentioned metrics. Within the report a final statement about the reached coverage

level is given. This statement is based on the *cov\_global* metric and is calculated as follows:

- $\text{cov\_global} \leq 33\%$  (low coverage)
- $34\% < \text{cov\_global} \leq 66\%$  (medium coverage)
- $67\% < \text{cov\_global} \leq 100\%$  (high coverage)

We have chosen this distribution of the *cov\_global* metric based on three observations:

1. The analysis model contains more model elements than the ones used within our algorithms
2. The test selection criteria which uses the annotation language uses only a part of the use case model for test selection
3. Dependent on the model complexity the 100% model coverage is not practicable in most cases because of the test explosion problem mentioned in Subsection 2.2.4

In order to provide an impression of the reached coverage of the analysis model, we decided to split the values between 0 and 100% in three groups with approximately each 34%. The first group (up to 34%) is called *low coverage*, because only small part of the analysis model (for example only some of the available use cases with no model links) were covered. A *medium coverage* (up to 66%) visualizes the situation where several inter-related elements of the analysis model were covered. *High coverage* (up to 100%) visualizes a coverage level which can be reached only by a strongly inter-related analysis model and exhaustive test selection criteria.

*Coverage levels*

An example of the coverage report for our running example from Subsection 2.4.4 is shown in Figure 65. The global coverage equals only 33% and therefore is low. The different coverage levels are visualized by the colors red (low), yellow (medium) and green (high). As shown in Figure 65 the color scheme is applied not only for the *cov\_global* metric, but also for other metrics like *cov\_dt*, *cov\_uc* and *cov\_dia*.

*Coverage report*

To visualize the coverage of use cases we show an example in Figure 66. There are five use cases within our running example. All use cases were covered by our *AllActions* test selection algorithm. For the use case *Search\_Customer* only one test case was generated. This coverage report visualizes the coverage measure-

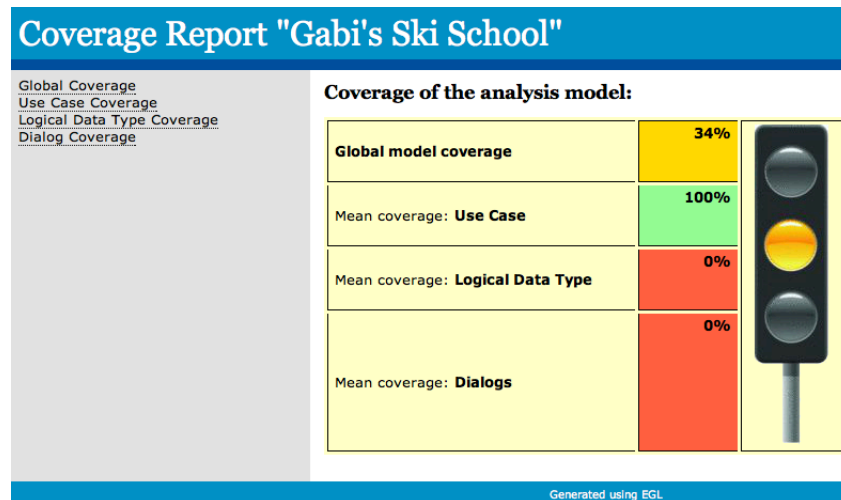


Figure 65: Example of the coverage report visualization (global coverage)

ment based on single action nodes of the use case' activity diagram as mentioned earlier.

As mentioned at the beginning of this subsection the difference between typical coverage measurement based on test selection criteria and our holistic approach lies in the usage of model relations. The coverage report supports the test manager by identifying the reasons for low coverage level (insufficient or acceptable). Besides the usage of weak test selection criteria, the following situations can lead to low overall coverage:

- Missing model relations between actor actions and dialog actions
- Missing model relations between dialog actions and dialog elements
- Missing model relations between dialog elements and logical data types

*Quality  
improvement of the  
analysis model*

The missing relations between model elements result in different coverage levels of the related model types. For example a use case can be covered with 100% but the according dialog only with 30%. This difference is caused by the missing relations between actor actions and dialog actions. The missing relations also impact the automated model analysis algorithm (see Subsection 5.4.2). As in there, also here the degree of model rela-

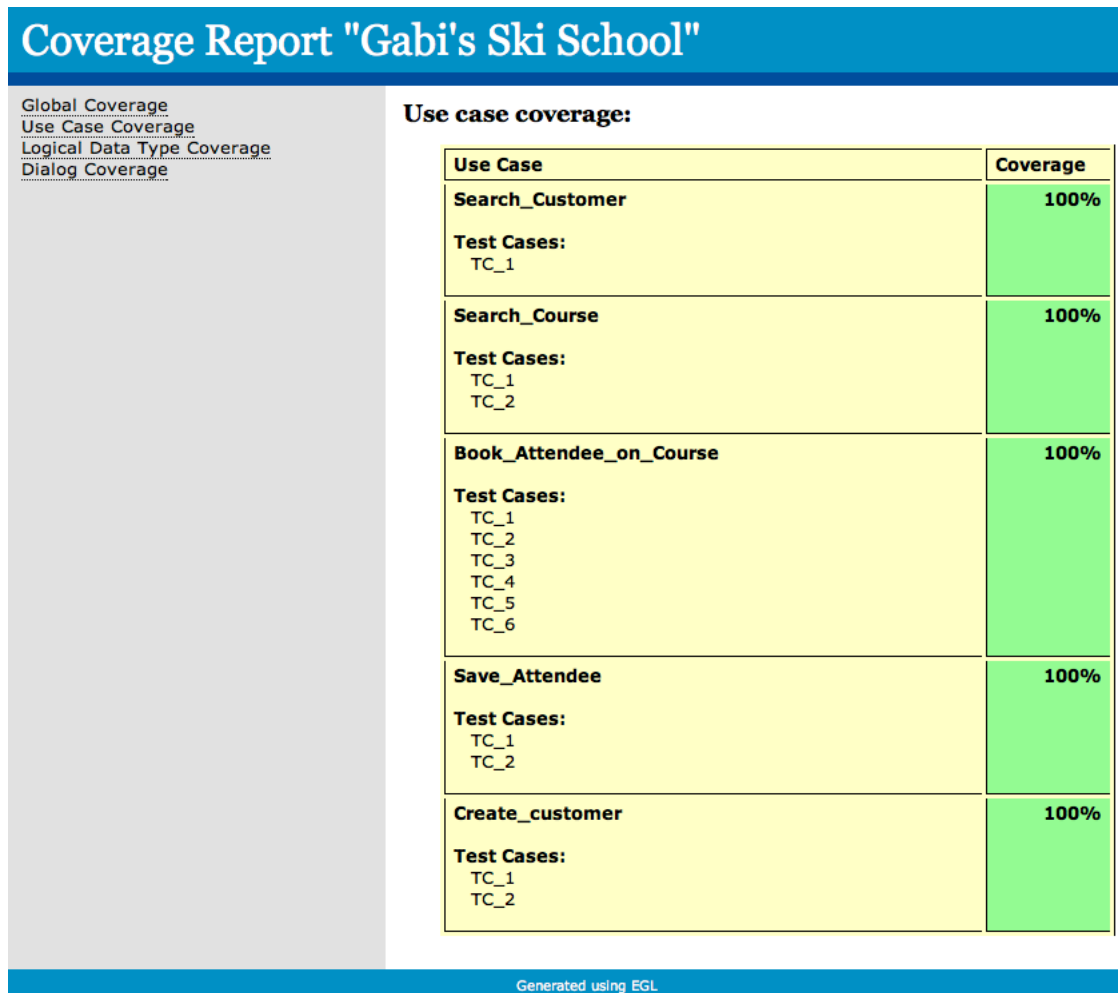


Figure 66: Example of the coverage report visualization (use case coverage)

tions transcribed by business analysts with respect to the analysis meta-model is important. Without empirical evidence at this points, we state that the following correlation holds for the holistic model coverage measurement:

$$\text{cov\_global(AM)} \sim \text{rel} \quad (5.13)$$

*Correlation with  
model relations*

Similar to the correlation introduced in Subsection 5.4.2, the *rel* variable symbolizes level of model relations instantiated according to the analysis meta-model by business analysts. In particular it is the degree of transcription of model relations at the model instance level. The *cov\_global* stands for the global coverage metric applied on the analysis model (AM). If the number of model relations is high, then the global model coverage grows. Within this equation, we omit the obvious correlation of the *cov\_global* to the other coverage metrics. The directly proportional correlation with the model relations is important, because on the impact on the model coverage and thus on the usage of the holistic view in model-based testing.

The coverage report visualizes the reached coverage level. In case of low coverage the test manager can map the coverage differences between model types with the missing relations in the analysis model. His role is to advise the business analysis team to incorporate the missing model relations. This way they improve the quality of the analysis model and a high quality test model can be generated.

The trace model is the central artefact used during the model coverage measurement process. The calculation of all introduced metrics is based on the information stored within the trace model and the analysis model. Besides the coverage measurement the trace model can be also used for impact analysis.

*Impact analysis  
reference*

Impact analysis deals with the identification of changes within a model and their impact on the related model elements. In our context the impact analysis should be performed automatically on the analysis and test model. Changes within the analysis model impact changes in the test model. Through the explicit traceability information stored in the trace model, we can automate great parts of the impact analysis process.



As the topic of impact analysis would require a separate phd thesis, we only refer to the approach from Farooq [Far10]. The author transforms a business process model described with BPMN to a test model described with UTP. This way the transformation process is very similar to ours, since we also use UTP as the test model representation. During this transformation process a trace model is stored. This model is used to identify the regressions test suites which have to be executed when the BPMN model has changed. To identify the changes within the BPMN model Farooq uses impact analysis techniques which compare the baseline and a delta BPMN model. Then the trace model is used to identify the affected parts of the UTP model. The impact analysis approach from [Far10] can be integrated into the model-based test specification process presented in this chapter by replacing the BPMN model with our analysis model. This integration can be implemented in the future and is not part of this thesis.

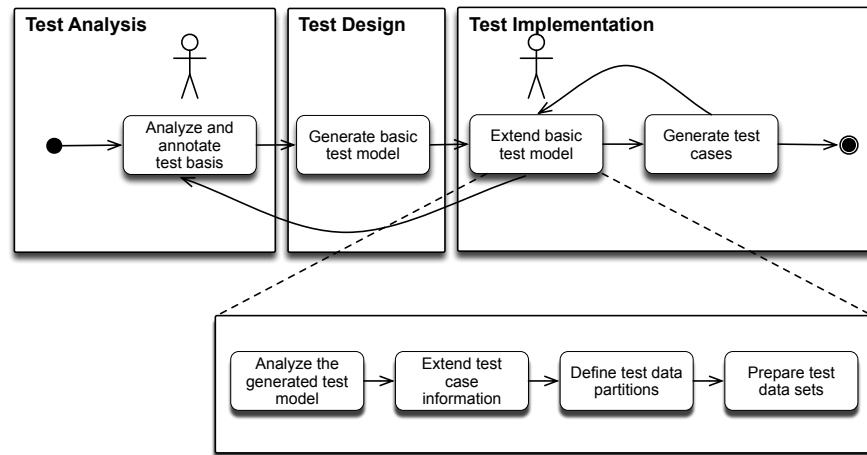
Our model coverage measurement approach incorporates the holistic view used during the transformation process. It enables the test manager the calibration of the test strategy and gives important hints for improving the quality of the analysis and test model. Since the coverage measurement is strongly coupled with the transformation process, the following problem arises: After the test model is automatically generated it is manually extended by the test designer with test data and expected results. Within this extension process some elements of the test model can be deleted or new elements be added. This change influences the coverage level. To recalculate the model coverage, the according changes within the trace model have to be performed. This task is done manually by the test designer. In the future version of our approach this task can be automated by providing impact analysis algorithms which automatically extend the trace model and reexecute the coverage measurement process.

*Changes within the test model*

## 5.5 STEP 3. EXTEND THE BASIC TEST MODEL

After applying several model transformation rules and selecting test cases from the analysis model, a basic test model is created. This model is described with a customized version of the UML

testing profile. The difference between the official OMG version [Obj07b] and the one used here was described in Subsection 2.3.



**Figure 67:** Refinement of the third step within the model-based test specification approach

The goal of the extension process is to guarantee the test independency needed in model-based testing (see Section 1.1) and to provide high-quality of the test model especially in terms of completeness. The manual model analysis from Section 5.3.1 has a similar goal related to the analysis model.

#### 5.5.1 Basic vs. extended test model

Prior to the introduction of the extension process, we first define the basic and extended test model.

**BASIC TEST MODEL** is the result of the execution of test selection, model analysis and model transformation algorithms. It contains of all elements belonging to the three UTP concepts: test architecture, test behaviour and test data.

**EXTENDED TEST MODEL** is the result of the manual extension process. It extends the basic test model with additional data partitions and precise pre- and postconditions definitions. Further, additional test cases can be added or the automatically generated test cases removed after a manual review process.

The test model is called basic, because several elements have to be extended or substantiated. Especially the logical test cases as a main part of the test behaviour concept have to be changed. The following list gives an overview on the logical test case elements which have to be changed in this step of the model-based test specification process:

- ConditionNote
- DataNote
- DialogNote
- TriggerNote

The test designer may also reconsider to delete some of the automatically generated test cases. For example the test selection algorithm has selected too many test cases, but a weaker test selection criteria selects too few. Another case is the business relevance of the generated test cases. For example the selected use case scenario is not critical and rarely used in the SUT. The business relevance can be incorporated within the annotation process from Subsection 5.3.2. This way risk-based testing is applied and omits the manual deletion of not needed test cases.

#### 5.5.2 Manual extension process

In order to systematically perform the extension process, we define the following steps:

1. Analyze the generated test model
  - a) Delete not relevant test cases
  - b) Check each test case missing information
  - c) If needed correct the system model and regenerate the test model
2. Extend test case information
  - a) Extend ConditionNote(s)
  - b) Extend DataNote(s)
  - c) Extend DialogNote(s) and TriggerNote(s)
3. Define the data partitions for each data pool
4. Prepare test data sets (external source)

Since each of the mentioned steps has to be performed by test designers, we provide a brief description for each of them in the next paragraphs.

#### *Delete not relevant test cases*

The automatic test selection can result in several test cases within the test model. For example a selection based on a medium complex activity diagram with the *AllPathsOneLoop* algorithm can result in 15-30 test cases. Some of the generated test cases can be not relevant for the test execution. This can be the case if the generated set of test cases is too large or "exotic" test cases<sup>7</sup> are included. Since the test designer can edit the generated test model directly in a modeling tool, he simply deletes the activity diagrams representing the not relevant test cases.

#### *Check each test case missing information*

##### *Completeness checks*

Besides the deletion step, the generated test cases can reveal missing information within the analysis model. Missing information can be identified by looking at the different UML notes of the test case activity diagram. If the pre-/postcondition, data, dialog and trigger notes are empty then this information was missing in the analysis model. In Figure 68 we provide a cut-out of the logical test case within the basic test model. As mentioned the data note is empty. A good practice is to analyze the coverage report for the coverage of use cases, dialogs and data models separately. If the coverage of those three model types strongly differs then relations between models are missing. The easiest way is to correct the analysis model and regenerate the test model. Other possibility is to manually extend the test model.

#### *Extend ConditionNote(s)*

The pre- and postcondition of use cases are often not sufficient for testing. Bad examples are post-conditions like "*The attendee is booked on the course units*" as shown in Figure 68. What exactly does "is booked" mean? In order to decide if a test case has

<sup>7</sup> We define "exotic" test cases as not relevant for the business from the customer point of view.

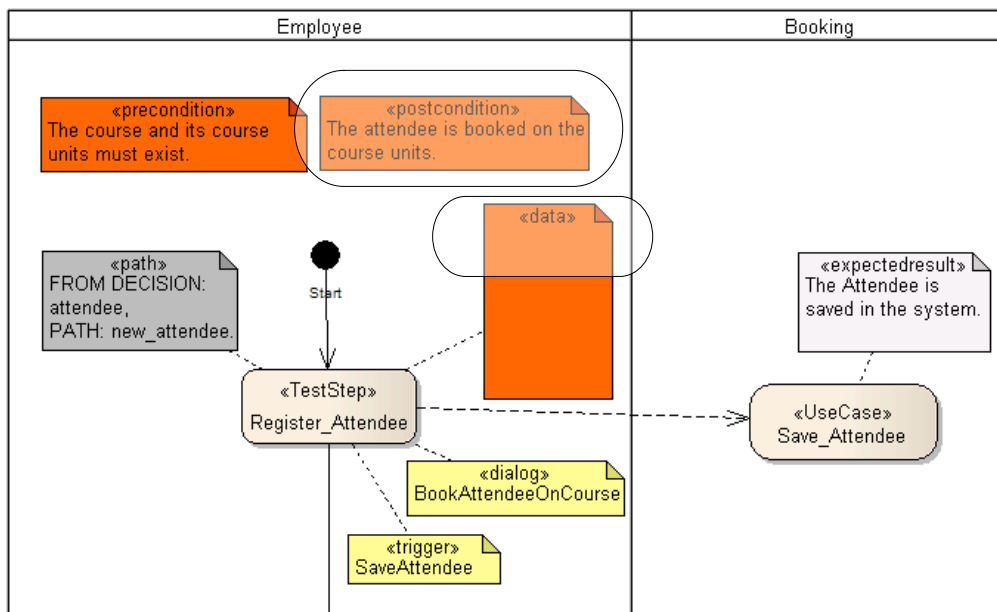


Figure 68: Example of the missing information in a logical test case within the basic test model

passed or failed during the test execution more detailed description is needed. Detailed information ensures that the condition is reachable, observable and analyzable. In the case of the mentioned example information about the shown confirmation window or new database entries is needed. Since this detail level is mostly needed only for testing, the extension or refinement step has to be done manually by test designers.

