



# Testing End-To-End Chains using Domain Specific Languages

Tobias Hartmann

Dissertation  
zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften  
—Dr.-Ing.—

Vorgelegt im Fachbereich 3 (Mathematik und Informatik)  
der Universität Bremen  
im Januar 2015

---

1. Gutachter  
Prof. Dr. Jan Peleska

2. Gutachter  
Prof. Dr. Martin Gogolla

Promotionskolloquium  
25. November 2015

Der Verfasser erklärt, dass er die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat.

Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind ausnahmslos nach bestem Wissen und Gewissen als solche kenntlich gemacht.

Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

---

Bremen, Januar 2015

# Contents

<b>Contents</b>	<b>iv</b>	
<b>List of Figures</b>	<b>xii</b>	
<b>List of Tables</b>	<b>xvi</b>	
<b>Listings</b>	<b>xvii</b>	
<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
2.1	Current Situation . . . . .	3
2.2	The V-Model . . . . .	4
2.3	The Solution . . . . .	6
<b>3</b>	<b>Methodological and Technical Foundations</b>	<b>9</b>
3.1	Definitions of DSL . . . . .	9
3.2	Advantages . . . . .	10
3.3	Disadvantages . . . . .	11
3.4	DSL Examples . . . . .	12
3.5	Visual and Textual Languages . . . . .	13
3.5.1	Visual Language Design . . . . .	15
3.5.2	Visual Notation . . . . .	18
3.5.2.1	Principles . . . . .	19
3.5.2.2	Semiotic Clarity . . . . .	20
3.5.2.3	Perceptual Discriminability . . . . .	21
3.5.2.4	Visual Distance . . . . .	22
3.5.2.5	Perceptual Popout . . . . .	23
3.5.2.6	Icons . . . . .	24
3.5.2.7	Redundant Coding . . . . .	25
3.5.3	Diagrams . . . . .	25
3.5.4	Evaluation of a Visual Notation . . . . .	26
3.6	Guidelines . . . . .	28

---

3.6.1	Literature . . . . .	30
3.6.2	Views . . . . .	30
3.6.3	General Guidelines . . . . .	33
3.6.3.1	Friendly . . . . .	35
3.6.3.2	Unfriendly . . . . .	36
3.6.4	Pattern . . . . .	36
3.6.5	Grid . . . . .	37
3.6.6	Layout . . . . .	37
3.6.6.1	Graph . . . . .	37
3.6.6.2	Object . . . . .	38
3.7	Influence on Productivity . . . . .	39
3.7.1	Detection of Faults . . . . .	39
3.7.2	Information and Inspection Cycles . . . . .	40
3.7.3	Studies . . . . .	41
3.8	Code and Design Smells . . . . .	43
3.8.1	Smells and Design Practice . . . . .	44
3.8.2	Coding Style Guides . . . . .	45
3.8.3	Why does code matter even if it is generated? . . . . .	46
3.9	DSL Phases . . . . .	48
3.9.1	Analysis Phase . . . . .	49
3.9.2	Design Phase . . . . .	50
3.9.3	Implementation Phase . . . . .	50
3.9.4	Use Phase . . . . .	52
3.10	Evolution of DSL . . . . .	52
3.11	Syntax and Semantics . . . . .	53
3.11.1	Well-Formed Model . . . . .	54
3.11.2	Syntax . . . . .	55
3.11.2.1	Abstract Syntax . . . . .	56
3.11.2.1.1	Properties . . . . .	56
3.11.2.1.2	Objects . . . . .	56
3.11.2.1.3	Relationships . . . . .	56
3.11.2.1.4	Roles . . . . .	57
3.11.2.1.5	Ports . . . . .	57
3.11.2.1.6	Graphs . . . . .	57
3.11.2.2	Concrete Syntax . . . . .	57
3.11.3	Semantics . . . . .	58
3.11.4	Constraints . . . . .	59
3.11.4.1	Regular Expressions . . . . .	60
3.11.4.2	Roles and Relationships . . . . .	61
3.11.4.3	Templates . . . . .	61
3.11.4.4	Predefined Lists . . . . .	62
3.11.4.5	Derivation Of Information . . . . .	62
3.11.5	Syntax and Semantics Checks . . . . .	63

---

3.12	Versioning . . . . .	64
3.12.1	Versioning of Meta-Models . . . . .	65
3.12.2	Versioning of Models . . . . .	65
3.12.3	Versioning of Code Generators . . . . .	65
3.12.4	Versioning of Code . . . . .	66
3.12.5	Achieving Versioning . . . . .	66
3.12.6	Model Compare . . . . .	68
3.12.6.1	First Scenario . . . . .	69
3.12.6.2	Second Scenario . . . . .	69
<b>4</b>	<b>Workflow</b>	<b>71</b>
4.1	Generic View . . . . .	71
4.2	Detailed View . . . . .	72
4.3	Development Process . . . . .	75
4.3.1	The “Standard” Development Process . . . . .	75
4.3.2	Documents and Requirements . . . . .	76
4.3.2.1	Inconsistency between Documents . . . . .	76
4.3.2.2	Relations between Model and Code . . . . .	77
4.3.2.3	Relations between Model and Requirements . . . . .	78
4.3.3	Solution . . . . .	79
4.3.3.1	Mapping of Requirements . . . . .	80
4.3.3.2	Documentation . . . . .	81
4.3.4	Requirements . . . . .	82
4.3.4.1	Door-Status-Controller Example . . . . .	83
<b>5</b>	<b>E-Cab Project</b>	<b>87</b>
5.1	Sub-Projects . . . . .	87
5.2	Location . . . . .	88
5.3	Testing Activities . . . . .	89
5.3.1	Test Scenario . . . . .	91
5.3.2	Test Bench . . . . .	91
5.3.3	Testing . . . . .	92
5.3.3.1	Objectives . . . . .	92
5.3.3.2	Model based Solution . . . . .	93
5.4	E-Cab Systems . . . . .	94
5.4.1	Check-In System . . . . .	94
5.4.2	Profile Data Store . . . . .	95
5.4.3	Notification System . . . . .	96
5.4.4	Baggage Systems . . . . .	97
5.4.5	Position Application . . . . .	98
5.4.6	On-Board Database . . . . .	101
5.4.7	Caterer . . . . .	101

---

<b>6</b>	<b>E-Cab Domain Specific Language</b>	<b>103</b>
6.1	Tool Support . . . . .	104
6.2	Main Scenario Graph . . . . .	104
6.2.1	Terminal Domain Object . . . . .	107
6.2.2	Caterer Domain Object . . . . .	107
6.2.3	Airport Domain Object . . . . .	108
6.2.4	Baggage Domain Object . . . . .	110
6.2.5	Aircraft Domain Object . . . . .	110
6.2.6	Scenario Connection Types . . . . .	111
6.3	Terminal Domain Graph . . . . .	111
6.3.1	Passenger Object . . . . .	112
6.3.2	Meal Database Object . . . . .	115
6.3.3	Flight Database Object . . . . .	116
6.3.4	Terminal Domain Connection Types . . . . .	117
6.3.5	Terminal Domain Checks . . . . .	118
6.4	Caterer Domain Graph . . . . .	118
6.4.1	Drink Item Object . . . . .	118
6.4.2	Meal Item Object . . . . .	120
6.4.3	Item Box Object . . . . .	120
6.4.4	Caterer Flight Object . . . . .	121
6.4.5	Caterer Domain Connection Types . . . . .	122
6.4.6	Caterer Domain Checks . . . . .	123
6.5	Baggage Domain Graph . . . . .	123
6.5.1	Baggage Application Object . . . . .	124
6.5.2	Baggage Domain Connection Types . . . . .	124
6.5.3	Baggage Domain Checks . . . . .	125
6.6	Airport Domain Graph . . . . .	125
6.6.1	Airport Layout Object . . . . .	126
6.6.2	Test Bench Configuration Object . . . . .	126
6.7	Airport Layout Graph . . . . .	127
6.7.1	Horizontal Border Object . . . . .	129
6.7.2	Vertical Border Object . . . . .	130
6.7.3	Startpoint Object . . . . .	130
6.7.4	Endpoint Object . . . . .	130
6.7.5	Airport Location Object . . . . .	131
6.7.6	Airport Waypoint Object . . . . .	133
6.7.7	Passenger Move Object . . . . .	134
6.7.8	Airport Area Object . . . . .	136
6.7.9	Airport Layout Info Object . . . . .	137
6.7.10	Airport Layout Connection Types . . . . .	138
6.7.11	Airport Layout Checks . . . . .	139
6.8	System Deployment and Configuration Graph . . . . .	141
6.8.1	System Deployment and Configuration Object . . . . .	142