#### *Extend Data/Note(s)*

The data notes are automatically created during the generation step. Each note consists of textual names which reference to data types used as input for a given test step. For example a test case step called *searchCourse* needs the *TypeOfCourse*, *LevelOfCourse*, *CourseUntil* and *CourseFrom* data types as input. The test designer has to review these automatically generated proposals. If further data types like *NoCourseUnits* or *MinAttendeeGroup* have to be used, then the data note together with the according data pool has to be extended. The opposite action has to be performed if too many data types were automatically generated.

*Extend DialogNote(s) and TriggerNote(s)*

Similar to the data notes two other types of notes (dialog and trigger note) are generated automatically. Beside the case where no dialog and therefore no trigger is needed, the test designer can extend the mentioned notes. For example if more than one dialog is used in a test case step, then the dialog note has to be extended. The according trigger note which references the dialog action used to trigger the test case step has also to be extended, because different dialog actions (on more than one dialog) are used.

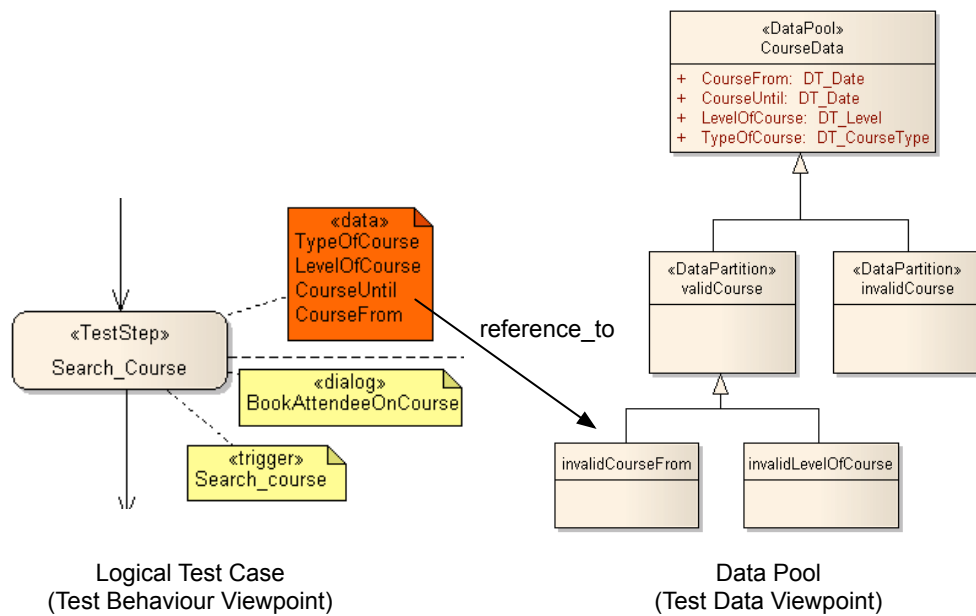
*Define the data partitions for each data pool*

As mentioned earlier the test model consists of one or more data pools for each use case being tested. In a test case step the single attributes of data pools are used within the data note. Each data note references to a certain data partition of a data pool through a so-called tagged value. Since this information cannot be derived from the analysis model, the test designer has to create it manually. First, the test designer has to define data partitions for each data pool. By default each data pool has a valid and invalid data partition. For example the partitions *validCourse* and *invalidCourse* of the data pool *Booking*. Following the category-partition method the *invalidCourse* can be further decomposed in *invalidCourseFrom*, *invalidLevelOfCourse*, etc.

The decomposition itself is useless, because the data partitions have to be used within test cases. The important task is to establish a link between a test case step and the data partition used within it. Technically it can be done within a modeling tool by using the tagged values within the data notes of a test case step. The tagged value should reference to the data partition used in a step. We will explain the purpose of this linkage in the next step description.

*Prepare test data sets (external source)*

Through the definition of the data pools and data partitions the test designer defined the logical structure of the test data. The next task is to concretize the data partitions with concrete data sets. For each data partitions a set of concrete values has to be



**Figure 69:** Example of the linkage between data notes in a test case and data partitions in the data pool

created. For example *invalidCourse* with values like *13.12.2010*, *18.12.2020*, *Advanced*, *Snowboard* as shown in Figure 69. The test designer can prepare more than one set of test data. A good practice is to use the boundary-value analysis from [SL05] for test data derivation here.

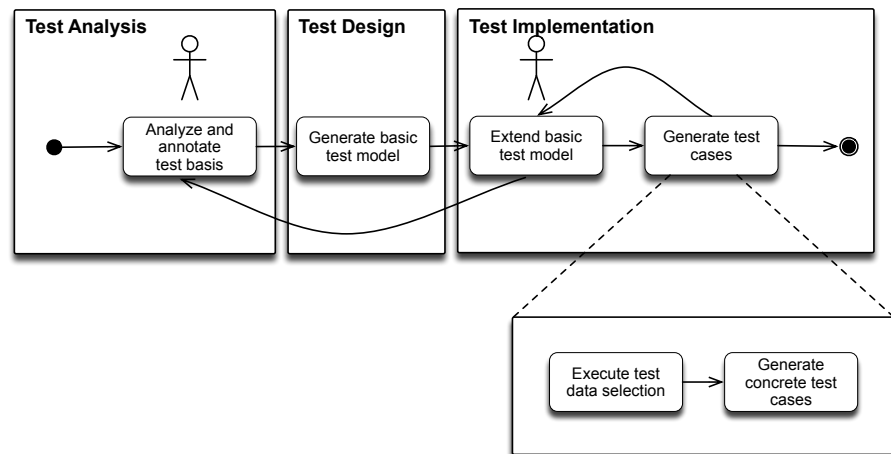
The definition of concrete test data sets can be performed at the model level by creating UML object diagrams. Since the amount of test data sets can be very high, we encourage the use of an external source like database or xml files to store the test data sets. It is very important to use the same logical structure as defined in the test data package.

During the test data preparation the same problem as within the model coverage measurement arises. The test model is being changed which influences the coverage level. In the current version of the model-based test specification process the model coverage measurement strongly depends on the execution of model transformation rules. This execution results in a basic test model which does not incorporate the changes made in the extension step. Those changes have to be manually appended into the trace model instance on which the coverage measurement depends.

If all needed test data sets are prepared, the generation of concrete test cases can be triggered. This step will be described in the next section.

## 5.6 STEP 4. GENERATE CONCRETE TEST CASES

The automatic generation of concrete test cases is the last step of the model-based test specification process. Those test cases are used by testers for manual or automatic test execution. The algorithms and manual tasks executed in the last three steps of the process lead to high-quality test cases, which fulfill the quality attributes mentioned in Chapter 3.



**Figure 70:** Refinement of the fourth step within the model-based test specification approach

The distinction between a test model and concrete test cases is motivated by the following requirements:

- REQ1 The logical test cases within the test model do not contain concrete test data, which is needed for the execution
- REQ2 The execution of concrete test cases depends on the platform of the used SUT
- REQ3 The management of test cases and their execution in projects is done within professional test management tools



The first two requirements are motivated from the problem statement (Section 1.1) and the discussion about different model types in the MDA (Section 2.5). The third requirement is based on the observations of large-scale industry projects within this thesis. To fulfill the first requirement, we use data combination algorithms. For the second and third requirement, we use the concept of platform-independent (PIM) and platform-specific models (PSM) known from MDA [Obj03]. Here, especially the model to text transformations are used.

#### 5.6.1 Excursion: Constraints in test data

The automatically generated and manually extended test model contains logical test cases and the logical structure of the test data. Concrete data sets are created manually or automatically by using test data generators [RBGW10] according to their logical structure within the test model. To create concrete test cases the logical test cases have to be combined with the concrete data sets.

In large-scale projects, the combination of concrete data sets has to incorporate constraint definitions. The problem here is that certain concrete data sets can not be combined with each other or the combination has to include only certain data partitions. The following cases should be regarded within the constraint definition:

*Combination of test data*

- *DataSet1* have to be combined with *DataSet2*
- *DataSet1* can not be combined with *DataSet2*
- *InvalidPartition1* can not be combined with *InvalidPartition2*
- *ValidPartition1* have to be combined only with *InvalidPartition2*

The need for the incorporation of data set constraints is motivated by the complexity of test data for testing information systems. Since such systems use complex, object-oriented data models this complexity is also given in the test data model. Within other system types like the embedded systems this complexity is not given since the data model structure is typically simpler.

Based on the different cases introduced above, we define the following logical operators, which are used to specify constraints between test data:

- AND
- NOT

The constraint definition with the mentioned operators has to be defined at the model and instance level. If constraints exist between data partitions, then the constraint definition takes place within the test model. For example the combination of several data partitions in the test data viewpoint. The other case are constraints between concrete data sets. For example the combination of concrete data sets for data partitions. Here the constraint definition has to be placed outside the model. Based on our observations from the industry research project conducted within this thesis, test data in large-scale projects is typically managed by using database systems. Those systems enable the definition of constraints (for example by using Structured Query Language [SQLo8]) directly within the database.

To define the constraints at the model level the Object Constraint Language [Obj06b] is used. This language defines the AND and NOT operators. At the instance level the SQL [SQLo8] defines the mentioned operators. Both languages support the definition of complex constraints.

#### *Problem complexity*

As mentioned earlier, concrete test data can be created manually or automatically generated with test data generators. Especially in the automated case the specification of constraints and its usage to generate test data according to the logical test data model is a non trivial problem. Mario Winter analyzed in [Win99] the generation of (test data) object constellations from UML class diagrams refined with OCL constraints. The author defined a problem called "Generierung von Objektkonstellationen (GOK)" in [Win99, p.184] and provided a proof for its NP-hardness. The GOK problem emerges, when concrete test data sets have to be automatically generated from a test model with constraint definitions. Winter references well-known test data generation approaches as [Bei95], which can be adapted for automated test data generation from the UTP test model in this thesis.

The specification of test data constraints is important for testing in large-scale project. However, this topic is not strictly related to the research problem of the missing holistic view. The technique

and languages mentioned in this subsection are useful to adopt the solution in a concrete project scenario. We do not further detail this topic in this thesis and reference it in the future work.

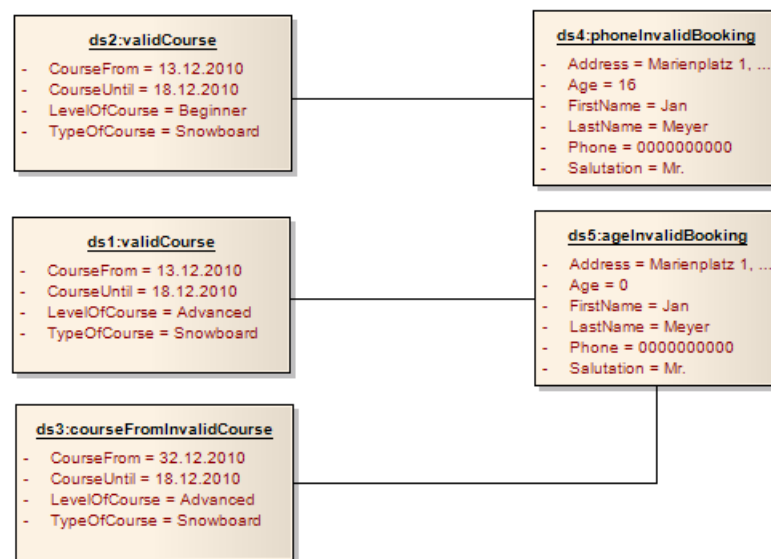
### 5.6.2 Test Data Selection

During the generation of the basic test model the selection of logical test cases was conducted. To generate concrete test cases another selection algorithm has to be defined. The goal of this algorithm is to combine concrete data sets according to a certain data coverage criteria (see Subsection 2.2.4). Similar to the selection of logical test cases with transition-based coverage criteria, the selection of test data is not restricted to single data coverage criteria.

In Section 2.2.4 we introduced several types of data coverage criteria. The simplest solution is to sequentially combine each data set created for the data partitions defined in the test model. In Figure 71 we provide a simple example. The concrete data sets *ds1* - *ds5* are combined according to the *SimpleCombination* test selection criteria. The data set pairs *ds2,ds4* and *ds1,ds5* are combined sequentially. Since no new concrete data set for the partition *ageInvalidBooking* is given, the data set *ds3* is combined with the last data set used for partition *ageInvalidBooking*. This way *ds3* is combined with *ds5*.

*Simple combination  
of test data sets*

To fulfill the *SimpleCombination* data coverage criteria, we define the according algorithm here. Since we do not define a meta-model for the output of the test data selection, no algebra operation (see meta-model algebra in Chapter 4) is specified here. Figure 72 depicts the algorithm as an activity diagram. In the first step the definition of all data partitions is read from the test model. Next the concrete data sets are read from an external source (for example a database). In the third step all test cases are read from the test model. The algorithm iterates through each found test case. For each test case the referenced data partitions are read. Then for all data partitions the according concrete data sets are read sequentially. If for one of the data partitions no more data sets can be found, then the last read data set is taken. This way for each combination of data sets read for the data partitions a concrete test case is generated.



**Figure 71:** Combination example of data sets according to the *SimpleCombination* test selection criteria

Since the application of the *SimpleCombination* algorithm omits several combinations, which could lead to the detection of further faults, stronger data coverage criteria like *N-wise coverage* or even *All-combinations* can be used here alternatively.

#### Test data management

In the last section we have mentioned that concrete test data sets should be stored outside the test model. This is motivated by the amount and complexity of test data sets in software engineering projects. External sources like databases provide advanced support for managing test data.

In order to use an external data source, the following requirements have to be fulfilled:

- REQ1 **Mapping between test model and external source** - the logical structure of test data is defined in the test model by data pools and data partitions. The structure of the data in the external source should be defined respectively.
- REQ2 **Constraint definition** - the external data source should support the definition of constraints as mentioned in Subsection 5.6.1.
- REQ3 **Consistency of data sets** - the concrete test data sets should be consistent according to the constraint definition and the logical test data structure defined in the test model.

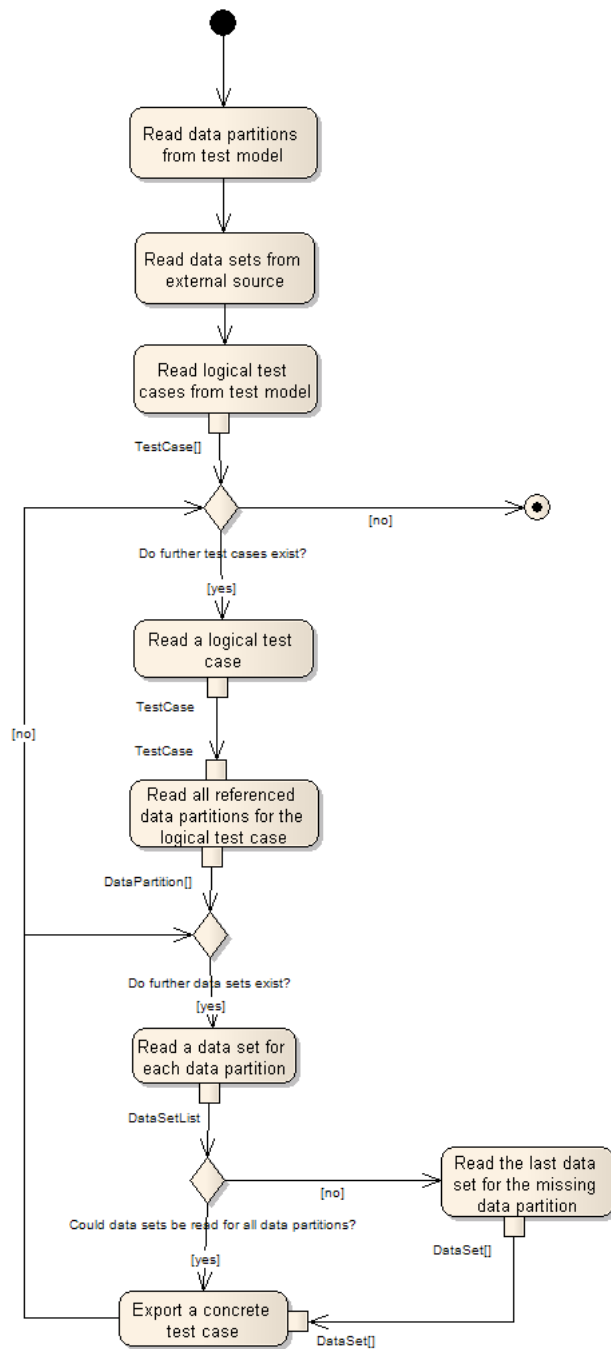


Figure 72: Algorithm fulfilling the *SimpleCombination* test selection criteria

The concrete test cases have the same structure as the logical test cases which were described in Subsection 2.1.1. The only difference lies in the description of test case steps. This description contains the concrete data sets which were combined using the algorithm from the last subsection. Listing 5.2 shows an excerpt of a concrete test case in the XML file format, which is based on the running example of this thesis. The important difference to a logical test case from the test model is the *input* field. In this example the input data is filled by concrete values. For example the *TypeOfCourse* equals *Snowboard*.