---

6.8.2	System Deployment and Configuration Checks . . . . .	143
6.9	Behavior Graph . . . . .	143
6.9.1	Erase Object . . . . .	144
6.9.2	Handle Incoming Message Object . . . . .	145
6.9.3	Opaque Object . . . . .	146
6.9.4	Send Message Object . . . . .	146
6.9.5	Store Message Object . . . . .	147
6.9.6	Wait for Messages Object . . . . .	147
6.9.7	Process Update Object . . . . .	148
6.9.8	Calculate Notification Object . . . . .	148
6.9.9	Wait Object . . . . .	149
6.9.10	Behavior Connection Types . . . . .	149
6.9.11	Behavior Checks . . . . .	150
6.10	Template Graph . . . . .	151
6.10.1	Database Object . . . . .	153
6.10.2	Message Object . . . . .	154
6.10.3	Data Structure Object . . . . .	155
6.10.4	Template Connection Types . . . . .	157
6.10.5	Template Checks . . . . .	157
6.11	All Domain Objects . . . . .	159
6.11.1	Note Object . . . . .	159
6.11.2	Test Bench Configuration Status Object . . . . .	160
6.11.3	Graph Information Object . . . . .	161
6.12	Generators . . . . .	162
6.12.1	MERL . . . . .	162
6.12.2	Types of Generators . . . . .	162
6.12.3	GUI Enhancements . . . . .	163
6.12.3.1	Airport Layout Enhancement . . . . .	164
6.12.4	Check Model . . . . .	166
6.12.4.1	Test Bench Configuration Status Object Check Generators . . . . .	167
6.12.4.2	Test Bench Configuration Status Object Indicators . . . . .	172
6.12.5	Model to Text . . . . .	175
6.12.5.1	Create target code . . . . .	176
6.12.5.1.1	Simulations . . . . .	176
6.12.5.1.2	Signal Definitions . . . . .	176
6.12.5.1.3	Interface Module . . . . .	177
6.12.5.1.4	Oracle . . . . .	177
6.12.5.2	Create intermediate format . . . . .	177
6.12.5.3	Create configuration . . . . .	178
6.12.6	Trigger external actions . . . . .	178
<b>7</b>	<b>Testing Scenario</b>	<b>181</b>
7.1	Testing Scenario - Location Service and Guidance System . . . . .	181



---

7.2	Airport Layout . . . . .	183
7.3	Simulated Original Equipment . . . . .	184
7.3.1	Behavior - Profile Data Store . . . . .	185
7.3.2	Behavior - On-Ground Baggage System . . . . .	185
7.3.3	Behavior - Notification System . . . . .	187
7.3.4	Behavior - Check-In System . . . . .	187
7.3.5	Behavior - A/C Baggage System . . . . .	188
7.3.6	Behavior - Position Application . . . . .	188
7.4	Position Updater Tool . . . . .	188
7.5	MetaEdit Bridge . . . . .	191
7.6	Deployment . . . . .	191
7.6.1	Deployment Test Bench . . . . .	193
7.6.2	Deployment PC . . . . .	194
<b>8</b>	<b>Test Data Generation</b>	<b>197</b>
8.1	Test Strategie . . . . .	198
8.2	E-Cab Workflow Test Data Generation . . . . .	199
8.3	Intermediate File - Airport and Passenger Data . . . . .	201
8.4	Test Data Generator Tool . . . . .	205
8.5	Passenger Movement Data . . . . .	206
<b>9</b>	<b>Creation of the Target</b>	<b>209</b>
9.1	Workflow to Target . . . . .	209
9.1.1	Prepare local environment . . . . .	211
9.1.2	Create Intermediate Files . . . . .	215
9.1.3	Create Test Bed . . . . .	217
9.1.3.1	Library Module . . . . .	217
9.1.3.2	Channel Description . . . . .	220
9.1.3.3	Signal Description . . . . .	221
9.1.3.4	Interface Modules . . . . .	223
9.1.3.4.1	IFM SMS . . . . .	224
9.1.3.4.2	IFM Notification System . . . . .	225
9.1.3.4.3	IFM MetaEdit+ . . . . .	226
9.1.3.4.4	IFM Injector Proxy . . . . .	227
9.1.3.4.5	IFM SOAP . . . . .	228
9.1.4	Create Test . . . . .	232
9.1.4.1	Oracles . . . . .	232
9.1.4.1.1	Verification of Transmitted Short Messages . . . . .	233
9.1.4.1.2	Verification of Requirements . . . . .	234
9.1.4.2	Simulated Systems . . . . .	235
9.1.4.2.1	Part 1 - Template Graph . . . . .	235
9.1.4.2.2	Part 2 - Helper Functions . . . . .	237
9.1.4.2.3	Part 3 - Behavior . . . . .	239

---

9.1.4.3	Test Data Stimulus . . . . .	247
9.1.5	Create Test Bench Configuration . . . . .	249
9.1.5.1	Test Configuration . . . . .	250
9.1.6	Create Target on Test Bench . . . . .	253
9.1.6.1	Helper Scripts . . . . .	253
9.1.6.2	Create Target . . . . .	255
<b>10</b>	<b>Test Results</b>	<b>257</b>
10.1	Course of Action . . . . .	257
10.1.1	Mrs. Jolie . . . . .	258
10.1.2	Mr. Bond . . . . .	259
10.1.3	Mr. Lennon . . . . .	259
10.1.4	Mr. Mustermann . . . . .	260
10.2	Results . . . . .	260
<b>11</b>	<b>Evaluation and Conclusion</b>	<b>263</b>
11.1	Approach . . . . .	263
11.2	Implementation . . . . .	264
11.3	Model Checks . . . . .	265
11.4	Organization of Generators . . . . .	266
11.5	Handling of Generators . . . . .	267
11.6	Variables . . . . .	268
11.7	Evolution of Objects . . . . .	268
11.8	Involvement of Stakeholders . . . . .	270
11.9	Test Run . . . . .	270
11.10	Conclusion . . . . .	270
<b>12</b>	<b>Outlook</b>	<b>273</b>
12.1	Extending Model Checks . . . . .	273
12.2	Detection of Faults during Run-Time . . . . .	273
12.3	Avoiding Code Smells . . . . .	274
12.4	Integration to Requirements Database . . . . .	274
12.5	Change Protocols . . . . .	275
12.6	Web Server . . . . .	275
12.7	Principle of Data Avoidance . . . . .	276
12.8	Temporary ID . . . . .	276
12.9	Code Generator . . . . .	276
12.10	Test Data Generation Improvements . . . . .	277
12.11	Design of the DSL . . . . .	278
<b>13</b>	<b>Appendix</b>	<b>279</b>
13.1	MetaEdit . . . . .	279
13.1.1	Extensions and Patches . . . . .	279

---

13.1.2	Start MetaEdit+ . . . . .	279
13.1.3	MetaEdit Bridge Commands . . . . .	280
13.1.4	MERL Reference . . . . .	283
13.2	SOAP Message Examples . . . . .	286
13.2.1	Register Profile Data Store Message . . . . .	286
13.2.2	Unregister Profile Data Store Message . . . . .	288
13.2.3	Check-In Passenger Message . . . . .	288
13.2.4	Check-Out Passenger Message . . . . .	289
13.2.5	Load Baggage Message . . . . .	290
13.2.6	Unload Baggage Message . . . . .	290
13.2.7	Load A/C Baggage Message . . . . .	291
13.2.8	Unload A/C Baggage Message . . . . .	291
13.2.9	Register Position Application Message . . . . .	291
13.2.10	Update Airport Location Message . . . . .	292
13.2.11	Inform Passenger Message . . . . .	292
13.3	Created Files . . . . .	293
13.3.1	Overview Created Files . . . . .	293
13.3.2	Signal Definition Generator Files . . . . .	295
13.3.3	Configuration Files . . . . .	296
13.3.3.1	Test Bench Project Configuration File . . . . .	296
13.3.3.2	Test Bench Test Configuration File . . . . .	297
13.3.4	Intermediate Format Files . . . . .	301
13.3.4.1	PAX Movement Test Data . . . . .	301
13.3.4.2	Airport Intermediate Format File . . . . .	301
13.3.4.3	Position Update Tool Intermediate Format File . . . . .	314
	<b>Bibliography</b>	<b>317</b>
	<b>Glossary</b>	<b>325</b>

# List of Figures

2.1	The V-Model . . . . .	5
3.1	Visual Information Processing . . . . .	13
3.2	The Solution Space . . . . .	14
3.3	Visual Dialects . . . . .	16
3.4	Semiotic Clarity . . . . .	20
3.5	Perceptual Popout . . . . .	23
3.6	Perceptual Popout: Combination of Visual Variables . . . . .	24
3.7	Hierarchies . . . . .	32
3.8	Views on the Model . . . . .	33
3.9	Preference Shape Ratio . . . . .	38
3.10	Cumulative fault discovery rate across the development life cycle . . . . .	41
4.1	Generic Workflow . . . . .	72
4.2	Detailed Workflow . . . . .	73
4.3	Relations between Model and Code . . . . .	78
4.4	Relations between Model and Requirements . . . . .	79
4.5	Example: Doors Controller . . . . .	84
5.1	A300 Mock-Up . . . . .	89
5.2	Floor Plan . . . . .	90
5.3	Cargo Mock-Up . . . . .	90
5.4	Test Bench Architecture . . . . .	92
5.5	Use Case - Check-In System . . . . .	94
5.6	Use Case - Profile Data Store . . . . .	95
5.7	Use Case - Notification System . . . . .	97
5.8	Use Case - On-Ground Baggage System . . . . .	97
5.9	Use Case - A/C Baggage System . . . . .	98
5.10	Use Case - Position Application . . . . .	99
5.11	Boarding Ticket . . . . .	100
5.12	RFID Chip implemented in the Boarding Ticket . . . . .	100
5.13	Use Case - On-Board Database . . . . .	101
5.14	Use Case - Caterer . . . . .	102

6.1	Scenario . . . . .	105
6.2	Scenario Graph Properties . . . . .	106
6.3	Terminal Domain Properties . . . . .	107
6.4	Terminal Domain Object . . . . .	107
6.5	Caterer Domain Object . . . . .	108
6.6	Airport Domain Properties . . . . .	109
6.7	Airport Domain Object . . . . .	109
6.8	Baggage Domain Object . . . . .	110
6.9	Aircraft Domain Object . . . . .	111
6.10	Terminal Domain Graph: Passenger Objects, Database Meal and Database Flight . . . . .	112
6.11	Passenger Properties . . . . .	114
6.12	Passenger Object . . . . .	115
6.13	Meal Database Properties . . . . .	115
6.14	Menu Overlay . . . . .	116
6.15	Meal Database Object . . . . .	116
6.16	Flight Database Properties . . . . .	117
6.17	Flight Database Object . . . . .	117
6.18	Graph Domain Caterer: Meals are defined and dedicated to flights .	119
6.19	Drink Item Object . . . . .	119
6.20	Drink Item Properties . . . . .	119
6.21	Meal Item Object . . . . .	120
6.22	Meal Item Properties . . . . .	120
6.23	Item Box Properties . . . . .	120
6.24	Item Box Object (Meals Only) . . . . .	121
6.25	Item Box Object (Drinks Only) . . . . .	121
6.26	Caterer Flight Object . . . . .	122
6.27	Caterer Domain Connection . . . . .	122
6.28	Baggage Application Object . . . . .	124
6.29	Baggage Application Object is Not Defined . . . . .	125
6.30	Airport Domain Graph . . . . .	125
6.31	Airport Layout Object . . . . .	126
6.32	Test Bench Configuration Object . . . . .	126
6.33	Graph Airport Layout . . . . .	128
6.34	Airport Layout Graph Properties . . . . .	129
6.35	Horizontal Border Object . . . . .	129
6.36	Vertical Border Object . . . . .	130
6.37	Startpoint Object . . . . .	130
6.38	Endpoint Object . . . . .	131
6.39	Airport Location Properties . . . . .	131
6.40	Airport Location Object . . . . .	132
6.41	Airport Location Object with Show Properties . . . . .	133
6.42	Airport Waypoint Properties . . . . .	134

6.43	Airport Waypoint Object . . . . .	134
6.44	Passenger Move Properties . . . . .	135
6.45	Passenger Move Object . . . . .	135
6.46	Airport Area Properties . . . . .	136
6.47	Graph Airport Area . . . . .	137
6.48	Airport Layout Info Object . . . . .	138
6.49	Airport Layout Connection Types . . . . .	138
6.50	Timeline . . . . .	139
6.51	System Deployment and Configuration Graph . . . . .	141
6.52	System Deployment and Configuration Object Properties . . . . .	142
6.53	System Deployment and Configuration Object . . . . .	142
6.54	Behavior Erase Object . . . . .	145
6.55	Behavior Handle Incoming Message . . . . .	145
6.56	Behavior Opaque Object . . . . .	146
6.57	Behavior Send Message Object . . . . .	146
6.58	Behavior Store Message Object . . . . .	147
6.59	Wait For Incoming Messages Object . . . . .	148
6.60	Process Update Object . . . . .	148
6.61	Calculate Notification Object . . . . .	149
6.62	Wait Object . . . . .	149
6.63	Behavior Example . . . . .	150
6.64	Database Properties . . . . .	153
6.65	Database Object . . . . .	153
6.66	Message Properties . . . . .	154
6.67	Message Object . . . . .	155
6.68	Data Structure Properties . . . . .	156
6.69	Data Structure Object . . . . .	157
6.70	Template Connection Types . . . . .	158
6.71	Template Connection Error . . . . .	158
6.72	Note Object Properties . . . . .	159
6.73	Note Object . . . . .	159
6.74	Test Bench Configuration Status . . . . .	160
6.75	Test Bench Configuration Status - With Error Message . . . . .	160
6.76	Test Bench Generator Hierarchy . . . . .	161
6.77	Graph Information Object . . . . .	161
6.78	Test Bench Configuration Status Object - Indication Generators . . . . .	172
6.79	Test Bench Configuration Object - Red Indicator . . . . .	173
6.80	Test Bench Configuration Object - Yellow Indicator . . . . .	174
6.81	Test Bench Configuration Object - Green Indicator . . . . .	175
7.1	Test Scenario Passenger Definition . . . . .	182
7.2	Test Scenario Airport Layout and Areas . . . . .	183
7.3	Scenario Overview . . . . .	184

7.4	Behavior Graph - Profile Data Store . . . . .	186
7.5	Behavior Graph - On-Ground Baggage System . . . . .	186
7.6	Behavior Graph - Notification System . . . . .	187
7.7	Behavior Graph - CheckIn System . . . . .	188
7.8	Behavior Graph - AC Baggage System . . . . .	189
7.9	Behavior Graph - Position Application . . . . .	189
7.10	Position Injector Tool . . . . .	190
7.11	Extended Scenario Overview . . . . .	193
7.12	MetaEdit+ API Tool Settings . . . . .	195
8.1	Workflow Test Data Generation . . . . .	200
8.2	Creation of Passenger Movement Data . . . . .	201
8.3	Test Data Generation Tool - Run Example . . . . .	206
9.1	E-Cab Scenario - MetaEdit+ Generator Tree . . . . .	210
9.2	get and set environment variables - MetaEdit+ Generator Tree . . . . .	211
9.3	create local directories - MetaEdit+ Generator Tree . . . . .	214
9.4	create intermediate files, test data - MetaEdit+ Generator Tree . . . . .	215
9.5	create TB Signal Description - MetaEdit+ Generator Tree . . . . .	221
9.6	create TB Interface Modules - MetaEdit+ Generator Tree . . . . .	223
9.7	create TB Oracles - MetaEdit+ Generator Tree . . . . .	233
9.8	create TB Oracles - MetaEdit+ Generator Tree . . . . .	233
9.9	create TB Oracles - MetaEdit+ Generator Tree . . . . .	234
9.10	create TB Simulated System Variables - MetaEdit+ Generator Tree . . . . .	235
9.11	create TB Simulated System Helper Functions - MetaEdit+ Generator Tree . . . . .	237
9.12	create TB Simulated System Create Behavior Methods - MetaEdit+ Generator Tree . . . . .	239
9.13	Behavior Graph Check-In System . . . . .	241
9.14	create TB Test Stimulus - MetaEdit+ Generator Tree . . . . .	247
9.15	create TB Configuration - MetaEdit+ Generator Tree . . . . .	250
9.16	create TB Test Configuration - MetaEdit+ Generator Tree . . . . .	250
9.17	Generator Tree - createTBHelperScripts . . . . .	253
9.18	Generator Tree - createTarget . . . . .	255
10.1	Mrs. Jolie - Message on lock screen . . . . .	258
10.2	Mrs. Jolie - received messages . . . . .	258
10.3	Mrs. Jolie - disable notifications via webservice . . . . .	259
10.4	Mrs. Jolie - confirmation of settings . . . . .	259
10.5	Mr. Bond - first notification on lock screen . . . . .	259
10.6	Mr. Bond - all messages on lock screen . . . . .	260
10.7	Mr. Lennon - first and last message . . . . .	260
11.1	Organisation of Generators - First and Second Approach . . . . .	267

# List of Tables

3.1	Regular Expressions: IP Ranges . . . . .	60
3.2	Regular Expressions: Port Ranges . . . . .	61
4.1	Door Status . . . . .	84
4.2	Requirements Example . . . . .	85
7.1	Configuration Simulated Systems . . . . .	194
7.2	Configuration Webservice . . . . .	194
7.3	Configuration MetaEdit+ Entities . . . . .	195
13.1	Created Files - Intermediate Files . . . . .	293
13.2	Created Files - Channels and Signals . . . . .	293
13.3	Created Files - Interface Modules . . . . .	294
13.4	Created Files - Oracles . . . . .	294
13.5	Created Files - Simulations . . . . .	294
13.6	Created Files - Test Stimulus . . . . .	294
13.7	Created Files - Config Files . . . . .	295
13.8	Created Files - Shell Scripts . . . . .	295



# Listings

6.1	Local Script Change View . . . . .	165
6.2	Local Script sendf5key.vbs . . . . .	166
6.3	Local Script _CHECK_TBConfStatus . . . . .	167
6.4	Local Script _setInitValues . . . . .	167
6.5	Local Script _setStaticInformation . . . . .	168
6.6	Local Script _CHECK_TBConfStatus_App1 . . . . .	169
6.7	Local Script _CHECK_TBConfStatus_App2 . . . . .	170
6.8	Local Script _CHECK_TBConfStatus_App3 . . . . .	171
6.9	Local Script _CHECK_TBConfStatus_App4 . . . . .	171
6.10	Local Script Generator for Label . . . . .	173
6.11	Local Script Red Indicator . . . . .	174
6.12	Local Script Green Indicator . . . . .	174
8.1	Snippet of Airport Export - Header . . . . .	201
8.2	Snippet of Airport Export - Passenger . . . . .	202
8.3	Snippet of Airport Export - States . . . . .	203
8.4	Snippet of Airport Export - Transitions . . . . .	204
8.5	Snippet of Passenger Movement Data . . . . .	206
9.1	Local Script _getset_environment_helperfunctions . . . . .	212
9.2	Local Script _getset_environment_graph_networkinformation . . . . .	213
9.3	Local Script _getset_environment_filenames_full_local_ifm . . . . .	214
9.4	Local Script _createDirectories_TB . . . . .	214
9.5	Local Script _helper_createDirectory . . . . .	215
9.6	Local Script _createIMF4TestDataGenerator_State_AirportLocations . . . . .	216
9.7	Local Script _createTBLibraryModule_MyLibrary_RTS_getCurrentScheduler . . . . .	218
9.8	Local Script _createTBLibraryModule_MyLibrary_RTS_getCurrentTimeInSeconds . . . . .	219
9.9	Local Script _createTBInterfaceModule_SMS_RTS_fillMaps . . . . .	224
9.10	Local Script _createTBInterfaceModule_IFM_NotificationSystem_RTS_fillAccessLists . . . . .	225
9.11	Local Script _createTBInterfaceModule_IFM_MetaEdit_RTS . . . . .	226