Listing 5.2: Example of a concrete test case

```
...
<teststep>
  <no>2</no>
  <name>Search_Course</name>
  <stereotype>TestStep</stereotype>
  <datapartition>validCourse</datapartition>
  <description>The employee enters the course data
    and uses the use case Search_Course for
    finding a suitable course.</description>
  <dialog>BookAttendeeOnCourse</dialog>
  <input>TypeOfCourse:Snowboard,
    LevelOfCourse:Beginner, CourseFrom:10
    .10.2010, CourseUntil:16.10.2010</input>
  <trigger>SearchCourse</trigger>
  <expectedResult></expectedResult>
</teststep>
....
```

In the last step of the algorithm introduced in Figure 72 concrete test cases are exported to predefined file formats. The example of a concrete test case provided above was already depicted in the XML file format. The transformation of concrete test cases into a platform-specific file format will be described in the next subsection.

### 5.6.3 Platform-specific test case generation

Following the idea of PIM and PSM known from the MDA [Obj03], our approach enables the generation of platform-specific test cases from the test model. For that model to text (M2T) transfor-

mations are used. We distinguish between the following targets of the M2T:

- Textual description for manual test execution (like Microsoft Excel<sup>8</sup>)
- Import format for test management tools (like XML<sup>9</sup>)
- Test script language (like xUnit<sup>10</sup>)

Each target type is fully customizable for the project needs, like the tools used for managing or executing test cases. The target types are defined using *templates* which can be interchanged each time concrete test cases are generated. Since the specification of concrete M2T rules would not provide further knowledge about the holistic model-based testing approach, we do not provide any concrete examples here.

In the case of transforming test cases to test scripts a problem of missing information arises. The abstraction level of the analysis model and the test model is high. Both models do not include information about the design of the source code. This way, a mapping between use case actions or test case steps and methods within the source code is missing. A similar mapping between the dialog model and its implementation within the source code is missing. In the case of dialogs, also the information about the state model can be included. As introduced in Subsection 3.2.6, there exist several approaches as [BBWo6, MBNo3, QJo9, XM08] for modelling dialog states and events on a very low-level. This kind of modelling allows the precise specification of the navigation through dialogs (for example order in which dialog elements have to be filled with test data) and the expected results. This information is needed for the automatic execution of test scripts.

*Additional  
information for  
PIM/PSM  
transformation*

To enable the automatic execution the missing mapping has to be established. We achieve it by introducing the concept of test adapters known from Utting and Legeard [UL07]. A test adapters implements the mapping between the source code and the test model. The implementation is done by specifying M2T rules and executing them in a model transformation framework. For the specification and execution the Epsilon Generation Language

<sup>8</sup> <http://office.microsoft.com/en-us/excel/>

<sup>9</sup> <http://www.w3.org/XML/>

<sup>10</sup> <http://www.junit.org/>

**Table 7:** High-level mapping between the test model and source code within a test adapter

Test model element	Source code element
TestStep, TriggerNote	Method name
DialogNote	Dialog name
Data Pool	Variables or Dialog element

(EGL) can be used. We have already introduced EGL and M2T in Section 2.5 about model transformations.

In Table 7 we propose a very high-level mapping between the elements of the test model and source code elements. The single steps of logical test cases are mapped to methods within the source code, which have to be executed. The *TriggerNote* is also mapped to a concrete source code method. The *DialogNote* is mapped to the concrete dialog implementation. Finally, the elements of the data pool from the test model is mapped to several variable within the source code. Additionally those variables can be used as dialog elements, which are also mapped to the data pool elements.

## 5.7 SUMMARY

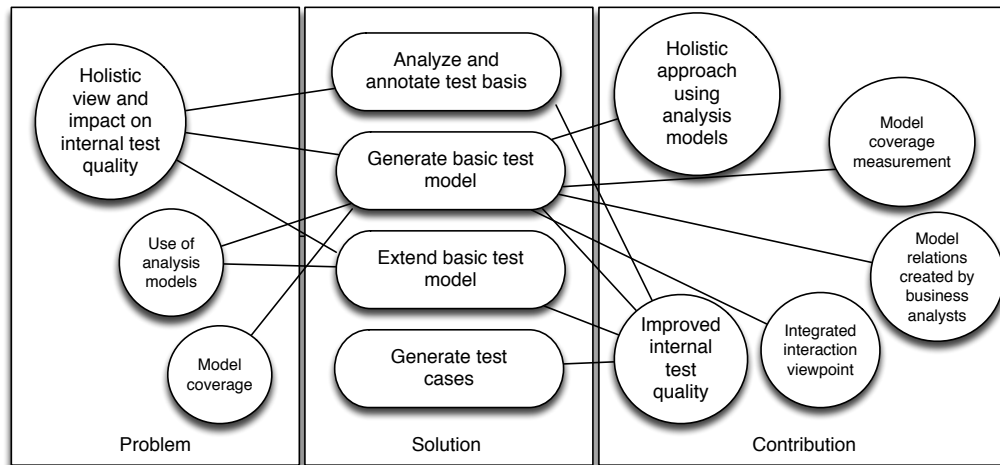
The output of the model-based test specification process described in this chapter are concrete test cases. Those test cases are *complete* as they contain information from all three views (structure, behaviour and interaction) of the analysis model. Through the manual extension process additional information which is independent from the analysis model is appended to the test cases. This way complete test cases ready for test execution are created.

Based on the trace links within concrete test cases, they are clearly *traceable* to test model. By using the trace model all elements of the test model are traceable to the analysis model. Further the trace model is used to measure the holistic coverage of the analysis model.



They are also *analysable* since the automatic generation process uses a predefined template for logical and concrete test cases. Especially the analysability of architectural and data aspects is supported by the relation between the logical test cases and the test architecture and test data viewpoint within the test model.

Since the logical test cases within the test model are defined by a graphical modeling notation and are refined by textual descriptions, the test cases within our approach are *understandable*.



**Figure 73:** Mapping of the research problem, solution and contribution of the phd thesis

In Figure 73 we introduce a mapping between the main research problems, the solution introduced in this chapter and the main contribution points of this phd thesis. The problem of the missing holistic view and its impact on the internal test quality is being solved in the first three steps of the model-based test specification process. The aspect of using analysis models for test purposes and the needed information independency is supported by the test model generation and extension steps. The measurement of the holistic coverage is done in step two of our process.

The main contribution of this work is the improvement of the internal test quality. This contribution is implemented in all steps of our process. Especially the completeness aspect. We improve the quality of test cases by manually improving the quality of the analysis model in the first step. Then, we collect several context-related information from this model by using the automated

model analysis algorithm in step two. We append additional and independent test information to the test cases in step three. By combining several concrete test data sets in the last step we improve the final completeness of test cases.

By combining algorithms for test selection, model analysis and model transformation we introduce a novel holistic approach for model-based system testing. Our method uses model relations within the analysis model which were created by business analysts. Since this model also consists of the interaction modelling viewpoint (especially GUI models), we incorporate them within our generation approach.

In the next chapter, we introduce two experiments which evaluate our approach according to the phd hypothesis and underpin the contribution points from Chapter 1.

---

## EVALUATION

---

In the last chapters, we introduced the theoretical concept of the meta-model algebra. We then presented the model-based test specification approach as a holistic solution for model-based system testing. All automated steps within this approach were specified with the meta-model algebra. Within the chapter, we introduce an experiment as a proof-of-concept for the mentioned approach in order to test the hypothesis of this phd thesis.

First, we briefly introduce the evaluation plan according to the experimental software engineering approach from Wohlin et al. [WRH<sup>+</sup>99]. We then briefly describe the tool support (Test Model Generator and Test Case Generator) implemented as a proof-of-concept for the holistic model-based testing approach. We use the Test Model Generator within an experiment based on a the thesis running example (see Subsection 2.4.4). At the end, we analyze the threats to validity of our experiment and discuss the results in relation to the thesis contribution.

### CONTENTS

6.1	Evaluation planning . . . . .	197
6.2	Tool support . . . . .	204
6.3	Experiment "Gabi's Ski School" . . . . .	208
6.4	Discussion of the results . . . . .	220
6.5	Summary . . . . .	224

---

### 6.1 EVALUATION PLANNING

In this section, we introduce the evaluation plan for the approach presented in this thesis. For this, we first define the evaluation goals according to the thesis hypothesis. We then introduce the

experiment design focusing on the used metrics. Afterwards we describe the setting of our experiment. Finally, we define the null and alternative hypotheses which are investigated in our experiments.

#### 6.1.1 Evaluation goals

The main goal of the evaluation is to accept or reject the phd hypothesis from Section 1.1 in terms of:

- Empirical evidence for the improvement of the internal test quality when applying the holistic view on analysis models in model-based system testing (see contribution points 1,3,4,5 from Section 1.2)
- Empirical evidence for the improvement of the model coverage of analysis models when applying the holistic view in model-based system testing (see contribution points 2,3,4,5 from Section 1.2)
- Empirical evidence for the effort optimization with automated use of analysis models as basis for test model generation (see contribution point 2 from Section 1.2)

and thus, show whether we tackled the research problems of a missing holistic view while using analysis models in model-based testing.

Table 8: Evaluation goal 1

Goal dimension	Value
Object of study	Analyze the <b>test model</b>
Purpose	for the purpose of <b>evaluation</b>
Quality focus	with respect to the <b>internal test quality</b>
Viewpoint	from the viewpoint of the test designer
Context	in the context of the test design phase, the purpose of <b>generating test models</b> from analysis models

Using the GQM template for goal definition from [BCR94] we define the goals mentioned before in a more detailed manner in Table 8, 8 and 10.

Table 9: Evaluation goal 2

Goal dimension	Value
Object of study	Analyze the <b>test model</b>
Purpose	for the purpose of <b>evaluation</b>
Quality focus	with respect to the <b>reached model coverage</b>
Viewpoint	from the viewpoint of the test designer
Context	in the context of the test design phase, the purpose of <b>generating test models</b> from analysis models

Table 10: Evaluation goal 3

Goal dimension	Value
Object of study	Analyze the <b>test model</b>
Purpose	for the purpose of <b>evaluation</b>
Quality focus	with respect to the <b>time effort</b>
Viewpoint	from the viewpoint of the test manager
Context	in the context of the test design phase, the purpose of <b>generating test models</b> from analysis models

The evaluation goals can be determined by performing one or more case studies or experiments. According to Wohlin et al. [WRH<sup>+</sup>99], a case study is performed within a real-life project during a predefined time slot. The experiment is a study performed in a controlled environment, in example we refer to hypothesis testing here [WRH<sup>+</sup>99]. Within this thesis, we perform an experiment rather than case studies, since we aim to influence certain parameters (as the model relations of the analysis model or test selection criteria) and observe the reaction in a controlled environment. The controlled environment consists of software and hardware artefacts which do not change during the experiment. Further, we assume that our input model is influenced only by one person within predefined parameters. We also assume a situation where a new software product is built based on the system specification (for example analysis model together with a design model). This way we minimize the possible side effects which could occur. Compared to a case study, where several project parameters (software, hardware, persons, specification changes, etc.) can dynamically change, we can minimize this bias by performing an experiment.

#### 6.1.2 Experiment design

We use the following metrics during the experiments' execution:

- number of generated logical test cases (LTC)
- time effort needed for the automatic generation (in minutes)
- reached global, use case, dialog and logical data type model coverage (in percentage according to the algorithm from Subsection 5.4.4)
- logical test case completeness (number of test case attributes filled with data)
- logical test case traceability (number of test model elements not traceable to the analysis model)
- logical test case understandability (subjective judgement by test experts on a scale 0-5 according to the method from Pennington in [Pen87])

- logical test case analysability (number of logical test cases without relation to test architecture or test data)

The number of generated LTC is needed to compare the different test selection algorithms from Subsection 5.4.1. The time effort is used to measure the effort improvement compared to a) the manual test design and b) the MBT scenario "manual modelling" from Pretschner and Philipps [PP05]. For this, we use the mean value of 2,67 person hours per one test case for manual test design. This value is based on the case study introduced in [RBGW10, p.362]. Based on several case studies as [UL07, PPW<sup>+</sup>05, RBGW10, HNo4, DNSV<sup>+</sup>08, Leg08, CSH03], we provide boundary values for time effort of the modelling, review and extension of the test modelling tasks in the mentioned MBT scenario.

The reached model coverage is needed to observe the effects of using the holistic view on the analysis model by measuring the coverage of the different modelling viewpoints. The four metrics for internal test quality attributes further reflect the application of the holistic view during test model generation. Since the measurement of test case understandability is a subjective measure we customize the method introduced by Pennington in [Pen87] by asking three independent test experts the following questions for a particular LTC generated with and without the holistic view:

1. Function: Can you describe the overall functionality of the LTC?
2. Data Flow: Can you describe where does the test data change in the LTC?
3. Control Flow: Can you describe the execution sequence of test steps the LTC contains?
4. Operations: Can you describe what does a particular test step does?
5. State: Can you describe what the content of test data at a particular point of execution is?

For each question a point is given. We constructed only closed questions, to provide a simple measurement. If one of the questions cannot be answered, then one point is subtracted from the sum. On the scale 0-5 the highest understandability (5) or the lowest understandability (0) can be reached. To sustain the ob-

jectivity of the test case understandability, we ask three independent test experts from industry (test managers at Capgemini TS) and research (researchers from the Software Quality Lab) to answer the mentioned questions. For this, we use a simple questionnaire template based on the five questions. The results are further cross-examined with the answers provided by the thesis author. Finally, a mean value over the reached points is calculated.

### 6.1.3 Setting

The metrics mentioned in the last chapter are collected during the experiments on predefined input data (analysis model) and output data (test and trace model). To influence the input analysis model with respect to parameters as model relations and test selection criteria, we have defined six different sets for each experiment. The sets differ in the parameter values of *models coupling* (number of links between model elements) and used *test selection criteria*. The model coupling has two possible values: *complete* (all model relations from the analysis meta-model are implemented) and *incomplete* (not all model relations of the meta-model are implemented). We are particularly interested in the model relations between the different modelling viewpoints, since we aim to solve the missing holistic view problem (see Section 1.1). The test selection criterion has three possible values: *AllActions*, *AllPathsOneLoop* and *AllPathsAnnotated* (see Subsection 5.4.1). We depicted the distribution in the following table:

Table 11: Sets definition

Name	Model coupling	Test selection criterion
Set 1	complete	AllActions
Set 2	incomplete	AllActions
Set 3	complete	AllPathsOneLoop
Set 4	incomplete	AllPathsOneLoop
Set 5	complete	AllPathsAnnotated
Set 6	incomplete	AllPathsAnnotated



Since we perform a controlled experiment, the model coupling is influenced manually. The goal here is to observe the outcome of the treatment with and without the holistic view during test model generation. By using the modelling tool Enterprise Architect the essential linkage between use case' actor action and dialog action (see Subsection 5.4.2) is deleted in all use case models to achieve the *incomplete* state of model coupling. This state represents the situation where only the behavioural modelling viewpoint, namely the use case models are used for test generation. The relation to the interaction and structure modelling viewpoints is missing.

As stated in Subsection 5.4.1 about test selection algorithm, the recursive implementation of the different coverage criteria differs marginally. On the other hand, the influence on the number of generated test cases should strongly differ. For evaluation purposes we implemented all mentioned test selection algorithms and use them in both experiments.

To perform a structured experiment as introduced by Wohlin et al. in [WRH<sup>+</sup>99], we first define the hypotheses (null and alternative) which have to be tested during the experiments' execution.

#### 6.1.4 Null Hypotheses

We define the following null hypotheses:

- The usage of the holistic view does not improve the internal test quality of test models
- The usage of the holistic view does not influence the global model coverage of the analysis model
- The usage of analysis models for the test model generation does not result in lower effort in the test design phase

#### 6.1.5 Alternative Hypotheses

In case the null hypotheses from the last subsection are rejected, the following alternative hypotheses are automatically accepted:

- The usage of the holistic view improves the internal test quality of test models

- The usage of the holistic view influences the global model coverage of the analysis model
- The usage of analysis models for test model generation results in lower effort in the test design phase

## 6.2 TOOL SUPPORT

In this section, we briefly introduce the tool support used to perform the experiments of this thesis.

### 6.2.1 Motivation

The goal of the tool development is to support the execution of the model-based test specification process from Chapter 5. According to the instantiation process of the meta-model algebra from Section 4.8, each algebra operation with its algorithm is implemented within one or more tools. There are four high-level process steps from which two namely *"generate basic test model"* and *"generate concrete test cases"* are fully automated. Since the output of the first step is at the same time the input for the second one, we have decided to separate the development into two standalone applications. We have named the tools according to the process steps: Test Model Generator and Test Case Generator. We now briefly explain the technical architecture of both tools.