---

9.12	Local Script <code>_createTBInterfaceModule_IFM_InjectorProxy_RTS_AM</code> . . . . .	227
9.13	Local Script <code>_createTBInterfaceModule_IFM_SOAP_RTS_fillAccessLists</code> . . . . .	228
9.14	Local Script <code>_createTBInterfaceModule_IFM_SOAP_RTS_fillAccessLists</code> . . . . .	229
9.15	Local Script <code>_insertDataStructure</code> . . . . .	229
9.16	Local Script <code>_createTBInterfaceModule_IFM_SOAP_RTS_constructLocationUpdate</code> . . . . .	231
9.17	Local Script <code>_createTBInterfaceModule_IFM_SOAP_RTS_soapInsertXMLEntity</code> . . . . .	232
9.18	Local Script <code>_createTBSimulatedSystem_createMessage</code> . . . . .	237
9.19	Local Script <code>_createTBSimulatedSystem_handleConnection</code> . . . . .	238
9.20	Local Script <code>_createTBSimulatedSystem_createBehaviourMethods</code> . . . . .	239
9.21	Local Script <code>_getMessageType</code> . . . . .	241
9.22	Local Script <code>_createTBSimulatedSystem_createBehaviourMethods_SendMessage - Section 1</code> . . . . .	242
9.23	Local Script <code>_createTBSimulatedSystem_createBehaviourMethods_SendMessage - Section 2</code> . . . . .	243
9.24	Local Script <code>_createTBSimulatedSystem_createBehaviourMethods_SendMessage - Section 3</code> . . . . .	244
9.25	Local Script <code>_pushBackStructureElements - Section 3</code> . . . . .	245
9.26	Local Script <code>_createTBSimulatedSystem_createBehaviourMethods_SendMessage - Section 4</code> . . . . .	246
9.27	Local Script <code>_createTBTestAM_PaxMove_RTS</code> . . . . .	248
9.28	Local Script <code>_createTBConfiguration_conf_am_IFM</code> . . . . .	251
9.29	Local Script <code>_createTBConfiguration_conf_am_SIM</code> . . . . .	251
9.30	Local Script <code>_createTBHelper_cleanDirectory</code> . . . . .	253
9.31	Local Script <code>_createTBHelper_compileTest</code> . . . . .	254
13.1	Local Script <code>startME.bat</code> . . . . .	280
13.2	ME Sample Session . . . . .	281
13.3	SOAP Message Register Profile Data Store . . . . .	286
13.4	SOAP Message Un-Register Profile Data Store . . . . .	288
13.5	SOAP Message Check-In Passenger . . . . .	288
13.6	SOAP Message Check-Out Passenger . . . . .	289
13.7	SOAP Message Load Baggage . . . . .	290
13.8	SOAP Message Unload Baggage . . . . .	290
13.9	SOAP Message Load A/C Baggage . . . . .	291
13.10	SOAP Message Unload A/C Baggage . . . . .	291
13.11	SOAP Message Register Position Application . . . . .	291
13.12	SOAP Message Update Airport Location . . . . .	292
13.13	SOAP Message Inform Passenger . . . . .	292
13.14	Local Script <code>_createTBSignalDescription_Global_Sigdef_HEADER</code> . . . . .	295

---

13.15Local Script _createTBSignalDescription_Global_Sigdef_RTS . .	295
13.16Local Script project.rtp . . . . .	296
13.17Local Script project.rtp . . . . .	297
13.18Passenger Movement Data . . . . .	301
13.19Airport Intermediate Format . . . . .	302
13.20Position Update Tool Intermediate Format File . . . . .	314



First, I would like to thank Prof. Dr. Jan Peleska for inspiration and advice during my Ph.D. studies and letting me be part of his work group. This gave me the opportunity to take part at interesting conferences and also enabled me to work closely with many companies so that “real-world” problems influenced my works.

Furthermore, I would like to thank Kirsten and Ulrich with whom I shared an office for many years. Ramin, thank you for so many discussions and feedback.

Thank you to all former colleagues from the E-Cab project and from Verified System who helped me in so many cases to set up the project. I appreciated to work with them.

Especially, I would like to thank my wife Julia and my sons Yale and Ian for their patience and support. Without all their help and understanding, I wouldn't have finished this work.



*“Simplicity is the final achievement.”*

*Frédéric Chopin*

*“Graphics reveal data.”*

*Edward R. Tufte*

*“Basically, it’s a bunch of shapes connected by lines.”*

*Dilbert*





---

---

# CHAPTER 1

---

## Abstract

Testing systems is a time consuming (and hence expensive) activity. Nevertheless, it is a very important and necessary step before using systems, especially safety critical systems. Therefore, many different test procedures are used: Unit Tests, Black-Box Tests, Software Integration Tests (SWI), Hardware-Software Integration Tests (HSI), Hardware-In-The-Loop Tests, just to name a few.

Especially in the avionics domain, a variety of systems and applications communicate with each other. Furthermore, they depend and rely on the received information.

However, some faults are only detected when all systems are connected and in operational mode. A new testing approach is to create model based End-To-End Chain scenarios with original and simulated equipment in any combination.

The first aim is to automatically derive test data and test cases from the model, which is defined by a Domain Specific Language (DSL). Test data generators can be attached to quickly create a variety of stimuli for the systems under test. Furthermore, the system under test can be stimulated by either original equipment — which is connected to the test bench — or the test bench can simulate equipment and create inputs for the tested systems. Any mixture of simulated and original equipment is possible and can be changed on the fly. In the end, the results from the system under test are collected. These results can then be displayed back in the model.

This method was used and improved in the project “E-Enabled Cabin (E-Cab)” in which the author was involved. Passengers traveling by plane are in the focus of this project. Complete services and service chains — ranging from booking at home with a computer, being en route using mobile devices to leaving the destination airport — are created and used by many systems communicating with each other.

A special focus is set on a guidance scenario at an airport. The user of the system signs in to a guidance and notification system which will inform him via his own digital equipment (mobile device/smart phone). The system notifies the user about his in-time status. Either he is in-time for his flight, or he needs to hurry up and proceed to the next area or he will be too late and cannot catch his flight.

The DSL itself is designed according to the comprehension of information processes. The ability of the human brain to process visual information in parallel — in contrast

to sequential processing of textual information — is described and applied in the design of the DSL and the concepts of the project.

The development of the DSL and the workflow is developed with the “real world” in mind. This means that the work fits in established workflows and enhances the current situation. As this project took place in the aircraft industry, the appropriate development standards, like DO178B and ABD100/200 (Airbus Directives ABD100/ABD200), build the foundation.

The generation of clean code is established by applying generator guidelines (through coding standards) in order to create maintainable tests and test data.

---

---

## CHAPTER 2

---

# Introduction

Testing complex networks of systems is becoming more and more relevant. Single system tests like Black-Box Tests or Hardware-In-The-Loop Tests are of course still necessary but cannot cover the whole complexity. Some failures can only be detected if the whole network of systems is taken into account. Therefore, a tool chain for testing end-to-end communication of service chains is the next step.

The author proposes a concept where test steps, test data and the configuration of the test bench are automatically derived from a model. This model must be tailored to the specific problem domain and must be able to be refined while the systems are still under development. The benefit is that the systems under test can be further developed and extended at the same time. The only task for the tester is to adjust the model and re-run the new generated test cases. No manual intervention is needed.

Also, the test strategy is defined in the model so that the same model can be used to run different kinds of tests (e.g. checking coverage or executing robustness tests). Because the logic of generating test data and test cases is done in the next step of the tool chain, the model itself does not need to be changed.

Another benefit is the easy adjustment of the model and therefore a complete change of the configuration of the test bench. When a new model is loaded, the configuration of the test bench is done automatically. No wiring needs to be changed. This improves the uptime and temporizes the real testing activities.

### 2.1 Current Situation

In many cases, requirement databases are used within large projects. In smaller projects, spreadsheets may be used. A common way is to export these requirements into a text document which is then passed to the next level. This text document is the transmittal between two layers and sometimes regarded as a maturity gate. In case of errors within the document, the database must to be updated. A baseline needs to be drawn to ensure that all parties work with the same data. The export is done again and handed over to the next layer. This loop will be executed each time a change in the requirements is necessary.

These documents are the foundation for the design of tests. This means, that the text documents must contain the requirements which must be tested. In most cases this is not sufficient and more documents are needed. For instance, the input and output signals for a system are defined within the requirements but the concrete bit patterns of the exchanged messages are described in an additional document.

The conclusion is, that all these documents must be stable and in sync. This could be a problem in larger projects with many documents which are not necessarily created or maintained within one workgroup but might be spread over different companies or even countries.

In some documents, models are used to describe aspects or give an overview of the specified system. In most cases, these models are not formally specified but used as “images” to illustrate the text. Also, these models are only screenshots from a modeling tool so that a further processing is not possible. If the model is changed due to requirement changes, the modeling tool needs to be started, the change must to be done, the screenshot must be taken and pasted into the requirement database. This leads to documents which are out of sync because the illustrations and the text might not match. Also, there is no connection between the model and the requirements as it is not possible to connect the elements of the model to the requirements. No traceability can be achieved.

## 2.2 The V-Model

Each project is set up according to one or more lifecycle models. Which of the models is used for the project is chosen at the beginning. As each model has its advantages and disadvantages, the choice is crucial for the project.

A common lifecycle model is the V-Model. This is a development model which defines phases and the transitions between them. It is used to describe a technique for a system’s development lifecycle.

The V-Model consists of two parts (see figure 2.1 *The V-Model*). The left part describes the development phase and the right part describes the validation and verification phase. The phases itself are divided into several tasks which are processed in order. Loops within the order are possible but no task is allowed to be skipped. In most cases, the result for each task is a document which is the input for the next task. These documents are part of the “maturity gate” before each task. Only if the documents satisfy the standards for this process, the work in one of the tasks is allowed to begin.

In the upper left corner of the V-Model, high-level requirements of a system are defined. Following the left wing to the bottom, these requirements are refined in

each task. In the end, the development is started. From the right lower corner to the right upper corner, the verification and validation phase takes place. This means that the product is tested and checked against all statements from the development phase.

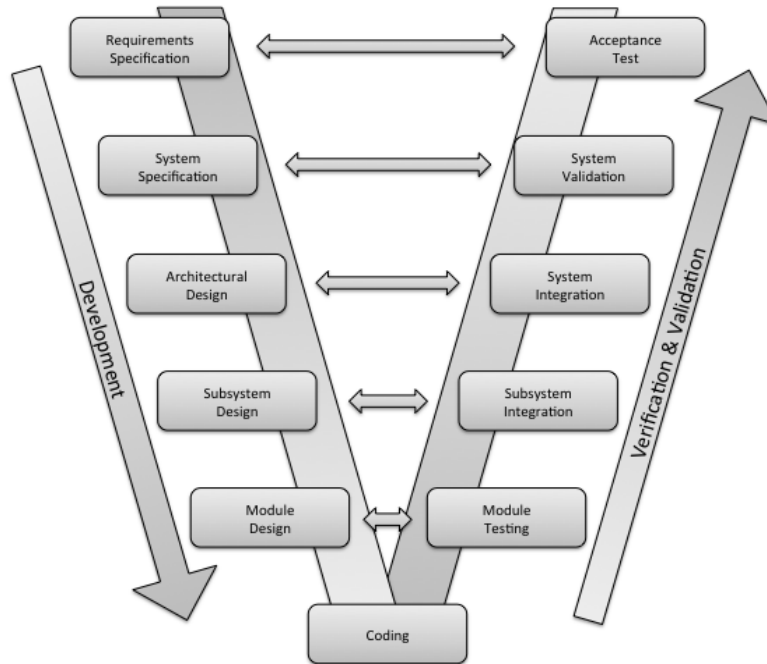


Figure 2.1: The V-Model

Within each transition between the left part of the V-Model to the right part, test procedures are executed or verification steps are applied. This means that a predefined procedure is conducted. For instance, in one task, units of the software are tested (unit tests) whereas in another task, a complete system test is executed. For each of these requirements and on all levels, a test or a verification step needs to be applied to check if all requirements are met.

Also, to prove one requirement a couple of tests can be used. It could be a one-to-many relation between requirements and the tests which prove the correct implementation. Because of traceability reasons, it is normally not wanted or allowed to have one test which covers a couple of requirements. If this is the case, the following situation could occur: there are two requirements and two tests for these requirements. The first test verifies the first requirement and the second test verifies the first and the second requirement. If the first test returns with the test result “pass” and the second test returns with the test result “fail”, the first requirement cannot be regarded as checked. In conclusion, each requirement is checked by its own tests and there is no need to act beyond these steps.

The problem exists that there could be requirements which contradict each other (mainly, this could occur on higher levels). As each requirement is tested separately, it could be that this is not detected. At the moment, there is no chance to detect this kind of error as in most cases the requirements are written down as text. There is no formalization (sometimes, the text is enriched by diagrams which are treated as graphics but with no further formal meaning). Some of these failures and errors could be spotted within the system test. In this test, system functions are tested (which are also requirement based) which are on a higher level and not broken down into single code or system aspects.

But it could be that failures are still not recognized even if all tests have been executed with the test result “pass”. In most complex systems or within networks of systems, different combinations of signals have not been taken into account especially when multi-purpose networks are used. For instance, within the avionics domain the Avionics Full-Duplex Switched Ethernet (AFDX) network is used to connect different systems. AFDX is a high-speed network which is based on ethernet and supports synchronous and asynchronous data transfers. All sent and received packets are pre-defined which means that no undefined signal could be sent or received by any application or system within the aircraft. Within each packet, messages are defined which contain one or more signals (again, messages and signals are also pre-defined). For instance, it could be that the target system is checked against all possible values of all signals and all test results are passed. A test could use one signal and stimulate the system under test by using all defined values for this signal. There could also be tests which use a combination of more than one signal. Still, it could occur that there is a combination of (maybe) independent signals within one message which trigger a state within the target system which is not specified or even defined. These kinds of errors are hard to reveal.

A possible solution are End-To-End Chain Tests, where possible data flows are taken into account. The requirements need to be formalized so that test data can be extracted automatically. These high-level tests with the original system under test and simulated environment could lead to the detection of errors which are hard to discover.

## 2.3 The Solution

One possible solution for formalized requirements are models. These models must be defined by a formalized language which is well-formed regarding syntax and semantics.

Starting with models, all kinds of data can be extracted from them. For instance, if a system is well-formed and correct, this means that all requirements are modeled

and there is no logic discrepancy, a simulation of this system could be generated. All states are defined, the incoming as well as the outgoing signals are described and their system's reactions and impacts. The model only needs to be translated into a target (for instance code or input for another tool in a tool chain).

Furthermore, if all states and the transitions between them are defined, test data for tests can be extracted. Each transition consists of a guard condition and an action which is triggered if the guard condition evaluates to be true. This means that full paths through the model (and therefore the specification) could be retrieved by placing the guard conditions with appropriate values. If the values are within the specified range, the normal behavior of the system can be tested. If the variables exceed the specified ranges, robustness tests can be executed or error conditions can be checked.

For an original system under test, a test oracle can be generated from the model. If the original system and the oracle behave exactly the same, they could be considered as equal (back-to-back test).

Also, the configuration of the test bench system can be retrieved from the formalized specifications. For instance, if the input and output signals are defined (bus system, message types, length of variables and their representation) the configuration for the hardware and signal layer can be extracted automatically. In most cases, the signals are defined on a higher level and must be mapped to hardware busses within the test systems. This mapping is retrieved and stored within the test system.

Time and money could be saved, if

- code is generated automatically
- requirements are checked within the model
- a well-formed model is ensured
- the model is used for generating tests
- the model is used for generating oracles
- quick checks of design against other systems under development can be executed
- certified code is produced
- code quality is ensured
- code standards are ensured
- design standards are ensured
- workflow within a company, for instance, Airbus Directives ABD100/ABD200, is ensured

In the following chapters, this solution is considered in detail. First, the methodological and technical foundations are described. This includes the groundwork and guidelines for domain specific languages. Furthermore, the new workflow is discussed as there are major differences in the development processes regarding traditional and model based solutions. Following this, the E-Cab project is explained on which this work is based. The resulting DSL is also described in detail. The next chapters consist of a closer look at the test scenario, the corresponding test data generation and the creation of the target (e.g. the complete test and environment on the test bench). The results are discussed afterwards and an outlook is given.



---

---

## CHAPTER 3

---

# Methodological and Technical Foundations

In this chapter, the methodological and technical foundations are explained. First, the term “Domain Specific Language (DSL)” is defined and examples of well-known DSLs are presented. The difference between graphical and textual languages is described.

In the next paragraphs, the advantages and disadvantages of such languages are depicted and a detail view on case studies is given in order to prove the author’s approach.

Also, the “physics of notation” [54] is depicted thoroughly as this is the basis for a graphical language. Furthermore, a style guide for such a language is developed: the “dos and don’ts” for good diagrams.

Furthermore, an introduction is given on code and design smells. Although, the target code might be generated automatically, the translation from model to code is an important step in case the target code is processed further or must be certified. In most cases, only validated code which applies to a certain code standard can be certified.

The process of creating and using a DSL is described, as well as the stakeholders for each of the phases. The different approaches (when and how) to develop a DSL are illustrated. This includes the evolution of a DSL, as in most cases, a language develops over time and new constructs are introduced or nuisances are eliminated.

Large models are difficult to handle and various stakeholders might need different information from a model. In these cases, views needs to be introduced. These views will make certain aspects visible and hide unnecessary data for this task.

### 3.1 Definitions of DSL

A DSL is a language which is designed for a particular problem domain. This means that constructs and techniques of the problem domain are used. By using the key concepts of the target domain, the language is tailored to the problem domain which

results in syntactical and semantical well-formed models.

“A Domain Specific Language (DSL) is designed to express the requirements and solutions of a particular business or architectural domain.” [82, p.311]

“A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.” [76, p.1]

“A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular problem domain.” [75, p.1]

Binding the language to the problem domain keeps the language rather small and simple compared to general-purpose languages, which cover all sorts of tasks. This approach enhances the productivity, maintainability and reusability of the created models.

“The key characteristic of DSLs according to this definition is their focussed expressive power.” [76, p.1]

## 3.2 Advantages

One of the most important advantages of using DSLs is the possibility to create a level of abstraction which makes understanding and maintaining a system more efficient. This is because it allows “solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs” [76, p.2].