### 6.2.2 Test Model Generator

The purpose of the Test Model Generator is to support the execution of all the algebra operations from Section 4.6. This means that the algorithms for test selection (see Subsection 5.4.1), model analysis (see Subsection 5.4.2), model transformations (see Subsection 5.4.3) and model coverage measurement (see Subsection 5.4.4) are implemented within the generator. The implementation of the Test Model Generator was part from Annette Heym's [Hey10] and Andreas Fichter's [Fic10] master thesis, which were supervised by the author of this thesis.

The input is a model created by business analysts within a modelling tool. For proper execution the meta-model properties from Section 4.6 have to be fulfilled. To edit the analysis and test model we use the tool from Sparx Systems called Enterprise Architect<sup>1</sup>.

The output is the basic test model. It is described with the customized version of the UML testing profile from Section 2.3. Further extension of this model is also performed within Enterprise Architect for the same reason as in the case of the analysis model. Additionally, a trace model is generated which is used to automatically create a coverage report.

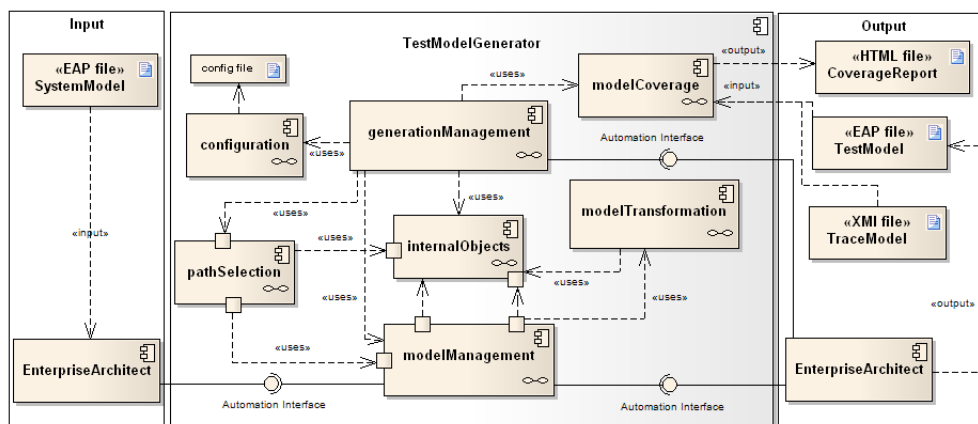


Figure 74: Architecture of the Test Model Generator

In Figure 74, we depicted the architecture of the Test Model Generator. To read input and write output models, we use the Enterprise Architect through a so called *Automation Interface*. We use this Java interface instead of model transformation languages like QVT, ATL or ETL (see Subsection 5.4.3), because of technical inconsistencies with the XMI specification of the Enterprise Architect. The central component of the test model generator is called *generationManagement*. The purpose of this component is to trigger the applications workflow. Within the *configuration* component all settings as input and output file paths, test selection algorithm to be used, etc. are read from a configuration file. The implementation of the test selection algorithms from Subsection 5.4.1 is hold in the *pathSelection* component. The model transformation algorithm is the main part of the *modelTransformation*

<sup>1</sup> <http://www.sparxsystems.com>

component. The management of the internal objects of the application (see *internalObjects* component) together with the model read/write process (see automated model analysis from Subsection 5.4.2) is done by the *modelManagement* component. Finally, the *modelCoverage* component is responsible for the generation of the coverage report based on the test and trace model.

To support the customization of the generation process the user is able to change or implement own test selection algorithms within the *pathSelection* component. Also, the customization of model transformation rules within the *modelTransformation* component is possible. Further, the form and content of the coverage report can be customized in the *modelCoverage* component with Epsilon Generation Language (EGL)<sup>2</sup>. EGL provides model-to-text transformations, which are compiled at runtime.

### 6.2.3 Test Case Generator

The Test Case Generator supports the last step of the model-based test specification process. The test data combination algorithm from Section 5.6 together with the PIM/PSM transformation are implemented within the generator. The implementation of the Test Case Generator was part of a bachelor thesis from Benjamin Niebuhr [Nie09], which was supervised by the author of this thesis.

As mentioned in the last subsection, the input for the Test Case Generator is the test model which was automatically generated and manually extended. It consists of logical test cases, the logical test data structure and the test architecture description to be used. The purpose of the Test Case Generator is to combine the logical test case and test data definitions from the test model with concrete test data sets from an external source. The algorithm for the generation of concrete test cases uses a holistic view on all three modelling viewpoints of the test model (see Section 2.3). The results are concrete test cases which are extended with platform-specific information for test execution.

In Figure 75, the architecture of the test case generator is shown. The main component called *TestcaseGenerator\_Manager* is responsible for the whole workflow of the test case generation process.

<sup>2</sup> <http://www.eclipse.org/gmt/epsilon/doc/egl/>

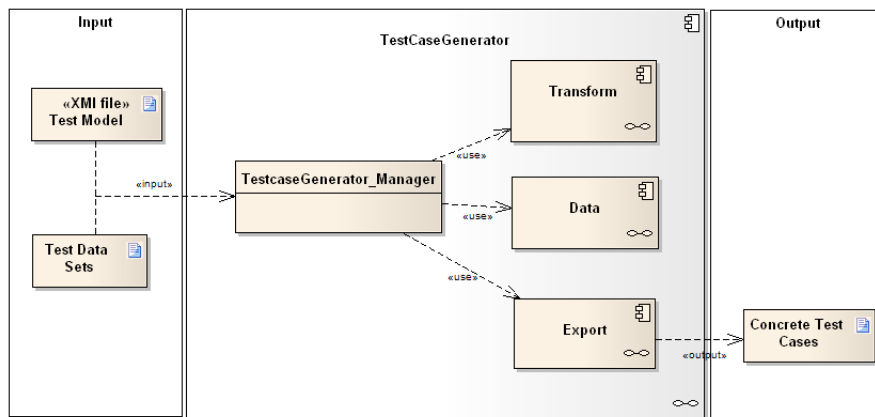


Figure 75: Architecture of the Test Case Generator

Three additional components support the single tasks. The *Data* component is responsible for reading the test model (which is the XMI [Obj07a] file exported from Enterprise Architect) and combining them with test data sets from an external source. The *Transform* component enables the transformation of all read data into an in-memory representation. Finally, the *Export* component transforms the concrete test cases into a platform-specific file format.

The three mentioned components (*Transform*, *Data* and *Export*) were implemented as Groovy<sup>3</sup> scripts. Groovy is chosen because of the same technical inconsistency with XMI problem as in the Test Model Generator. Groovy enables the interpretation at runtime and is standardized as a second official language for the Java Virtual Machine. This way, Groovy can be used similar to ETL for declarative specification of M2T transformations. This requirement was also identified during the elicitation of project needs in the industry research project conducted within this thesis.

#### 6.2.4 Used technology stack

For the implementation of the Test Model and Test Case Generator the following technology stack was chosen:

- Java SDK as the main programming language

<sup>3</sup> <http://groovy.codehaus.org/>

- Groovy SDK as the second programming language to support adaptability at runtime (instead of ETL)
- Java API for the Enterprise Architect to access the analysis model and to transform it into a test model (instead of ETL)
- Eclipse as the development environment, which supports both Java and Groovy
- Epsilon Generation Language to generate the coverage report
- Sparx Systems Enterprise Architect as the standard modelling tool
- XML Metadata Interchange (XMI) format as the export/import file format for UML models

#### 6.2.5 Used environment

To allow a controlled execution environment during both experiments, a constant hard- and software set has to be chosen. For the execution of the experiment presented here, the following environment was used:

- Apple iMac 2,66 GHz Intel Core i5 with 8 GB RAM
- Mac OS 10.6.6 with Parallels (virtualisation software)
- Windows XP SP2 (within Parallels 5)
- Java SDK 1.6.0\_23
- Enterprise Architect 7.1

### 6.3 EXPERIMENT "GABI'S SKI SCHOOL"

The following experiment uses the running example of this phd thesis, which was already introduced in Subsection 2.4.4.

#### 6.3.1 Input model

The composition of the input analysis model for the fictive project called "Gabi's Ski School" is shown in Table 12.

**Table 12:** Composition of the analysis model for "Gabi's Ski School"

Modelling Viewpoint	Models
Structure	3 conceptual components, 3 data models, 1 logical data type model
Behaviour	5 use cases with 5 activity diagrams, 5 application functions
Interaction	2 dialogs with 29 dialog elements and 10 dialog actions

This kind of analysis model is representative for a small software engineering project. Based on our industry observations, a typical large-scale project scales up to 50 use cases with several dialogs and conceptual component. However, for the purpose of a controlled experiment this set of input data is sufficient.

The most complex activity diagram within the analysis model contains:

- 7 action nodes (steps)
- 7 decision nodes

This information is relevant for the approximated number of generated logical test cases and the time effort for the automated model analysis, model transformation and model coverage measurement algorithms. By using the McCabe's complexity metric [McC76] for control-flow models with binary branches, the maximal number of 8 LTC for the most complex activity diagram is needed. Unfortunately, this metric does not consider backward-loops, which is the case in several diagrams of the analysis model. This way, the calculated number is only a guidance for the maximum number of LTC. The complexity has an impact on the time effort of the mentioned algorithms, since for each selected path of the activity diagram each algorithm has to be executed once.

### 6.3.2 Results

Using the analysis model from the previous subsection we have performed an experiment with six sets of input data (see Subsection 6.1.3), which resulted in six different test and trace models. The manual extension of this test model and generation of concrete test cases was omitted, since the main contribution of this work is to provide a holistic method for generating test models from analysis models. Additional completeness of logical test cases from the test model can be achieved by performing the last two steps of the model-based test specification process from Section 5.5 and Section 5.6.

Table 13: Experiment results

Metric	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Time effort	01:58	01:05	02:40	01:26	02:24	01:02
No. LTC	13	13	25	25	7	7
Global model coverage	64%	33%	64%	33%	58%	27%
Use case coverage	100%	100%	100%	100%	80%	80%
Dialog coverage	54%	0%	54%	0%	54%	0%
Data type coverage	38%	0%	38%	0%	38%	0%
Completeness	7/7	4/7	7/7	4/7	7/7	4/7
Traceability	0	0	0	0	0	0
Understandability	4/5	2/5	4/5	2/5	4/5	2/5
Analysability	0	13	0	25	0	7

The results of the performed experiment are shown in Table 13. In the next subsection we provide an interpretation of the experiment results.


### 6.3.3 Interpretation of results

Based on the results from our first experiment, several observations can be concluded. For better understanding, we group the observations and describe them in detail.




### Reached model coverage

The main focus of this phd thesis is the holistic usage of analysis models in model-based system testing. The goal of a holistic approach is to use several modelling viewpoints and cover as much information as possible needed to specify high-quality test models. Unlike the current MBT approaches (see Chapter 3 about the related work), we want to measure the model coverage of all modelling viewpoints and not only the behavioural models.

Global model coverage	64%	
Mean coverage: <b>Use Case</b>	100%	
Mean coverage: <b>Logical Data Types</b>	38%	
Mean coverage: <b>Dialogs</b>	54%	

(a) Complete, AllActions

Global model coverage	27%	
Mean coverage: <b>Use Case</b>	80%	
Mean coverage: <b>Logical Data Types</b>	0%	
Mean coverage: <b>Dialogs</b>	0%	

(b) Incomplete, AllPathsAnnotated

Figure 76: Comparison of the global model coverage

In Table 13 we observe strong differences between sets with different *model coupling* levels. For example in *Set1* we reached 64% *global model coverage* when the analysis model contained model links between the behaviour, interaction and structure modelling viewpoints. An analysis model without those model links used for test model generation resulted in a *global model coverage* of only 33% in *Set2*. The missing coverage of the interaction and structure viewpoint is mirrored by the 0% value of the *dialog* and *data type coverage*. We have depicted parts of the coverage reports in Figure 76 a) and b). The best global coverage could be

reached in *Set 1* and the worst in *Set 6*. As introduced in Subsection 5.4.4, the coverage report visualizes the current state of the model coverage by using different colours.

Though, in *Set3* and *Set4* more logical test cases were generated by the *AllPathsOneLoop* criterion, the reached coverage stays the same. This equal results are due to the measurement of *use case coverage* based on action nodes. We have discussed this topic in detail in Subsection 5.4.4. In the first four sets, all actions of the activity diagram are covered. Since we analyse the linkage between actor actions, dialog actions and logical data types, the other coverage metrics are equal in the mentioned sets.

While the *use case coverage* reached 100% in the first four sets and 80% in the last two, the dialog and logical data type coverage never reached more than 54%. This is caused by the fact that not all dialog elements and logical data types were used in the use cases. We depicted a part of the coverage report for the logical data type model (*Set1* and *Set2*) in Figure 76. The state presented there is questionable because a) not all model links were set or b) there exist dialog elements and data types not used in the use cases. In Figure 77 b), all logical data types have 0% coverage. Since we influence the model coupling during our experiment only b) is possible.

The missing 62% of logical data types in Figure 77 a) are used in further parts of the analysis model, as print output or batch specifications (see [SSE09]). The missing 46% of dialog elements and actions are probably obsolete in the analysis model and should be discussed with business analysts within a project. This way, the results of the holistic coverage measurement support the the investigation of the analysis model coverage.

In the last two sets of our experiment, the reached *global model coverage* is lower than in the other sets. This is caused by the *AllPathsAnnotated* selection criteria used here. Only the paths with annotated action nodes were selected from the activity diagram. This way only 80% of all action nodes were selected. This lowers the global model coverage in *Set5* and *Set6*.

In all sets used in our experiment, a 100% *global model coverage* was never reached. This result is obvious while looking at the metrics introduced in Subsection 5.4.4. To reach 100%, the remaining *use case*, *dialog* and *data type coverage* would also have to reach 100%. Further, the *global model coverage* is not calculated

Data Type	Coverage	Data Type	Coverage
DT_SkiPiste	0%	DT_SkiPiste	0%
DT_Count	0%	DT_Count	0%
DT_Surname	100%	DT_Surname	0%
DT_CourseName	0%	DT_CourseName	0%
DT_Amount	0%	DT_Amount	0%
DT_Char_256	0%	DT_Char_256	0%
DT_Number_10	0%	DT_Number_10	0%
DT_Firstname	100%	DT_Firstname	0%
DT_Address	100%	DT_Address	0%
DT_Time	0%	DT_Time	0%
DT_Name	0%	DT_Name	0%
DT_TimeOfDay	0%	DT_TimeOfDay	0%
DT_RaceType	0%	DT_RaceType	0%
DT_Level	100%	DT_Level	0%
DT_CustomerNumber	0%	DT_CustomerNumber	0%
DT_Date	100%	DT_Date	0%
DT_Race	0%	DT_Race	0%
DT_CourseNumber	0%	DT_CourseNumber	0%
DT_CourseType	100%	DT_CourseType	0%
DT_PhoneNumber	100%	DT_PhoneNumber	0%
DT_RaceTime	0%	DT_RaceTime	0%
DT_Age	100%	DT_Age	0%
DT_Employment	0%	DT_Employment	0%
DT_Gender	100%	DT_Gender	0%

(a) Complete

(b) Incomplete

Figure 77: Comparison of the logical data type model coverage

to all possible elements of the analysis model, but with respect to the mentioned types, which represent the different modelling viewpoints.

### *Model coupling and test quality*

One of the goals of this evaluation is to provide empirical evidence on the improvement of the internal test quality of the generated test model. To do so, we have instrumented the analysis model by adding (even-numbered sets) or removing (odd-numbered sets) the model links defined within the analysis meta-model.

The first observation is that the model coupling influences the completeness of logical test cases within the test model. In the case of incomplete analysis model, three out of seven attributes of a logical test case (see Subsection 2.1.1) could not be generated. In the other case, complete logical test cases with all seven attributes filled with data could be generated.

We have also observed that no change of the test models' traceability is given. This is caused by the fact that our algorithms always preserve the traceability information for model coverage purposes.

The understandability of the generated test cases was rated by four independent test experts (see Appendix A.1). Also here a correlation between the model coupling and understandability could also be observed here. In Figure 78, we depicted a part (first three test steps) of the logical test case generated for the *Book\_Attendee\_On\_Course* use case. This test case was sent to the three independent test experts to rate the understandability. In Figure 78 a), the complete test case (model coupling = complete) is shown. The opposite case is shown on the right side of Figure 78 b).