“The power of domain-specific languages comes from the abstractions incorporated in the modeling concepts. These abstractions make modeling more intuitive and efficient than manually writing code for a target platform.” [81, p.456]

Domain Specific Programs enhance productivity and maintainability [53], [79], [67], [81], [76], [36]. Case studies have shown that an increase of productivity is gained by using domain specific languages and models (see chapter 3.7.3 *Studies*).

At model level, validation can be ensured by implementing and executing checks on the model. This means that a well-formed model can be created which can be

regarded as syntactically and semantically correct (see chapter 3.11 *Syntax and Semantics*).

Test cases and test data can be extracted directly from the model. As the system is specified by the model, all parameters for a test are already defined and could be gathered from there (see chapter 8 *Test Data Generation*).

Furthermore, by using a graphical representation of a DSL, more information is transferred to the observer in a shorter time, compared to a description in ordinary language [68], [61].

“In addition, diagrams can convey information more concisely and precisely than ordinary language. Information represented visually is also more likely to be remembered due to the picture superiority effect.” [54, p.756]

“Furthermore, of all methods for analyzing and communicating statistical information, well-designed data graphics are usually the simplest and at the same time the most powerful.” [73, p.10]

### 3.3 Disadvantages

There are disadvantages regarding the used tools as the “tool environment is crucial” [53, p.437]. Furthermore, “the availability of tools was perceived as the most influential factor” [53, p.437].

“When discounting the problems with the use of immature tools, the loss in productivity was 10%.” [53, p.438]

By comparing hand-coded software with automatically generated code, the potential loss of efficiency greatly depends on the skills of the programmer and the quality of the code generator. There might be a drawback, because the generated code might not be as specific for the target and might not use the appropriate functions provided by the platform (see chapter 3.2 *Advantages*) [76].

Another disadvantage could be that a new defined DSL must be implemented into the process of a company. This means that the users must be educated and trained to use the new approach. This is especially the case if the used software might have flaws and workarounds need to be introduced.

In most cases, the creation and implementation of a DSL is not the primary goal of a company. They need to use a tool. Regarding the phases in which a DSL is defined, implemented and used (see chapter 3.9 *DSL Phases*), some time must be invested

before the DSL can be used in the last phase. This means, that the company must invest time and money before the DSL can be regarded as productive.

Wu et al. state that tool support for DSL is limited when compared to tool support for general purpose languages. As “support for unit testing a DSL program is absent” [83, p.125], a framework is developed to overcome this deficiency.

Kleppe sees more advantages in a “high level, general purpose, software language” [38, p.221] instead of building many different DSLs.

“Second, domain specific languages are positioned as languages that can be developed by the domain experts themselves. If the supporting tools allow each expert to define his own domain specific language, the world would see a new version of the story of the Tower of Babel [...]. None of the experts, even in the same domain, would be able to understand the language built by one of his colleagues.” [38, p.222]

### 3.4 DSL Examples

Literature claims that there are several DSLs which are commonly used but not regarded as a DSL. So called “little languages” which are common in any Unix environment are regarded as DSLs. Known examples are LEX, YACC and AWK.

“Over the past few years many different languages have been developed, such as Yacc for describing grammars and parsers (Johnson, 1975), PIC for drawing pictures (Kernighan, 1982), PRL5 for maintaining database integrity within digital switching systems (Ladd and Ramming, 1994), Morpheus for specifying network protocols (Abbot and Peterson, 1993), and GAL for specifying video device drivers (Thibault et al., 1997).” [75, p.2]

Furthermore, the languages SQL, BNF, HTML, and Make are mentioned [76]. Other examples include CSS, PHP, HQL, JPQL, and DOT. An example for an executable test specification language is TTCN-3 [18].

A common practice is to embed a DSL into another language in order to enrich the target [22], [84]. The DSL will inherit the infrastructure from the other language and can therefore re-use the tools and concepts from that language. For instance, the language Real-Time Tester Language (RTTL) from Verified’s RT-Tester is embedded into C-Code [77]. A preprocessor will translate the additional commands to C-Code and add the necessary adjustments to the conventional C-Code. The result is a common C-File which can be compiled using a standard C-Compiler.

Haxthausen and Peleska propose a DSL for a railway and tram control system which can be checked with regard to syntax and semantics by additional tools [32].

A DSL called “DirectFlow” for information flow systems is presented by Lin and Black [45]. Another example is Hume [24] which is a “domain specific language for resource-limited systems such as the real-time embedded systems domain” [25, p.140]. A driver specification language is described by Kutter et al. [43].

### 3.5 Visual and Textual Languages

The human being is a visual oriented being and visual languages are the first known written languages (for instance cave-paintings which are more than 32,000 years old). Most of the information is collected by the visual system of the human brain. As this visual system can operate parallel, visual information can be processed faster than other inputs.

“We like receiving information in visual form and can process it very efficiently: around a quarter of our brains are devoted to vision, more than all our other senses combined.” [54, p.756]

Moody [54] points out that there are differences on how textual and visual languages are processed by the human mind. Within textual languages, information is encoded by using sequences of characters. This representation is linear (one-dimensional). Visual languages are spatial (two-dimensional).

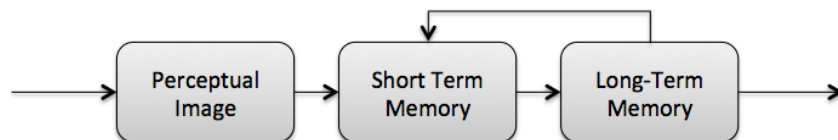


Figure 3.1: Visual Information Processing

Kosslyn [40] describes the canonical theory which divides visual information processing into three general phases (see figure<sup>1</sup> 3.1 *Visual Information Processing*). In the first phase, lines and marks and their orientation are recognized. In the second phase, the output from the first phase is further processed. Here, the lines and marks are organized into “perceptual units”. Kosslyn describes rules which “dictate how lines and marks will be grouped” [40, p.501]. One of these rules argues that “three lines that meet to form an enclosed figure are not seen as three lines, but as a triangle” [40, p.501]. These perceptual units are stored in the short-term memory (“working

<sup>1</sup>Original image was published in [40].

memory”) which is limited by capacity. The information from the first phase is then processed together with information which was stored previously in the long-term memory. In this step, the new information is compared with the “knowledge” a person gained over time. At the end, the relevant information from the working memory might be stored in the long-term memory.

“Human visual processing can be divided into three general phases, each of which has properties that place constraints on display design.”  
[40, p.499]

Furthermore, Moody describes that the human brain uses two different systems (dual channel theory) for processing pictorial and verbal information. Textual information is processed sequentially by the auditory system whereas visual representations are processed parallel by the visual system [54].

This means, that graphical languages could be processed faster and information can be retrieved in shorter time.

Furthermore, Moody describes the work by Newell and Simon [65], [64] and shows that human beings can be compared to informational processing systems. As software could be optimized for the target hardware, a visual language could be designed to be processed by the human brain in an optimal way. “Principles of human graphical information processing provide the basis for making informed choices among the infinite possibilities in the graphic design space.” [54, p.760]

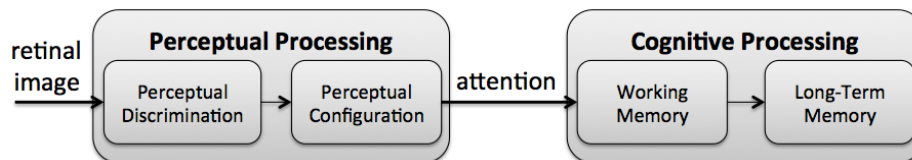


Figure 3.2: The Solution Space

The processing of graphical information is divided into two phases (see figure<sup>2</sup> 3.2 *The Solution Space*). The first phase is called perceptual processing phase which consists of two parts. The first part is the perceptual discrimination where parts of the retinal image are parsed and elements are separated from the background. The second part is the perceptual configuration where structures, patterns and relationships between elements are detected.

The second phase is the cognitive phase. According to Moody, this phase consists of two parts: the “working memory” and the “long-term memory”. The working memory is regarded as a temporary storage area. It is described as an area with

---

<sup>2</sup>Original image was published in [54].

limited capacity and duration. The long-term memory has unlimited capacity but is described as slow in comparison to the working memory. All knowledge is stored in this area. The information of the visual image is aligned with this prior knowledge. Of course, as this knowledge might differ on an individual basis, the speed and accuracy of processing images could vary [54].

The perceptual phase and the cognitive phase differ in the ways they are processed by the human brain. Perceptual processes are described as automatic and very fast as several processes are executed parallel. In contrast, the cognitive phase is described as “conscious control of attention” [54, p.761]. This process is described as slow and sequential but effortful.

“A major explanation for the cognitive advantages of diagrams is computational offloading: they shift some of the processing burden from the cognitive system to the perceptual system, which is faster and frees up scarce cognitive resources for other tasks. The extent to which diagrams exploit perceptual processing largely explains differences in their effectiveness.” [54, p.761]

Therefore, it is important that each visual language is based on the knowledge of processing visual information. If this information is incorporated into the design of the language, it can be assured that the language is processed by the human brain as effective as possible.

“Thus effective graph design requires an understanding of human visual information processing.” [40, p.499]

Larkin and Simon discuss the computational efficiency between informationally equivalent sentential and diagrammatic representations [44].

“Diagrams automatically support a large number of perceptual inferences, which are extremely easy for humans.” [44, p.98]

However, only if the knowledge of “how to construct a good diagram” [44, p.99] exist, these advantages are present.

### 3.5.1 Visual Language Design

Moody pointed out that textual and visual languages are processed differently by the human brain (see chapter 3.5 *Visual and Textual Languages*). He states that this has an impact on the design of a visual language.

“These differences mean that fundamentally different principles are required for evaluating and designing visual languages.” [54, p.756]

However, even though visual notations are used for years, “such principles are far less developed than those available for textual languages” [54, p.756].

“Visual notations form an integral part of the language of software engineering (SE). Yet historically, SE researchers and notation designers have ignored or undervalued issues of visual representation.” [54, p.756]

Furthermore, while “SE has developed mature methods for evaluating semantics of notations, comparable methods for evaluating visual notations are notably absent” [54, p.756]. As consequence some dialects, a different visual language of a notation, exist. For instance, the Data Flow Diagrams [16] exist in two equivalent types: “the DeMarco dialect, consisting of circular ‘bubbles’ and curved arrows and the Gane and Sarson dialect, consisting of rounded rectangles (‘routangles’) and right-angled lines” [54, p.758] (see figure<sup>3</sup> 3.3 *Visual Dialects*).

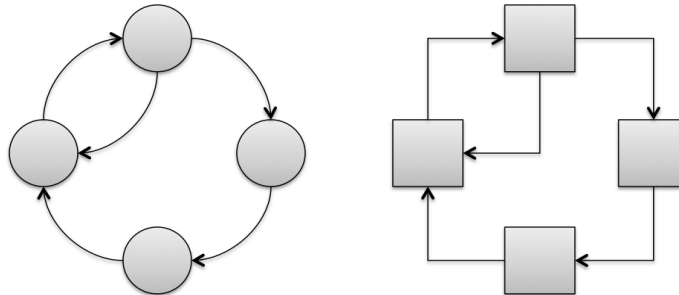


Figure 3.3: Visual Dialects

“Despite the fact that these notations have been used in practice for over 30 years, there is still no consensus on which is best: neither market forces nor Darwinian-style selection mechanisms seem to be able to identify the ‘fittest’ alternative. Without sound principles for evaluating and comparing visual notations, there is no reliable way to resolve such debates. As a result, the choice of an appropriate dialect normally comes down to personal taste.” [54, p.758]

“Very little is documented about why particular graphical conventions are used. Texts generally state what a particular symbol means without

---

<sup>3</sup>Original image was published in [54].



giving any rationale for the choice of symbols or saying why the symbol chosen is to be preferred to those already available. The reasons for choosing graphical conventions are generally shrouded in mystery.” [54, p.756]

Tufte declares ground rules for graphical displays which he derives — among others — from “two great inventors of modern graphical design [...] J.H. Lambert (1728–1777), a Swiss-German scientist and mathematician, and William Playfair (1759–1823), a Scottish political economist” [73, p.32]. His work focuses on diagrams but the core statement is also applicable for graphical languages. He states that the principles for a theory for graphics “above all else show the data” [73, p.92].

“Data graphics should draw the viewer’s attention to the sense and substance of the data, not to something else. The data graphical form should present the quantitative contents.” [73, p.91]

Starting with the fact that “the human information-processing system has limited capacity” [47, p.44], it should therefore be avoided that the graphs lead to a “cognitive overload” [47, p.45]. The available processing capacity must not be exceeded.

Tufte states the following ground rules:

“Show the data

Induce the viewer to think about the substance rather than about methodology, graphical design, the technology of graphics production, or something else.

Avoid distorting what the data has to say

Present many numbers in a small space

Make large datasets coherent

Encourage the eye to compare different pieces of data

Reveal the data at several levels of detail, from a broad overview to the fine structure

Serve a reasonable clear purpose: description, exploration, tabulation, or decoration

Be closely integrated into with the statistical and verbal description of the data set” [73, p.14]

These ground rules form Tufte’s Principles of Graphical Excellence:

“Graphical excellence is the well-designed presentation of interesting data — a matter of substance, of statistics, and of design

Graphical excellence consists of complex ideas communicated with clarity, precision, and efficiency

Graphical excellence is what gives to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space

Graphical excellence is nearly always multivariate

And graphical excellence requires telling the truth about data” [73, p.51]

### 3.5.2 Visual Notation

In this chapter, the key concepts of a visual notation are presented. A good visual notation will take advantage of the human information processing abilities. This means that the human capabilities like perception, memorization and conceptualization must be taken into account.

“An optimal graphical display for a human might not be optimal for robots or creatures from another planet. Indeed, graphics are effective because they exploit properties of human information-processing abilities.” [40, p.501]

“To be most effective in doing this, they need to be optimised for processing by the human mind.” [54, p.757]

However, the right choice of using visuals which fit together is not a trivial task, the “meaning we wish to communicate often gets lost in graphics that are cluttered” [15, p.157]. In the end, the diagram “should tell a story, as well as exhibit good design principles” [15, p.159].

Each visual notation is formed of a visual vocabulary and a visual grammar. The visual vocabulary consists of “graphical symbols [which] are used to symbolize (or perceptually represent) semantic constructs” [54, p.757]. Both together form the concrete syntax of this notation. Also, the semantics of the vocabulary needs to be defined (see chapter 3.11 *Syntax and Semantics*). These three parts form the “Anatomy of a Visual Notation” [54].

“A valid expression in a visual notation is called a visual sentence or diagram. Diagrams consist of symbol instances (tokens), arranged according to the rules of the visual grammar.” [54, p.757]

In order to decode the stored information from such a diagram, a solid visual notation is needed. This notation should be based on how the human brain processes information as “the visual form of notations significantly affects understanding” [54, p.758].

“[...] the form of representations has an equal, if not greater, influence on cognitive effectiveness as their content. Human information processing is highly sensitive to the exact form information is presented to the senses: apparently minor changes in visual appearance can have dramatic impacts on understanding and problem solving performance.” [54, p.758]

### 3.5.2.1 Principles

Nine principles are defined in [55] which must be taken into account in order to create a cognitive effective visual notation. These principles are summarized here and some of them are explained in more detail in the following sections.

“Improving a visual notation with respect to any of the principles will increase its cognitive effectiveness (subject to tradeoffs among them).” [55, p.145]

**Semiotic Clarity** means that each single symbol should be mapped to one semantic construct and each single semantic construct should be mapped to exactly one symbolic representation (1:1 mapping).

**Perceptual Discriminability** means that all symbol representations must be distinguishable from each other. It should be avoided that symbols can be mistaken for one another.

**Semantic Transparency** means that the used symbols should reflect their meaning.

**Complexity Management** should be included in order to handle complexity.

**Cognitive Integration** means that it should be possible to integrate information from different kinds of diagrams.

**Visual Expressiveness** means that the designer of the DSL should not limit the usage of the visual variables when specifying the graphical symbols.

**Dual Coding** means that text should be used to annotate graphics.

**Graphic Economy** means that there should be a limited set of symbols in order to keep the overall number of symbols manageable.

**Cognitive Fit** means that a visual dialect of the visual notation should be used depending on the tasks/audiences.

Of course, some of these principles seem to contradict each other. For instance, Cognitive Integration where information might be collected and gathered from several different diagrams might object the principle of Graphical Economy which tries to keep the number of symbols manageable. On the other hand, the principle Visual Expressiveness could emphasize Perceptual Discriminability because the usage of different properties of a symbol helps to distinguish between them.

### 3.5.2.2 Semiotic Clarity

Semiotic Clarity means that there should be no mismatches between a semantic construct and its graphical symbol. For instance, it should be avoided that there are several semantic constructs which are presented by the same symbol. Also, a construct shall not be presented by different symbols.

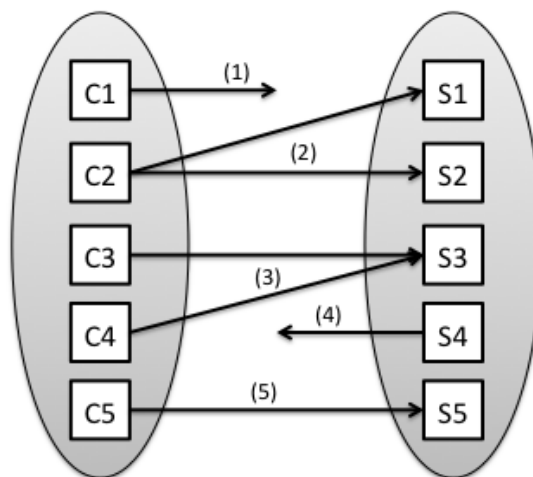


Figure 3.4: Semiotic Clarity

Moody [54] describes four possible anomalies (see figure<sup>4</sup> 3.4 *Semiotic Clarity*) which should be avoided:

**Symbol redundancy** This occurs when a construct can be mapped to more than one symbol (2).

**Symbol overload** This occurs when more than one construct is mapped to one symbol (3).

**Symbol excess** This occurs when a symbol exists which cannot be mapped to a construct (4).

**Symbol deficit** This occurs when a construct exists which cannot be mapped to a symbol (1).

The only valid mapping between a construct and a symbol is depicted in (5).

### 3.5.2.3 Perceptual Discriminability

In order to decode the information from a diagram, the symbols and their relationships must be interpreted. Each symbol must be distinguishable from another one so that this interpretation can be done correctly.