The generated logical test cases have not reached 5/5 for complete analysis model because of missing information in the analysis model. For example in Figure 78 a) there is no indication what test data have to be used as input in the *Select\_Course* test step. This way, the data flow question could not be answered properly. In the case of the incomplete analysis model the questions about data-flow, operations and state could not be answered, which leads to the 2/5 rating. In Figure 78 b) the data, dialog and trig-

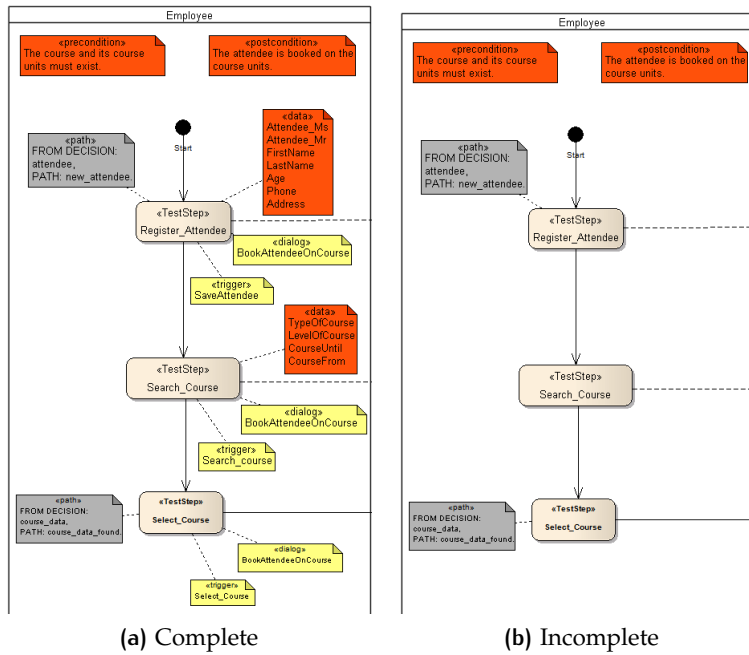


Figure 78: Comparison of complete and incomplete logical test cases

ger notes are missing completely and therefore influence the understandability rating. The answers collected with our questionnaire from three independent test experts and the thesis author are depicted in the Appendix A.1.

To measure the analysability, we have inspected the existence of relations to the test architecture or test data package within the test model. Since the generation of such relations strongly depends on the automated model analysis algorithm from Subsection 5.4.2, the model coupling of the analysis model is important here. In the case of a complete analysis model, also complete information about the test architecture and test data package could be generated. This leads to highly-analyzable logical test cases. In the case of incomplete analysis models all generated test cases are hard to analyze according to our metric.

#### Time effort

The second goal of this evaluation is the influence of the time effort needed to generate logical test cases which are part of a test model. To compare our results we use the mean value needed in manual test design which is 2,67 person hours / test case

as introduced in [RBGW<sub>10</sub>, p.362]. In Table 13, the mean time effort needed to generate the test model equals 01:45 minutes. On average, 15 test cases have been generated. The time effort needed to design the average 15 logical test cases in manual test design equals 40 hours. This leads to a significant improvement of the time effort. But the simple comparison of 01:45 minutes (automated generation) and 40 hours (manual test design) is not plausible, since further tasks (as modelling, model review, model extension, etc.) and the resulting additional effort have to be included.

This kind of effort comparison also strongly depends on the chosen model-based testing scenario. A paper called "Effort Comparison of Model-based Testing Scenarios" [GMS<sub>10</sub>], dealing with this problem, was published as part of this phd thesis. Based on the identified literature, seven different model-based testing scenarios were systematically compared according to different parameters. Without providing empirical evidence, we have shown in [GMS<sub>10</sub>] that the comparison of the scenario used in this thesis (using analysis models for test model generation) with other scenarios according to the needed effort is favourable.

**Table 14:** Comparison of time effort (in PH) in different scenarios

Scenario	Modelling	LTC Generation	Review & Extension
Manual test design	0	2,67	0
Manual modelling	30 - 2040	0,05 - 0,68	NA
Model from model	0	0,02	LTC*0,2

To provide a more plausible discussion, we compare the time effort in Table 14 for the following three scenarios: *manual test design*, *manual modelling scenario* from Pretschner and Philips [PP05] and *model from model* scenario from Gldali et. al. in [GMS<sub>10</sub>]. We compare the scenarios based on the time effort needed for *modelling* (effort needed to create a test model), *LTC generation* (effort needed to generate one LTC) and *manual review & extension* (effort needed to review the generated LTC and extend them with test data). We use the person hour (PH) unit for each metric in this table.

The time effort for the *manual test design* scenario is based on the case study from Rossner et al. [RBGW10, p.362]. In this scenario no effort for creating a test model and for reviewing & extending the LTC is needed. This is obvious since in manual test design the LTC are designed using the tester's knowledge (also called the mental test model) about the SUT requirements rather than using an explicit test model.

To collect empirical data for the *manual modelling* scenario, we have analyzed our literature survey from Chapter 3 with respect to the *Case Study* evaluation criteria. We have also identified further publications (as [UL07, PPW<sup>+</sup>05, RBGW10, HNo4, DNSV<sup>+</sup>08]) and unpublished talks (as [Lego8, CSH03]), which provide empirical data. Unfortunately, most case studies analyze the fault-finding rate or model coverage as the effectivity metrics. Only minor publications consider efficiency metrics (like modelling effort, test generation effort, etc.). For example, in Utting and Legeard [UL07, p.36] the time needed to create a test model for a hypothetical example is given by 30 PH. In a talk from the EuroStar 2008 conference, Bruno Legeard provided a case study for testing a large-scale financial system [Lego8]. According to this study, the time effort needed here was 69 person days, which equals 552 PH. Finally, the modelling time needed in the case study introduced by Rossner et al. [RBGW10, 357] was calculated with 255 person days, which equals 2040 PH. Based on this three exemplary case studies, we have approximated the modelling effort in the boundaries from 30 to 2040 PH.

The mean time effort for *LTC generation* based on the three mentioned references resulted in the boundary from 0,05 PH (EuroStar 2008 case study [Lego8]) to 0,68 PH (Rossner et al. [RBGW10, p.362]). To the best of our knowledge, we are not aware of case studies, which provide effort metrics for reviewing and extending test models or even the generated LTC. That is why the NA (not available) value is depicted in Table 14.

Finally, the *model from model* scenario, which is analyzed in this thesis, results in no effort needed for modelling test models. Since we use analysis models created by business analysts, this initial task of MBT is not needed. Additionally, the effort for the testability check and annotation of use cases (see Section 5.3) could be measured in future experiments. The mean value for LTC generation based on the experiment results equals 0,02 PH. For the *review & extension* task, we assume a time effort of 0,2 PH

(10 minutes) for each generated LTC. This value is based on the experiences collected during the generation of concrete test cases with the Test Case Generator. For more reliable values, further experiments or case studies have to be performed in the future.

The main goal of Table 14 is to depict the time effort improvement of our approach with respect to the modelling task. The values referenced in this table have to be analyzed with caution. First, we compare pure PH without considering aspects like system type, complexity of the SUT, test designers skill level or the complexity of generated LTC in the referenced case studies. Second, high modelling effort can improve the test independency level (see discussion Chapter 1 or 3) and this way lead to a higher fault-detection rate. That is why this comparison gives only a broad idea and should not be understood as a verification of the effort improvement.

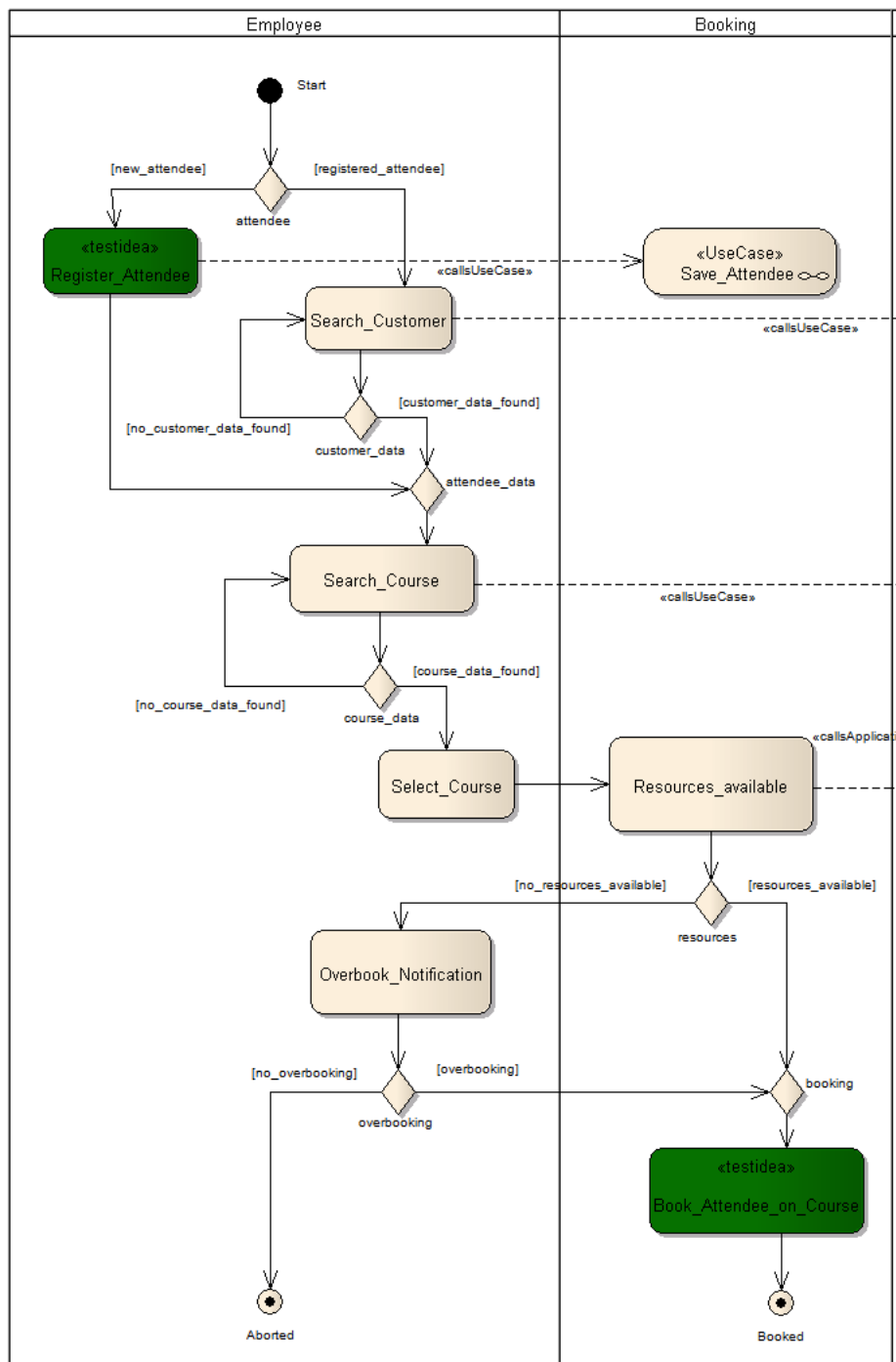
Another observation is the difference in time effort between the used sets. The time effort in the case of incomplete analysis models is always lower than by using the complete analysis model. This is caused by the additional time effort for automated model analysis (see Subsection 5.4.2) for each generated logical test case. If no model links exist, then no additional information is collected. Another observation is that this additional time effort increases proportionally with the number of generated test cases. Future work can improve the algorithms in Section 5.4 to mark the already visited model links during the automated model analysis. This way, the additional time effort could be improved.

### *Test selection*

In Subsection 5.4.1, we claimed that the number of generated test cases strongly depends on the test selection criteria used. While analyzing Table 13 this correlation is visible. The usage of a weak coverage criteria as *AllActions* results in a medium number of 13 test cases. The number of test cases is almost doubled when using a much stronger coverage criteria as *AllPathsOneLoop*.

In the last case, we have annotated a maximum of two action nodes within the use case' activity diagram. An example of the annotated use case *Book\_Attendee\_On\_Course* is shown in Figure 79. From the business point of view, the actor action *Register\_Attendee* is executed very often and therefore critical. The sys-





**Figure 79:** Annotated use case *Book\_Attendee\_On\_Course*

tem action called *Book\_Attendee\_On\_Course* is also critical for the business, since the booking in the system is essential for this use case. Through this annotation the number of logical test cases could be reduced from 16 (selecting with *AllPathsOneLoop*) to 4. The usage of the *AllPathAnnotated* coverage criteria resulted in a much lower total number of seven test cases. Assuming that this scale effect also holds on more complex models, a reduction of almost 50% in the number of generated logical test cases can be observed.

The improvement of the test quality and time effort combined with the observed importance of the usage of the holistic view and model relations, leads us to the evidence of the contrary for the null hypotheses from Subsection 6.1.4. This way, all alternative hypotheses are accepted.

## 6.4 DISCUSSION OF THE RESULTS

In this section we discuss the results of the performed experiment. We especially inspect the threats to the internal, construct and external validity of our evaluation.

### 6.4.1 Internal validity

*Side effects*

The internal validity describes the degree of influence through side effects, which might distort the results reached during experiment's examination [WRH<sup>+</sup>99]. The side effects in our case can be triggered only by the person planning and executing the experiment. We minimize all external influences by providing a single controlled environment and predefined tasks (as the manipulation of model relations and change of test selection criteria) which influence the experiments results. The input models used in the experiments were created using the same modelling approach from Salger et al. [SSE09]. Further, the environment with respect to the hard- and software, was not changed during the evaluation phase.

*Manual  
manipulation of the  
input model*

However, the following two aspects may provoke side effects during the experiment's execution. First, the manipulation of model links has to be defined precisely. In the first experiment,

we have deleted all model links between the use case actor action and the dialog action for all use cases of the analysis model. If the manipulation considered only a part of the model links, the results of this experiment might change. Since we always deleted the same model links the internal validity of the first experiment holds.

A second aspect, which may influence the results is the manual test annotation process. In the first and second experiment we have annotated a maximum of two action nodes of each use case' activity diagram. Annotating other, more or less nodes will lead to other experiments results in terms of the number of generated test cases. Since in all sets used in the experiments always the same action nodes were annotated, the internal validity is still guaranteed.

*Manual use case  
annotation*

#### 6.4.2 Construct validity

The validity of the experiment's construction concerns the relation between the theory (in our case the holistic model-based testing approach) and collected observations [WRH<sup>+</sup>99]. Within our theory we have shown the relation between the model relations from the analysis meta-model and the usage of the holistic view during the test model generation (see Subsection 5.4.2) and holistic model coverage measurement (see Subsection 5.4.4). This way, we have constructed a cause *usage of model relations*, which leads to two effects: *improvement of internal test quality* and *improvement of the model coverage level*.

*Relation between  
theory and  
observations*

The cause/effect construction is defined at the theory level. Within our evaluation we have constructed the treatments *model coupling* and *test selection criteria*, which resulted in several outcomes (for example difference in the global model coverage level) discussed in the last section. In order to discuss the generalisation of the treatment/outcome relation, its validity to the cause/effect construction from the underlying theory has to be discussed first.

There are different threats to the construct validity, which focus on design or social aspects of the evaluation. Within the design threats the *inadequate preoperational explication of constructs* states that the theory could be not clear enough and this way the experiment cannot be sufficiently clear. In the first three chapters of this thesis, we have clearly defined the research problem of the

*Clear problem and  
solution definition*

missing holistic view and its effect on the internal test quality and model coverage. To design a holistic model-based testing approach, we have defined the meta-model algebra which enables the specification of meta-model properties and algorithms (operations on meta-models) needed for our approach. Finally, in Chapter 5 the approach, its relation to the research problems and the cause/effect relation were shown.

*Mono-operation bias*

Another important threat is the *mono-operation bias*, which questions the underrepresentation of the experiment by using a single variable, case, subject or treatment. In our evaluation, we have used two treatments (model coupling and test selection criteria) and measured several variables (see metrics from Subsection 6.1.2). Since we used only one subject, this threat can be violated during the generalisation of the evaluation results. The only possible solution is to perform more experiments on different subjects in the future.

*Mono-method bias*

During the measurement of the experiments results, the *mono-method bias* validity threat can be violated. While using a single measurement method for measuring a variable (for example the *completeness* of generated LTCs), the results can be misleading. Except for the *understanability*, where we cross-checked the results with three independent test experts, all variables in our experiment use single measurement methods. The mono-method bias threat is still not violated, since we use objective metrics (except for understandability) which are not based on subjective valuations.

*Interaction of treatments*

Since we use different treatments (model coupling and test selection criteria) in our experiment, the validity threat *interaction of different treatments* can occur. To enable a correct conclusion, we have constructed six sets which use all possible combinations of the mentioned treatments. This way the interaction of treatments in all possible combinations is evaluated.

*Measurements and generalisation*

In our experiment, we violate the *restricted generalisability across constructs* threat. Especially while observing the *time effort* variable we do not measure other possible variables as initial modelling time, time needed to extend the generated test model, etc. This way, our construction does not enable the generalisation of the time effort improvement according to the hypotheses of this thesis. We discuss this issue in the next subsection.

Finally, the social threat to construct validity, namely the *experimenter expectancies* has to be discussed. While the experiment was performed by the author of this thesis, this can bias the results of the experiment. To minimize this threat, the evaluation results were reviewed by several test experts.

### 6.4.3 External validity

Besides the internal validity, the generalisation of experiment results is discussed to provide new evidence for the research community. This kind of generalisation is widely known as external validity [WRH<sup>+</sup>99]. In the case of this phd thesis, the results of the experiment underpin the contribution of the holistic approach.

The main observation, which can be generalised is the correlation between the model coupling (level of model relations instantiated within the analysis model based on its meta-model definition) and the internal test quality. Besides the traceability, all other quality attributes were improved in the case of a complete analysis model. Missing model relations made it not possible to use the holistic view and therefore to use all three modelling viewpoints (structure, behaviour and interaction) in a context for test generation purposes. In our experiment we have used the exemplary modelling approach from Salger et al. [SSE09] which incorporates the mentioned modelling viewpoints. Modelling approaches fulfilling the property of different modelling viewpoints and relations between them (see meta-model property model relations in Section 4.5), improve the internal test quality while using our holistic model-based system testing approach. However, the proof of this statement requires further experiments with analysis models created according to other modelling approaches.

*Model coupling and  
internal test quality*

Besides the aspect of test quality improvement, we have observed the correlation between the holistic view and reached model coverage. Since incomplete analysis models disable the usage of several modelling viewpoints, the reached coverage significantly decreases. This correlation has been shown in the experiment and can be generalised since the holistic coverage measurement approach presented in this thesis always considers several modelling viewpoints. This generalisation assumes the usage of

*Holistic view and  
model coverage*

analysis meta-models fulfilling the meta-model property model relations (see Section 4.5).

*Time effort  
improvement*

As mentioned in the interpretation of the experiments results, we have observed a strong improvement of the time effort needed to create logical test cases. This result indicates the general improvement of time effort while using models created by business analysts for model-based testing. The indication in our case is based on some assumptions. First, the comparison with manual test design has always to consider the model-based testing scenario as described in [GMS10]. Second, the effort for additional tasks as for example the modelling effort, model review, etc. has to be added to the time effort needed for the generation of the test model. To the best of our knowledge, such representative statistics for the time effort needed to perform the additional tasks do not exist in the literature or industry case studies. This way, the generalisation of the time effort improvement for the particular model-based testing scenario cannot be fully generalised.

*Improvement of  
understandability*

Another candidate for the generalisation is the improvement of the understandability, while using the holistic model-based system testing approach. The problem with understandability is its subjective nature. We have solved it by asking three independent test experts according to the customized method from Pennington [Pen87]. In order to generalise the improvement of the test case understandability, further experiments with different analysis models and more test experts are needed.

Regarding the generalisation of the experiment performed within this thesis, we conclude that the phd hypotheses defined in Chapter 1 are accepted. Both evaluation goals (test quality and time effort improvement) were met and are based on the experiments results. The restrictions and assumptions regarding the generalisation of our experiment was described within this subsection.

## 6.5 SUMMARY

In this chapter, we have shown the evaluation of the research approach presented in this thesis. We have introduced the evaluation plan with its goals and metrics used. Since we aimed at generating test models automatically, the according tool support

was briefly introduced. We have performed an experiment of two different analysis models. Within the experiment, we used six sets of data with respect to the model coupling and test selection criteria. The experiment results underpin the hypotheses of this phd thesis and its contribution. We have introduced the aspects of the internal, construct and external validity to provide a objective interpretation of the results.





---

## SUMMARY AND OUTLOOK

---

Using all viewpoints of analysis models for model-based system testing, while improving the internal quality of test artefacts, is how this thesis can be summarized. In the last six chapters we have defined the main research problem of the *missing holistic view* and two subproblems of *using analysis models for test generation* and *measuring coverage of the analysis model*. Based on the problem definition, we defined three phd hypotheses. After defining several preliminaries, we have shown a wide literature survey for the mentioned research problems. This survey showed that holistic approaches are missing and that the topic of internal test quality in model-based testing is poorly considered.

Then, we performed a top-down design of a holistic approach for model-based system testing. First, we defined a high-level meta-model algebra, which allowed us to specify operations performed on meta-models and the required properties of meta-models. In the second step, we defined a four-step approach with two manual and two automated steps. The automated steps were specified with the meta-model algebra, since they operate on an analysis, test and trace meta-model.

To accept or reject the phd hypotheses, we performed an experiment based on fictive example representing typical specification of SRS for business information systems. We discussed the experiment results and its generalisation, while considering several threats to validity.

Until now, we have summarized the research problems and solutions developed in this thesis. In the next two sections, we will provide a detailed summary of the reached contributions and give a outlook on future work.

## 7.1 SUMMARY OF CONTRIBUTIONS

In the following, we discuss how far the contribution points from Section 1.2 were reached in this thesis. In this discussion we reference to the results from the last six chapters.

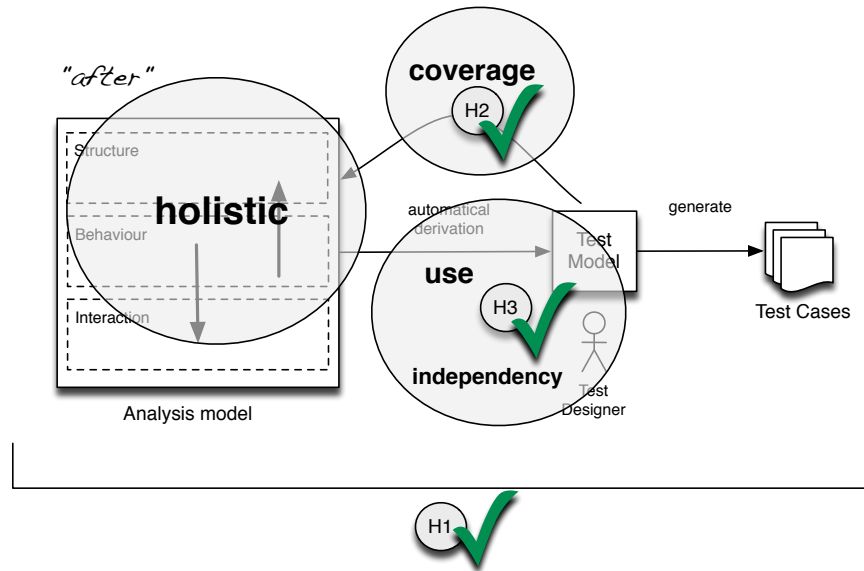


Figure 80: Reached contributions of this phd thesis

*Visualization of  
research problems  
and hypotheses*

To visualize the thesis results we introduce Figure 80. We symbolise the investigated research problems with big circles named after the problem definition from Chapter 1. The small circles represent all three parts of the phd hypothesis. *H1* stands for the improvement of internal test quality. *H2* depicts the impact on the reached model coverage of the analysis model. Finally, *H3* stands for the improvement of time effort in MBT scenario. As depicted in Figure 80 all the problems were tackled in this thesis with appropriate solutions. Also each of the hypotheses was accepted by performing an experiment, which is depicted with the green symbol. A similar visualization was already presented in the summary of Chapter 5 (see Section 5.7). Knowing this, we discuss each contribution point in detail now.

### Improvement of the internal test quality

The application of the holistic view leads to the improvement of the internal test quality. We discussed this topic while introducing the research approach. Especially the application of the algebra operation extract (see Section 4.6) and the algorithm for automated model analysis (see Subsection 5.4.2) lead to more complete, understandable and analysable test models. The according hypothesis from Section 1.1 was accepted by providing empirical evidence in the evaluation chapter. The correlation between holistic view and test quality improvement was shown within our particular example of an analysis model. The generalisation of this correlation and several validity threats were discussed in Section 6.4.

### Use of analysis models for test model generation

Our approach is an example of the *model from model* model-based testing scenario [GMS10]. We showed how to generate test models described with UTP directly from analysis models. Different to the approaches found in the literature (see discussion in Subsection 3.2.1 and 3.2.4), we combine the test selection and further algorithms with model transformations. Compared to model-based testing scenarios, where the test model has to be created manually, our approach improves the time effort needed for manual modelling. In the experiment performed in Chapter 6, we have provided first (to the best of our knowledge) empirical evidence for the effort improvement by this model-based testing scenario. This way, we also contribute to the empirical body of knowledge within MBT, which is still deficient as stated by Dias Neto et al. in [DNSV<sup>+</sup>08]. However, these results can not be fully generalised as discussed in Section 6.4.3.

Besides the time effort improvement, our approach also supports the quality assurance of the analysis model. Within the step 3 of our research approach (review and extension of the test model in Section 5.5), we showed that test designers can find faults in the analysis model by analyzing the automatically generated test model. Further, the investigation of the coverage report supports the quality assurance of the analysis model as shown within our evaluation in Section 6.3.3.

### Use of several modelling viewpoints

Our holistic approach for model-based testing is the first one using more than two modelling viewpoints while generating test models from analysis models. To the best of our knowledge, we could not identify any approach fulfilling all the requirements (see evaluation criteria in Section 3.1) derived from the problem statement. In our approach we have combined several algorithms for test selection, automated model analysis, model transformation and model coverage measurement, which operate on the different modelling viewpoints. This combination of algorithms is novel in the domain of model-based system testing. By the usage of several modelling viewpoints, we could improve the internal quality of test models generated automatically.

Further, through abstraction of the mentioned algorithms, we defined a meta-model algebra (see Chapter 4). This language allows the specification of behavioural aspects of method engineering and meta-modelling. It also incorporates the definition of informal meta-model properties needed to execute the algebra operations. This way, we can specify a holistic model-based system testing approach on a very high-level. Through this abstraction, the comparison with other MBT approaches and extension of our approach with further techniques (as model checking, constraint modelling, etc.) is supported.

### Use of the integrated interaction viewpoint

In Subsection 3.2.6, we have discussed the topic of model-based GUI testing. We have shown that most of the current approaches make use of GUI models separated from the analysis model. In our approach we use the interaction viewpoint integrated into the analysis model. This way, the relation between behavioural models, as use cases and dialog information can be used in a context. The logical test cases of the generated test model incorporate information about the user interface through dialog notes. Further, through the relation between dialog elements and logical data types in the exemplary analysis model, the data types for equivalence class analysis can be identified.

Within the discussion in Subsection 3.2.6, we have pointed out that GUI models as the event interaction graph from [MSP01]

specify the behavioural aspects, which are used to generate test cases. In our approach we do not generate test cases directly from the interaction viewpoint, but use this information to append GUI information to logical test cases. This way, the internal test quality in terms of completeness and understandability is improved, but not the external test quality as the fault-detection rate as in the current approaches.

#### Use of model relations created by business analysts

The usage of model relations for testing is not new. We have discussed several exemplary approaches in Subsection 3.2.5. Different to the discussed approaches, our solution uses model relations already implemented by business analysts in the analysis model. The opposite case, where testers incorporate model relations within a test model, seems to be a common practice in current approaches. Through the analysis of the reached holistic model coverage, the users of our approach can find missing relations in the analysis model and therefore improve its quality (see Subsection 6.3.3).

The general correlation between the holistic view, model coverage and the level of model relations in the analysis model was defined in Subsection 5.4.2 and 5.4.4. We have provided empirical evidence for this correlation in that we performed an experiment. The improvement of test quality by the usage of the holistic view and the improvement of the reached model coverage depend on the level of model coupling as shown in Subsection 6.3.3.

#### Holistic model coverage measurement

Different as in the current approaches, we measure the coverage of several modelling viewpoints. In the first part of this phd thesis (Chapter 1 and 2), we have shown that there exist several coverage criteria in the literature, which have to be combined for the holistic coverage measurement. With the algebra operation cover and its related property traceability (Section 4.5 and 4.6), we have defined the coverage measurement on the meta-model level. In the second step of our approach (Section 5.4), we generate a trace model to automatically calculate several cover-

age metrics. The result of the measurement is a coverage report. This report was used in the experiment to provide empirical data and to prove the general correlation between the holistic model coverage and model relations.

## 7.2 OUTLOOK

The development of the holistic model-based system testing approach revealed several problems, which were ostracized in this thesis. Subsequently, we briefly describe each identified problem.

### Impact analysis and change incorporation

In Subsection 5.4.4, we have discussed an important problem when changes in the analysis or the test model occur. We have referenced an impact analysis approach (see [Far10]) which operates on similar meta-models. This approach aims to identify changes in the analysis model, and identify affected parts of the test model. However, the opposite case, when changes in the test model occur, also has to be investigated. This is especially important for the model coverage measurement since it can affect the global coverage. For the industrial application of our solution, the referenced method has to be customized and integrated into the Test Model Generator.

### Generation of infeasible paths

The automated generation of the test model incorporates the selection of test cases. Our solution and evaluation uses three similar transition-based coverage criteria to select paths. We defined a new *AllPathAnnotated* criterion to lower the number of generated paths while at the same time regarding the priority of single use case actions (see Subsection 5.4.1). We have also mentioned the topic of infeasible paths, which can be selected by several transition-based coverage criteria. For example a guard taken in one decision node can influence the selection in subsequent decision nodes. To omit this problem, the test selection

algorithms should use decision tables (excluding the infeasible combinations) in the future.

#### Automated test data selection

As discussed in Subsection 5.6.1, the holistic approach can be improved by enabling an automated generation of concrete test data. For this, the logical structure of test data pools from the test model can be used. Additional constraints can be incorporated at the test model (for example with OCL) or database (for example with SQL) level. Even though there exist several approaches for test data generation, it is still a NP-hard problem as proved in [Win99].

Besides the mentioned problems, several future work packages were identified.

#### Formalization of the meta-model algebra

The meta-model algebra enables the behaviour specification of operations and properties of meta-models used in this approach. While we introduced how to specify operations (template and visual representation) and algorithms (UML activity diagrams with object flow), the meta-model properties are specified only with textual descriptions. As discussed in Section 4.9, more formal languages as OCL or graph transformation languages can be used here. Together with further formalisations, also more research should be done on the properties of algebra operations (known from elementary algebra).

#### Modelling and generating the test oracle

While introducing model-based testing (Section 2.2) and evaluating several approaches in our literature survey (see Chapter 3), we have mentioned that MBT can be used to generate the test oracle. This way, the comparison between the expected result and the result of an manual or automated execution of test cases can be performed. In our exemplary analysis model, the expected results are derived from the use case and application function's

postconditions. As shown in several examples in this thesis, the postcondition of generated logical test cases is always incorporated. But, the expected results for single test steps occur only if other use cases or application functions are called. In order to generate test cases with test oracles, further research should investigate the extension of the analysis modelling approach (expected results for each use case step).

#### Non-functional testing

The focus of the holistic approach for model-based system testing lies on functional testing. Each analysis model (or SRS) in general, consists also of quality requirements for aspects as performance, usability, etc. For example use cases can additionally specify the time behaviour and this way incorporate quality requirements. The approach presented in this thesis, can be easily extended to support time behaviour for performance testing. The test selection and model analysis algorithms have to be customized according to the new meta-model, which incorporates time behaviour. For other quality requirements as usability, approaches from the usability testing field are more suitable.

#### Application in different domains

Throughout this thesis, we have focused on the domain of business information systems. We have used the definition from Neto et al. [NRP05], where the information systems support the business workflow, the user communicates with the system through a graphical user interface and a persistence layer to the underlying data exists. Since the application of the holistic view and the usage of developer models for MBT can be generalised, the application of our approach in other domains as embedded systems should be investigated.

## 7.3 FINAL STATEMENT

Model-based testing is the next level of software testing. The international research community and several industry companies



have identified the need for using and researching this technique. The idea of a holistic view together with using developer models for MBT adds a meaningful and practical aspect to this research field.



---

## EXPERIMENT RESULTS

---

In this appendix, we provide additional information for the experiment performed in Chapter 6. First, the detailed description of the understandability survey is presented. Then, the coverage reports for several sets of the experiment are depicted.

### A.1 UNDERSTANDABILITY QUESTIONNAIRES

To sustain objectivity in the analysis of the understandability of generated LTC, we have performed a questionnaire with three independent test experts from Capgemini Technology Services and the Software Quality Lab (s-lab) plus the author of this thesis. In Table 15 we have depicted the answers given for an exemplary *complete* logical test case. Compared to it, we depicted the answers for the *incomplete* logical test case in Table 16. The mean value derived from Table 15 is  $4/5$  and from Table 16 is  $2/5$ .

**Table 15:** Comparison of interview answers for **complete** logical test case

Question	Interviewer 1 (Capgemini)	Interviewer 2 (Capgemini)	Interviewer 3 (s-lab)	Author (s-lab)
Q1	yes	no	yes	yes
Q2	yes	yes	no	no
Q3	yes	yes	yes	yes
Q4	yes	no	yes	yes
Q5	yes	yes	yes	yes
<b>Sum</b>	5/5	3/5	4/5	4/5

**Table 16:** Comparison of interview answers for **incomplete** logical test case

Question	Interviewer 1 (Capgemini)	Interviewer 2 (Capgemini)	Interviewer 3 (s-lab)	Author (s-lab)
Q1	yes	yes	yes	yes
Q2	no	no	no	no
Q3	yes	yes	yes	yes
Q4	yes	no	yes	no
Q5	no	no	no	no
<b>Sum</b>	3/5	2/5	3/5	2/5

In Figure 81 we depicted the questionnaire template used to interview the independent test experts. The questionnaire was send via e-mail to all participants. Additionally, the complete/incomplete test cases as UML activity diagrams together with a textual description of each test and check step was attached to the mentioned e-mail.

The interviewed persons had the possibility to place additional notes for each question in Figure 81. Two improvements were stated by all participants. First, the additional textual description of each test step is necessary to understand both logical test cases. The automatically generated test model incorporates this information. For the purpose of the questionnaire, the textual description was extracted in a separate pdf file. Second, the questionnaire should use the term "logical test data" and not only "test data". This is necessary, because the logical test case does not contain concrete test data values for which the synonym "test data" is often used. Instead, "logical test data" (for example equivalence classes, logical data types, etc.) is used in each data note. Both problems were identified, while discussing the questionnaire results with each participant.

## A.2 COVERAGE REPORTS

To provide a detailed view on the results of the model coverage measurement from Table 13 in Section 6.3, we depict the complete coverage report for Set 1 and 2 (AllActions) and Set 5 and 6 (AllPathsAnnotated). Since the only difference between sets

**Understandability of TC1\_complete**


		[X]
<b>1 Function: Can you describe the overall functionality of the logical test case?</b>		yes <input type="checkbox"/> no <input type="checkbox"/> <i>Notes</i>
<input type="text"/>		
<b>2 Data flow: Can you describe where does the test data change in the logical test case?</b>		yes <input type="checkbox"/> no <input type="checkbox"/> <i>Notes</i>
<input type="text"/>		
<b>3 Control flow: Can you describe the execution sequence of the test steps?</b>		yes <input type="checkbox"/> no <input type="checkbox"/> <i>Notes</i>
<input type="text"/>		
<b>4 Operations: Can you describe what a particular test step tests?</b>		yes <input type="checkbox"/> no <input type="checkbox"/> <i>Notes</i>
<input type="text"/>		
<b>5 State: Can you describe what ist the content of test data at a particular point of the logical test case?</b>		yes <input type="checkbox"/> no <input type="checkbox"/> <i>Notes</i>
<input type="text"/>		

**Figure 81:** Questionnaire template used to analyze the understandability of logical test cases.

1,2 and 3,4 is the number of generated LTC, we omit set 3 and 4 here.


#### A.2.1 Report for Set 1 and 2

##### Coverage of the analysis model:

Global model coverage	64%	
Mean coverage: <b>Use Case</b>	100%	
Mean coverage: <b>Logical Data Types</b>	38%	
Mean coverage: <b>Dialogs</b>	54%	

(a) Set 1

##### Coverage of the analysis model:

Global model coverage	34%	
Mean coverage: <b>Use Case</b>	100%	
Mean coverage: <b>Logical Data Type</b>	0%	
Mean coverage: <b>Dialogs</b>	0%	

(b) Set 2

Figure 82: Global coverage from the coverage report of Set 1 and 2

**Use case coverage:**

Use Case	Coverage
<b>Search_Customer</b> Test Cases: TC_1	100%
<b>Search_Course</b> Test Cases: TC_1 TC_2	100%
<b>Book_Attendee_on_Course</b> Test Cases: TC_1 TC_2 TC_3 TC_4 TC_5 TC_6	100%
<b>Save_Attendee</b> Test Cases: TC_1 TC_2	100%
<b>Create_customer</b> Test Cases: TC_1 TC_2	100%

(a) Set 1

**Use case coverage:**

Use Case	Coverage
<b>Search_Customer</b> Test Cases: TC_1	100%
<b>Search_Course</b> Test Cases: TC_1 TC_2	100%
<b>Book_Attendee_on_Course</b> Test Cases: TC_1 TC_2 TC_3 TC_4 TC_5 TC_6	100%
<b>Save_Attendee</b> Test Cases: TC_1 TC_2	100%
<b>Create_customer</b> Test Cases: TC_1 TC_2	100%

(b) Set 2

**Figure 83:** Use case coverage from the coverage report of Set 1 and 2

Dialog element	Coverage	Dialog element	Coverage
Age	100%	Age	0%
Phone	100%	Phone	0%
CourseUntil	100%	CourseUntil	0%
Attendee_Mr	100%	Attendee_Mr	0%
LevelOfCourse	100%	LevelOfCourse	0%
Attendee_Ms	0%	Attendee_Ms	0%
TypeOfCourse	100%	TypeOfCourse	0%
Search_course	0%	Search_course	0%
LastName	0%	LastName	0%
CourseFrom	100%	CourseFrom	0%
Course	0%	Course	0%
overbooking_no	0%	overbooking_no	0%
Course_details	0%	Course_details	0%
SaveAttendee	0%	SaveAttendee	0%
Search_Customer	0%	Search_Customer	0%
Address	100%	Address	0%
FirstName	0%	FirstName	0%
overbooking_yes	0%	overbooking_yes	0%
BookAttendee	0%	BookAttendee	0%
IsBooked	0%	IsBooked	0%

(a) Dialog elements - Set 1

(b) Dialog elements - Set 2

Dialog action	Coverage	Dialog action	Coverage
Select_Title	0%	Select_Title	0%
Search_Course	100%	Search_Course	0%
Add_Course	0%	Add_Course	0%
Search_Customer	100%	Search_Customer	0%
Select_Course	100%	Select_Course	0%
overbooking	100%	overbooking	0%
Book_Attendee	100%	Book_Attendee	0%
Save_Attendee	100%	Save_Attendee	0%
Select_CourseUnit	0%	Select_CourseUnit	0%

(c) Dialog actions - Set 1

(d) Dialog actions - Set 2

Figure 84: Dialog coverage from the coverage report of Set 1 and 2



Coverage of the logical data type model:

Data Type	Coverage
DT_SkiPiste	0%
DT_Count	0%
DT_Surname	100%
DT_CourseName	0%
DT_Amount	0%
DT_Char_256	0%
DT_Number_10	0%
DT_Firstname	100%
DT_Address	100%
DT_Time	0%
DT_Name	0%
DT_TimeOfDay	0%
DT_RaceType	0%
DT_Level	100%
DT_CustomerNumber	0%
DT_Date	100%
DT_Race	0%
DT_CourseNumber	0%
DT_CourseType	100%
DT_PhoneNumber	100%
DT_RaceTime	0%
DT_Age	100%
DT_Employment	0%
DT_Gender	100%

(a) Set 1

Coverage of the logical data type model:


Data Type	Coverage
DT_SkiPiste	0%
DT_Count	0%
DT_Surname	0%
DT_CourseName	0%
DT_Amount	0%
DT_Char_256	0%
DT_Number_10	0%
DT_Firstname	0%
DT_Address	0%
DT_Time	0%
DT_Name	0%
DT_TimeOfDay	0%
DT_RaceType	0%
DT_Level	0%
DT_CustomerNumber	0%
DT_Date	0%
DT_Race	0%
DT_CourseNumber	0%
DT_CourseType	0%
DT_PhoneNumber	0%
DT_RaceTime	0%
DT_Age	0%
DT_Employment	0%
DT_Gender	0%

(b) Set 2

Figure 85: Logical data type coverage from the coverage report of Set 1 and 2


## A.2.2 Report for Set 5 and 6

**Coverage of the analysis model:**

<b>Global model coverage</b>	<b>58%</b>	
Mean coverage: <b>Use Case</b>	<b>80%</b>	
Mean coverage: <b>Logical Data Types</b>	<b>38%</b>	
Mean coverage: <b>Dialogs</b>	<b>54%</b>	

(a) Set 5

**Coverage of the analysis model:**

<b>Global model coverage</b>	<b>27%</b>	
Mean coverage: <b>Use Case</b>	<b>80%</b>	
Mean coverage: <b>Logical Data Types</b>	<b>0%</b>	
Mean coverage: <b>Dialogs</b>	<b>0%</b>	

(b) Set 6

Figure 86: Global coverage from the coverage report of Set 5 and 6

**Use case coverage:**

Use Case	Coverage
<b>Search_Customer</b> Test Cases: TC_1	100%
<b>Search_Course</b>	0%
<b>Book_Attendee_on_Course</b> Test Cases: TC_1 TC_2 TC_3 TC_4	100%
<b>Save_Attendee</b> Test Cases: TC_1	100%
<b>Create_customer</b> Test Cases: TC_1	100%

(a) Set 5

**Use case coverage:**

Use Case	Coverage
<b>Search_Customer</b> Test Cases: TC_1	100%
<b>Search_Course</b>	0%
<b>Book_Attendee_on_Course</b> Test Cases: TC_1 TC_2 TC_3 TC_4	100%
<b>Save_Attendee</b> Test Cases: TC_1	100%
<b>Create_customer</b> Test Cases: TC_1	100%

(b) Set 6

Figure 87: Use case coverage from the coverage report of Set 5 and 6

Dialog element	Coverage	Dialog element	Coverage
Age	100%	Age	0%
Phone	100%	Phone	0%
CourseUntil	100%	CourseUntil	0%
Attendee_Mr	100%	Attendee_Mr	0%
LevelOfCourse	100%	LevelOfCourse	0%
Attendee_Ms	0%	Attendee_Ms	0%
TypeOfCourse	100%	TypeOfCourse	0%
Search_course	0%	Search_course	0%
LastName	0%	LastName	0%
CourseFrom	100%	CourseFrom	0%
Course	0%	Course	0%
overbooking_no	0%	overbooking_no	0%
Course_details	0%	Course_details	0%
SaveAttendee	0%	SaveAttendee	0%
Search_Customer	0%	Search_Customer	0%
Address	100%	Address	0%
FirstName	0%	FirstName	0%
overbooking_yes	0%	overbooking_yes	0%
BookAttendee	0%	BookAttendee	0%
IsBooked	0%	IsBooked	0%

(a) Dialog elements - Set 5

(b) Dialog elements - Set 6

Dialog action	Coverage	Dialog action	Coverage
Select_Title	0%	Select_Title	0%
Search_Course	100%	Search_Course	0%
Add_Course	0%	Add_Course	0%
Search_Customer	100%	Search_Customer	0%
Select_Course	100%	Select_Course	0%
overbooking	100%	overbooking	0%
Book_Attendee	100%	Book_Attendee	0%
Save_Attendee	100%	Save_Attendee	0%
Select_CourseUnit	0%	Select_CourseUnit	0%

(c) Dialog actions - Set 5

(d) Dialog actions - Set 6

Figure 88: Dialog coverage from the coverage report of Set 5 and 6

Coverage of the logical data type model:

Data Type	Coverage
DT_SkiPiste	0%
DT_Count	0%
DT_Surname	100%
DT_CourseName	0%
DT_Amount	0%
DT_Char_256	0%
DT_Number_10	0%
DT_Firstname	100%
DT_Address	100%
DT_Time	0%
DT_Name	0%
DT_TimeOfDay	0%
DT_RaceType	0%
DT_Level	100%
DT_CustomerNumber	0%
DT_Date	100%
DT_Race	0%
DT_CourseNumber	0%
DT_CourseType	100%
DT_PhoneNumber	100%
DT_RaceTime	0%
DT_Age	100%
DT_Employment	0%
DT_Gender	100%

(a) Set 5

Coverage of the logical data type model:

Data Type	Coverage
DT_SkiPiste	0%
DT_Count	0%
DT_Surname	0%
DT_CourseName	0%
DT_Amount	0%
DT_Char_256	0%
DT_Number_10	0%
DT_Firstname	0%
DT_Address	0%
DT_Time	0%
DT_Name	0%
DT_TimeOfDay	0%
DT_RaceType	0%
DT_Level	0%
DT_CustomerNumber	0%
DT_Date	0%
DT_Race	0%
DT_CourseNumber	0%
DT_CourseType	0%
DT_PhoneNumber	0%
DT_RaceTime	0%
DT_Age	0%
DT_Employment	0%
DT_Gender	0%

(b) Set 6

Figure 89: Logical data type coverage from the coverage report of Set 5 and 6



---

## BIBLIOGRAPHY

---

- [AD97] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, 1997.
- [AFGC03] Anneliese Amschler Andrews, Robert B. France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for UML design models. *Software Testing, Verification Reliability*, 13(2):95–127, 2003.
- [AK03] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20:36–41, 2003.
- [BBH05] Fevzi Belli, Christof J. Budnik, and Axel Hollmann. A holistic approach to testing of interactive systems using statecharts. In *Proceedings of the 2nd South-East European Workshop on Formal Methods (SEEFMo5)*, pages 59–73, 2005.
- [BBW06] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification Reliability*, 16(1):3–32, 2006.
- [BCR94] Victor Basili, Gianluigi Caldiera, and H. Dieter Rombach. *Encyclopedia of Software Engineering*, chapter The Goal Question Metric Approach, pages 528–532. John Wiley & Sons, 1994.
- [Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.
- [Bel01] Fevzi Belli. Finite-state testing and analysis of graphical user interfaces. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 34–43, 2001.
- [Bel03] Fevzi Belli. A holistic view for finite-state modeling and testing of user interactions. Technical report, University of Paderborn, Germany, 2003.

- [BGL<sup>+</sup>07] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and Mark Utting. A subset of precise UML for model-based testing. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104. ACM, 2007.
- [BGM10] Dominik Beulen, Baris Güldali, and Michael Mlynarski. Tabellarischer Vergleich der Prozessmodelle für modellbasiertes Testen aus Managementsicht. *Softwaretechnik-Trends*, 30(2):6–9, 2010.
- [Bin99] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.
- [BJL<sup>+</sup>05] Fabrice Bouquet, Eddie Jaffuel, Bruno Legeard, Fabien Peureux, and Mark Utting. Requirements traceability in automated test generation: application to smart card software validation. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM.
- [BL02] Lionel C. Briand and Yvan Labiche. A UML-Based Approach to System Testing. Technical report, Carleton University, 2002.
- [BM08] Graham Bath and Judy McKay. *The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates*. Rocky Nook Inc., 2008.
- [BM10] Cristiano Bertolini and Alexandre Mota. A Framework for GUI Testing Based on Use Case Design. In *ICSTW '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 252–259. IEEE Computer Society, 2010.
- [BME<sup>+</sup>07] Grady Booch, Robert Maksimchuk, Michael Engel, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications, 3rd Edition*. Addison-Wesley, 2007.
- [Boe79] Barry W. Boehm. Guidelines for verifying and validating software requirements and design specification. In *Proceedings of Euro IFIP*, pages 711–719, 1979.



- [BRDG<sup>+</sup>08] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer Verlag, 2008.
- [Bri96] Sjaak Brinkkemper. Method engineering: Engineering of information systems development methods and tools. *Information and Software Technology*, 4:275–280, 1996.
- [BRR<sup>+</sup>06] M. Busch, Dai Zhen Ru, Chaparadza R., Hoffmann A., Lacmene L., Ngwangwen T., Ndem G. C., Ogawa H., Serbanescu D., Ina Schieferdecker, and Justyna Zander-Nowicka. Model transformer for test generation from test models. In *9th International Conference on Quality Engineering in Software Technology*, 2006.
- [BS81] Stanley Burris and H.P. Sankappanavar. *A Course in Universal Algebra*. Springer-Verlag, 1981.
- [Coc01] Alistair Cockburn. *Writing effective use cases*. Addison-Wesley, 2001.
- [CPT<sup>+</sup>06] T. Chen, Pak-Lok Poon, Sau-Fun Tang, T. Tse, and Yuen Tak Yu. Applying testing to requirements inspection for software quality assurance. *Information Systems Control Journal*, 2006.
- [CS08] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electronic Notes in Theoretical Computer Science*, 195:171–188, 2008.
- [CSHo3] Ian Craggs, Manolis Sardis, and Thierry Heuillard. AGEDIS Case Studies: Model-Based Testing in Industry. In *Presented at the 1st European Conference on Model Driven Software Engineering*, Nuremberg, 2003.
- [CYXZ05] Wei Chen, Qun Ying, Yunzhi Xue, and Chen Zhao. *Unifying the Software Process Spectrum*, chapter Software Testing Process Automation Based on UTP - A Case Study, pages 222–234. Springer-Verlag Berlin Heidelberg, 2005.
- [Daio4] Zhen Ru Dai. Model-Driven Testing with UML 2.0. In *2nd European Workshop on Model Driven Architec-*

ture with an emphasis on Methodologies and Transformations, 2004.

- [Daio6] Zhen Ru Dai. *An Approach to Model-Driven Testing - Functional and Real-Time Testing with UML 2.0, U2TP and TTCN-3*. PhD thesis, Fraunhofer FOKUS, 2006.
- [DGNP04] Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen, and Holger Pals. From Design to Test with UML – Applied to a Roaming Algorithm for Bluetooth Devices. In *Testing of Communicating Systems. Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems (TestCom2004), Oxford, United Kingdom, March 2004. Lecture Notes in Computer Science 2978*. Springer Verlag, 2004.
- [DM03] Christian Denger and Maricel Mora. Test case derived from Requirement Specification. Technical report, Fraunhofer IESE, 2003.
- [DNSV<sup>+</sup>08] Arilo Dias Neto, Rajesh Subramanyan, Marlon Vieira, Guilherme Horta Travassos, and Forrest Shull. Improving Evidence about Software Technologies: A Look at Model-Based Testing. *IEEE Softw.*, 25(3):10–13, 2008.
- [DNSVT07] Arilo Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme Tracassos. A Survey on Model-based Testing Approaches: A Systematic Review. Technical report, Siemens Corporate Research, 2007.
- [DNT09] Arilo Claudio Dias Neto and Guilherme Horta Travassos. Model-based testing approaches selection for software projects. *Information and Software Technology*, 51(11):1487–1504, November 2009.
- [DSWO04] Dong Polo Deng, Phillip C. Y. Sheu, Taehyung Wang, and Akira K. Onoma. Model-based testing and maintenance. In *ISMSE '04: Proceedings of the IEEE Sixth International Symposium on Multimedia Software Engineering*, pages 278–285, Washington, DC, USA, 2004. IEEE Computer Society.
- [EFWo1] Ibrahim El-Far and James Whittaker. *Encyclopedia of Software Engineering*, chapter Model-Based Soft-

- ware Testing, pages 825–837. John Wiley & Sons, 2001.
- [EHRSo2] Gregor Engels, Jan Hendrik Hausmann, Heckel. Reiko, and Stefan Sauer. Testing the Consistency of Dynamic UML Diagrams. In *Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA (USA)*, 2002.
- [EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification*. Springer-Verlag Berlin Heidelberg, 1990.
- [ES10] Gregor Engels and Stefan Sauer. A Meta-Method for Defining Software Engineering Methods. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 411–440. Springer Berlin / Heidelberg, 2010.
- [Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.
- [Far10] Qurat-ul-ann Farooq. A Model Driven Approach to Test Evolving Business Process based Systems. In *Proceedings of the MODELS 2010 Doctoral Symposium*, pages 19–24, 2010.
- [Fic10] Andreas Fichter. Messung und Bewertung der Modellabdeckung anhand der Traceability-Informationen eines Modelltransformation-sprozesses. Master thesis, Hochschule Furtwangen University, Fakultät Wirtschaftsinformatik, Studiengang Application Architectures, September 2010.
- [FKN<sup>+</sup>92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L Finkelstein, and Michael Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

- [FLo2] Falk Fraikin and Thomas Leonhardt. SeDiTeC - testing based on sequence diagrams. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on Automated Software Engineering*, pages 261–266, 2002.
- [FS05] Mario Friske and Holger Schlingloff. Von Use Cases zur Test Cases: Eine systematische Vorgehensweise. In *Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme, MBEES 2005*, pages 1–10, 2005.
- [GECM<sup>+</sup>09] J. Javier Gutiérrez, María Jos´ Escalona Cuaresma, Manuel Mejías, Isabel Ramod, and Joaquín Torres. An approach for Model-Driven test generation. In *Proceedings of the IEEE International Conference on Research Challenges in Information Science, RCIS 2009*, pages 303–312. IEEE, 2009.
- [GECMT05] J.Javier Gutiérrez, María Jos´ Escalona Cuaresma, Mejías Manuel, and Joaquín Torres. Generation of test cases from functional requirements. A survey. In *4th Workshop on System Testing and Validation*, 2005.
- [GEMT06] J. Javier Gutiérrez, María Jos´ Escalona, Manuel Mejías, and Joaquín Torres. Derivation of Test Objectives Automatically. In *Advances in Information Systems Development: New Methods and Practice for the Networked Society*, pages 435–446, 2006.
- [GG75] John Goodenough and Susan Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 2:156–173, 1975.
- [GJM<sup>+</sup>10] Baris Güldali, Stefan Jungmayr, Michael Mlynarski, Stefan Neumann, and Mario Winter. Starthilfe für modellbasiertes Testen. *OBJEKTSpektrum*, 3:63–69, 2010.
- [GMS10] Baris Güldali, Michael Mlynarski, and Yavuz Sancar. Effort Comparison of Model-based Testing Scenarios. In *Proceedings of 1st International Workshop on Quality of Model-Based Testing (QuoMBaT 2010)*, pages 28–36, Paris, France, 2010.
- [GMWE09] Baris Güldali, Michael Mlynarski, Andreas Wübbeke, and Gregor Engels. Model-Based System Testing Using Visual Contracts. In *Proceedings*

of Euromicro SEAA Conference 2009, Special Session on Model Driven Engineering, pages 121–124, Washington, DC, USA, 2009. IEEE Computer Society.

- [GPRo6] Volker Gruhn, Daniel Pieper, and Carsten Röttgers. *MDA*. Springer, 2006.
- [GSDo5] Hans-Gerhard Gross, Ina Schieferdecker, and George Din. Model-based built-in tests. *Electronic Notes in Theoretical Computer Science*, 111:161–182, 2005.
- [Hau05] Jan Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2005.
- [Hey10] Annette Heym. A model-based testing approach for business information systems. Master thesis, University of Augsburg, Faculty of applied computer science, Chair of Software Engineering and Programming Languages, February 2010.
- [HGBo8] Bill Hasling, Helmut Goetz, and Klaus Beetz. Model Based Testing of System Requirements using UML Use Case Models. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 367–376, 2008.
- [HIMoo] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. Uml-based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70, New York, NY, USA, 2000. ACM.
- [HL03] Reiko Heckel and Marc Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6):33–43, 2003.
- [HMo8] Wolfgang Hesse and Heinrich Mayr. Modellierung in der Softwaretechnik: Eine Bestandsaufnahme. *Informatik Spektrum*, 31(5):377–394, 2008.
- [HN03] Alan Hartman and Kenneth Nagin. Model driven testing - AGEDIS architecture interfaces and tools. In *1st European Conference on Model Driven Software Engineering*, pages 1–11, 2003.

- [HN04] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. *SIGSOFT Software Engineering Notes*, 29(4):129–132, 2004.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
- [HT05] Wolfgang Hesse and Thomas Tilley. Formal concept analysis used for software analysis and modelling. In *Formal Concept Analysis*, number 3626 in Lecture Notes in Computer Science, pages 259–282. Springer Berlin / Heidelberg, 2005.
- [HVFR05] Jean Hartmann, Marlon Vieira, Herbert Foster, and Axel Ruder. A UML-based approach to system testing. *Innovations in Systems and Software Engineering*, 1:12–24, 2005.
- [IEE98] IEEE Computer Society. IEEE Recommended Practice for Software Requirements Specifications. <http://standards.ieee.org/findstds/standard/830-1998.html>, 1998.
- [IEE08] IEEE Computer Society. IEEE Standard for Software and System Test Documentation. <http://standards.ieee.org/findstds/standard/829-2008.html>, 2008.
- [ISBP07] Timea Illes-Seifert, Lars Borner, and Barbara Paech. Testfallgenerierung aus semi-formalen Use Cases. In *Informatik 2007. Informatik trifft Logistik. Band 2. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, pages 404–409, 2007.
- [ISO04] ISO/IEC Standard No. 9126: Software engineering - Product quality; Parts 1-4, 2001-2004.
- [ISO07] ISO: ISO/IEC 24774:2007 Software engineering - metamodel for development methodologies, 2007.
- [IST] ISTQB Glossary of Testing Terms Version 2.1.
- [Jac92] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [Jou05] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference*

- on Model Driven Architecture ECMDA workshop on traceability Nuremberg Germany*, pages 29–37, 2005.
- [Jun99] Stefan Jungmayr. Reviewing software artifacts for testability. In *EuroSTAR99*, November 1999.
- [Kru03] Philippe Kruchten. *The Rational Unified Process. An Introduction*. Addison-Wesley Object Technology Series, 2003.
- [KSWo1] Georg Kösters, Hans-Werner Six, and Mario Winter. Coupling use cases and class models as a means for validation and verification of requirements specifications. *Requirements Engineering*, 6:3–17, 2001.
- [KVdABVo8] Tim Koomen, Leo Van der Aalst, Bart Broekman, and Michiel Vroon. *TMap Next: Ein praktischer Leitfaden für ergebnisorientiertes Softwaretesten*. dpunkt.verlag, 2008.
- [Lego8] Bruno Legeard. Model-Based Testing of a Financial Application - A Case Study. In *Presentation at the EuroStar 2008 Conference*, 2008.
- [Lig09] Peter Liggesmeyer. *Software Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2009.
- [LMdG<sup>+</sup>09] Beatriz Pérez Lamancha, Pedro Reales Mateo, Ignacio Rodríguez de Guzmán, Macario Polo Usaola, and Mario Piattini Velthius. Automated model-based testing using the UML testing profile and QVT. In *MoDeVva '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.
- [Loho6] Marc Lohmann. *Kontraktbasierte Modellierung, Implementierung und Suche von Komponenten in serviceorientierten Architekturen*. PhD thesis, University of Paderborn, 2006.
- [LSSo7] Tilo Linz, Hans Schäfter, and Andreas Spillner. *Software Testing Foundations: A Study Guide for the Certified Tester Exam - Foundation Level - ISTQB compliant*. Rocky Nook, 2007.

- [MBNo3] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society, 2003.
- [McC76] Thomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.
- [McMo4] Phil McMinn. Search-based software test data generation: A survey. *ACM Software Testing, Verification & Reliability*, 14:105–156, 2004.
- [MGSE09] Michael Mlynarski, Baris Güldali, Melanie Späth, and Gregor Engels. From Design Models to Test Models by Means of Test Ideas. In *MoDeVVA '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10, New York, NY, USA, 2009. ACM.
- [MJV<sup>+</sup>10] Qaisar A. Malik, Antti Jaaskelainen, Heikki Virtanen, Mika Katara, Fredrik Abbors, Dragos Truscan, and Johan Lilius. Model-Based Testing Using System vs. Test Models - What Is the Difference? *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:291–299, 2010.
- [MLLF09] Anders Mattsson, Björn Lundell, Brian Lings, and Brian Fitzgerald. Linking Model-Driven Development and Software Architecture: A Case Study. *IEEE Transactions on Software Engineering*, 35(1):83–93, 2009.
- [Mly10] Michael Mlynarski. Holistic Model-Based Testing for Business Information Systems. In *Proceedings of 3rd International Conference on Software Testing, Verification and Validation*, pages 327–330, Paris, France, April 2010. IEEE.
- [MN10] Atif Memon and Bao N. Nguyen. Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End. *Advances in Computers*, 80:121–162, June 2010.
- [Moo56] Edward Forrest Moore. Gedanken-experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.



- [MQPo5] Jacqueline A. Mc Quillan and James F. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report, Department of Computer Science, 2005.
- [MSPo1] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267. ACM, 2001.
- [MVGVKo6] Tom Mens, Pieter Van Gorp, Dániel Varró, and Gábor Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, 152:143–159, 2006.
- [MXXo6] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for UML activity diagrams. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 2–8. ACM, 2006.
- [NFTJo6] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jazeque. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32:140–155, 2006.
- [Nieo9] Benjamin Niebuhr. Test case generation from UML models described with the UML Testing Profile. Bachelor thesis, University of Applied Sciences Brandenburg, Faculty for Applied Computer Science, December 2009.
- [NRPo5] Pedro Santos Neto, Rodolfo Resende, and Clarindo Padua. A method for information systems testing automation. In Oscar Pastor and Joao Falcao e Cunha, editors, *CAiSE 2005: Advanced Information Systems Engineering*, pages 504–518. Springer Berlin / Heidelberg, 2005.
- [NZRo7] Leila Naslavsky, Hadar Ziv, and Debra J. Richardson. Towards traceability of model-based testing artifacts. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 105–114. ACM, 2007.

- [OA99] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In *UML'99 - The Unified Modeling Language*, volume 1723, pages 416–429. Springer Berlin / Heidelberg, 1999.
- [Obj03] Object Management Group. MDA Guide Version 1.0.1. <http://www.omg.org/cgi-bin/doc?omg/03-06-01>, June 2003.
- [Obj06a] Object Management Group. Meta-Object Facility (MOF) Specification, Version 2.0. <http://www.omg.org/spec/MOF/2.0/>, 2006.
- [Obj06b] Object Management Group. Object Constraint Language Version 2.0. <http://www.omg.org/spec/OCL/2.0/PDF>, 2006.
- [Obj07a] Object Management Group. MOF 2.0/XMI Mapping, Version 2.1.1. <http://www.omg.org/spec/XMI/2.1.1/>, 2007.
- [Obj07b] Object Management Group. UML Testing Profile Version 1.0. <http://utp.omg.org/>, 2007.
- [Obj08a] Object Management Group. Meta-Object Facility (MOF) 2.0 Query/View/Transformation Specification. <http://www.omg.org/spec/QVT/1.0/>, 2008.
- [Obj08b] Object Management Group. Software & Systems Process Engineering Meta-Model Specification, Version 2.0. <http://www.omg.org/spec/SPEM/2.0/>, 2008.
- [Obj09] Object Management Group. Unified Modeling Language Version 2.2. <http://www.omg.org/spec/UML/2.2/>, 2009.
- [Pen87] Nancy Pennington. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*, pages 100–113. Ablex Publishing Corp., 1987.
- [PJJ<sup>+</sup>07] Simon Pickin, Claude Jard, Thierry Jeron, Jean-Marc Jezequel, and Yves Le Traon. Test Synthesis from UML Models of Distributed Software. *IEEE Transactions on Software Engineering*, 33(4):252–269, 2007.

- [PKSo2] Martin Pol, Tim Koomen, and Andreas Spillner. *Management und Optimierung des Testprozesses*. dpunkt.verlag, 2002.
- [PPo5] Alexander Pretschner and Jan Philipps. *Model-Based Testing of Reactive Systems*, chapter Methodological Issues in Model-Based Testing, pages 281–291. Springer Verlag, 2005.
- [PPW<sup>+</sup>05] Alexander Pretschner, Wolfgang Prenninger, Stephan. Wagner, Christian Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and Thomas Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.
- [PR10] Klaus Pohl and Chris Rupp. *Basiswissen Requirements Engineering*. dpunkt.verlag, 2010.
- [PWGWo8] Christian Pfaller, Stefan Wagner, Jörg Gericke, and Matthias Wiemann. Multi-Dimensional Measures for Test Case Quality. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08*, pages 364–368, 2008.
- [QJ09] Siyou Qian and Fan Jiang. An event interaction structure for GUI test case generation. In *International Conference on Computer Science and Information Technology*, pages 619–622, 2009.
- [RBGW10] Thomas Roßner, Christian Brandes, Helmut Götz, and Mario Winter. *Basiswissen Modellbasierter Test*. dpunkt.verlag, 2010.
- [RJ01] Balasubramaniam Ramesh and Matthias Jarke. Towards reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27:58–93, 2001.
- [RS05] Hridesh Rajan and Kevin Sullivan. Generalizing AOP for Aspect-Oriented Testing. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 14–18. ACM, 2005.
- [Rum03] Bernhard Rumpe. Model-based testing of object-oriented systems. In *Formal Methods for Components and Objects, International Symposium, FMCO*

- 2002, Leiden. LNCS 2852, pages 380–402. Springer Verlag, 2003.
- [SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.
- [SE09] Michael Spijkerman and Tobias Eckardt. Modellbasiertes Testen auf Basis des fundamentalen Testprozesses. *Softwaretechnik-Trends*, 29(4):5–8, 2009.
- [Shao3] Mary Shaw. Writing good software engineering research papers: minitutorial. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 726–736. IEEE Computer Society, 2003.
- [SK06] Motoshi Saeki and Haruhiko Kaiya. On relationships among models, meta models, and ontologies. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006.
- [SL05] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus und Weiterbildung zum Certified Tester*. dpunkt.verlag, 2005.
- [SQL08] ISO: ISO/IEC 9075 :2008 Information technology – Database languages – SQL, 2008.
- [SS97] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, pages 80–88, 1997.
- [SSE09] Frank Salger, Stefan Sauer, and Gregor Engels. Integrated specification and quality assurance for large business information systems. In *ISEC '09: Proceeding of the 2nd annual conference on India software engineering conference*, pages 129–130, New York, NY, USA, 2009. ACM.
- [TECMG09] Arturo Henry Torres, María Jos´ Escalona Cuaresma, Mejías Manuel, and J.Javier Gutiérrez. A MDA-Based Testing. A comparative study. In *Proceeding of 4th international conference on Software and Data Technologies ICSOFT 2009*, pages 269–274. INSTICC Press, 2009.

- [TTC05] ETSI Standard ES 201 873-1 V3.1.1 (2005-06): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2005.
- [UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.
- [UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report Technical Report ISSN 1170-487X, The University of Waikato, 2006.
- [VGE08] Hendrik Voigt, Baris Güldali, and Gregor Engels. Quality Plans for Measuring the Testability of Models. In *Proceedings of the 11th International Conference on Quality Engineering in Software Technology (CONQUEST 2008), Potsdam (Germany)*, pages 353–370. dpunkt.verlag, 2008.
- [VLH<sup>+</sup>06] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of GUI testing using a model-driven approach. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14. ACM, 2006.
- [Wag06] Stephan Wagner. A Literature Survey on the Quality Economics of Defect Detection Techniques. Technical report, Institut für Informatik, Technische Universität München, July 2006.
- [Wei09] Stephan Weißleder. Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, pages 211–225. Springer, 2009.
- [Win99] Mario Winter. *Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen die Anforderungsspezifikation*. PhD thesis, University of Hagen, Department of Computer Science, 1999.
- [WL10] Stephan Weißleder and Hartmut Lackner. System Models vs. Test Models - Distinguishing

- the Undistinguishable? In *GI Jahrestagung (2)'10*, pages 321–326, 2010.
- [WRH<sup>+</sup>99] Claes Wohlin, Per Runeson, Martin Höst, Anneliese von Mayrhauser, Björn Regnell, Anders Wesslén, and Magnus Ohlsson. *Experimentation in Software Engineering: An Introduction*. Springer, Berlin, 1999.
- [Xie06] Qing Xie. Developing cost-effective model-based techniques for gui testing. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 997–1000, 2006.
- [XMo8] Qing Xie and Atif M Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):1–35, 2008.
- [ZVS<sup>+</sup>07] Benjamin Zeiß, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In *Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI) 105. Copyright Gesellschaft für Informatik*, pages 231–242. Köllen Verlag, Bonn, 2007.