“Perceptual discriminability is the ease and accuracy with which graphical symbols can be differentiated from each other. This relates to the first phase of human visual information processing: perceptual discrimination. Accurate discrimination between symbols is a prerequisite for accurate interpretation of diagrams.” [54, p.762]

A visual variable describes a property of a symbol. The basis is a set of eight variables [54]:

- Horizontal Position
- Vertical Position
- Shape
- Size
- Color
- Brightness

---

<sup>4</sup>Original image was published in [54].

- Orientation
- Texture

Another definition of visual variables is made by Delpont et al. [15]. Eight variables are defined which are similar to Moody's definition. However, one variable is the position which means the coordinates on the diagram (this is split into two variables in Moody's definition: Horizontal and Vertical Position). This includes the depth, if a third axis existed. The additional variable is motion.

“A visual variable can be seen as being selective when isolating a specific instance of the variable is effortless. Locating the variable should be easy, most variables are selective, however shape is not. It is easy to locate a green object from a bunch of other coloured objects, but locating a triangle in between a sea of rectangles is a lot more difficult and thus shape is not selective.” [15, p.158]

Each variable does have a different impact on the reception of the symbol. The brightness of a hue has a great impact, because “it can represent emotions, space, movement, and the illusion of light” [15, p.159].

“People find it much easier to distinguish variations in brightness than say in hue or saturation differences.” [15, p.159]

“Closely related to value is the visual variable of colour. This is a very powerful variable and can take on as many variations as the eye can distinguish. Changing the colour of a variable is seen as changing the hue without changing the value.” [15, p.159]

#### 3.5.2.4 Visual Distance

As constructs are represented as symbols, they must differ in order to distinguish between these constructs. This is done by creating a visual distance between their symbols, meaning that different kinds of colors or shapes are used, for example.

“Discriminability is primarily determined by the visual distance between symbols.” [54, p.762]

Each symbol has several properties like shape or size. These properties are called visual variables. The visual distance is determined by the count of visual variables which differ between the symbols. Of course, if the variables between two representations are similar and they only differ in a very small part, it cannot be assured

that the symbols are recognized as two different symbols. For instance, the same shape and size is chosen and color is the only variable used for distinction. If the chosen hues do not differ much, chances are that the symbols are not used like it was intended. This means that some kind of spacing is needed for one variable. The more variables are used and the more space is used within these variables, a clear distinction between symbols is possible.

“In general, the greater the visual distance between symbols, the faster and more accurately they will be recognised. If differences are too subtle, errors in interpretation may result.” [54, p.762]

Of course, the size of the gap within the variables depend greatly on the knowledge of the user. An experienced user of the problem domain and its vocabulary will notice differences in similar symbols faster than a newcomer.

“Requirements for discriminability are higher for novices than for experts as we are able to make much finer distinctions with experience.” [54, p.762]

### 3.5.2.5 Perceptual Popout

Symbols can be detected and processed automatically if they differ in at least one unique visual variable. For instance, a colored triangle can be spotted quickly in a diagram with other triangles which have a different color.

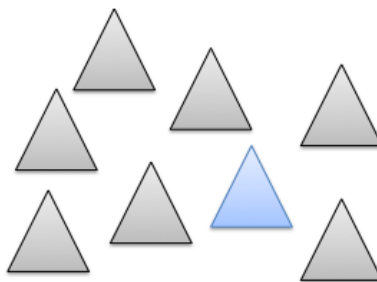


Figure 3.5: Perceptual Popout

“According to feature integration theory, visual elements with unique values for at least one visual variable can be detected pre-attentively and in parallel across the visual field. Such elements appear to ‘pop out’ from a display without conscious effort.” [54, p.763]

If a combination of visual variables is needed to distinguish between symbols, none of the symbols will “pop out”. For instance, if a combination of shape, fill pattern and color is used, two values for these three variables can describe eight different symbols. Of course, each of these symbols represents one unique construct. Therefore, by using such an approach more attention is required because more effort is needed in order to link symbols to constructs.

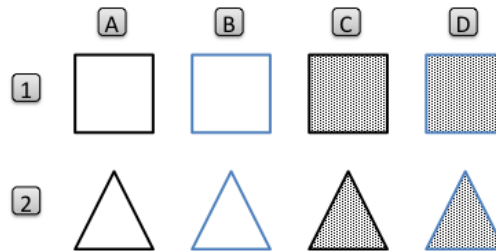


Figure 3.6: Perceptual Popout: Combination of Visual Variables

“[...] visual elements that are differentiated by unique combinations of values (conjunctions) require serial search, which is much slower and error-prone.” [54, p.763]

On the other hand, less variables are necessary if combinations are used. For instance, if the color set is reduced in order to print the diagrams later on, the distinction needs to be made by other variables.

Also, grouping of symbols is possible. If different relationships between objects need to be pictured, different colors and patterns can be used. A color could be used to describe a type of relationship whereas the pattern (solid or dotted) can describe the importance. This is used in maps where lines stand for borders of an area and their color or thickness is used to determine between borders of a country, state, district or town.

“Also colour is associative, meaning that marks belonging of the same colour can be grouped into a category.” [15, p.159]

### 3.5.2.6 Icons

In most cases, icons are little images which describe a certain aspect or action as they “resemble the concepts they represent” [54, p.764]. Icons are used worldwide and some are even used as standard signs, for example the emergency exit sign.



“Iconic representations speed up recognition and recall and improve intelligibility of diagrams to naive users. They also make diagrams more accessible to novices: a representation composed of pictures appears less daunting than one composed of abstract symbols.” [54, p.764]

A good designed icon is recognized and speeds up the understanding of a diagram. This feature can be used to assign icons to other symbols in order to increase the distinction between objects. Objects get more self-explanatory if standardized shapes are used to describe a type of an object and an icon for the specific function.

“Finally, they make diagrams more visually appealing: people prefer real objects to abstract shapes.” [54, p.764]

### 3.5.2.7 Redundant Coding

In order to support the distinction between objects on the diagram, redundancy should be used. This means that the visual distance between symbols could be increased by using more than one visual variable (see chapter 3.5.2.2 *Semiotic Clarity*). For instance, not only the shape differs between two objects but also the color or size.

“Redundancy is an important technique in communication theory to reduce errors and counteract noise. The visual distance between symbols can be increased by redundant coding: using multiple visual variables to distinguish between them.” [54, p.763]

For instance, regarding the two symbols “A1” and “B2” of figure 3.6 *Perceptual Popout: Combination of Visual Variables*, two variables are used: shape and color.

### 3.5.3 Diagrams

Diagrams could be interpreted differently if the diagram is not fully specified and well-formed (see chapter 3.11 *Syntax and Semantics*). This could be the case within one workgroup, although chances are high that contradictions could be invalidated by discussing the diagram within a meeting. Of course, this is easier, if the workgroup is located nearby. If the workgroup is scattered over different locations these discussions might not take place. Chances are, that the diagrams are then interpreted differently without knowing the understanding of the other group members.

“More often than not, it is more luck than planning when the design diagrams are interpreted consistently across geographically dispersed organizations.” [79, p.36]

Keeping in mind that models are mostly used as an add-on for describing a system in a traditional development process by embedding the model as an image into the documentation, they have no direct link to the implementation. In most cases, there is no transformation of information from the model into the code.

“In a traditional development process, models are, however, kept totally separate from the code as there is no automated transformation available from those models to code. Instead developers read the models and interpret them while coding the application and producing executable software.” [36, p.4]

Developers tend to differentiate between diagrams and coding [36]. If a diagram is understood as intended, this would help the developer to apprehend the function of the described system. However, the model is only used for reasons of documentation and might not be considered as a basis for the implementation.

“Developers generally differentiate between modeling and coding. Models are used for designing systems, understanding them better, specifying required functionality, and creating documentation. Code is then written to implement the designs.” [36, p.4]

Furthermore, the following steps like debugging, testing and maintaining would also take place on the lowest level, the coding level. Both, the code and the diagrams, are not regarded as the same thing and no relation between these levels are considered.

“Debugging, testing, and maintenance are done on the code level too. Quite often these two different ‘media’ are unnecessarily seen as being rather disconnected, although there are also various ways to align code and models.” [36, p.4]

### 3.5.4 Evaluation of a Visual Notation

As there are no design rules but design suggestions, it is not an easy task to evaluate a visual notation. The main task of a visual notation “is to facilitate human communication and problem solving” [54, p.757]. Therefore, the main purpose is the transport of information. The recipient needs to apprehend the information correctly and potentially in a short time.

“The surprising thing about modelling is that the notation is simple but using it correctly and effectively is considered quite difficult to achieve.” [33]

The goal is to create a check, if the information is received as intended. One possible solution is to establish a test which questions the reception of information. The more answers are regarded as being “correct”, the better is the design of the visual notation. Of course, if most of the answers must be considered as “wrong”, a change of the visual notation might be the appropriate next step.

“A commonly accepted variable measuring the level of comprehension, for example, is ”correctness“, i.e. the number of correct answers given to a (test) questions [sic!].” [68, p.80]

To enhance this approach, the test could contain two different kinds of notation which depict the same topic. The test person could then choose a notation which he could “better” understand. To create such a test at least two semantic equally notations which express the same information are needed. These notations must differ in their representation.

“Apart from semantic equality, the expressions being compared need to be expressed at an equal degree of compression.” [68, p.82]

Regarding visual notations, an appropriate test could consist of a model which describes a certain aspect and a textual — prosaic or mathematical — description.

“To do so, testers could either look at the question or look at the expression (an algorithm, in this case). This is an interesting solution for the aforementioned problem of separating comprehension time and response time. Scalan’s [61] outcome was that structured flow charts are beneficial.” [68, p.87]

But not only the “correct” answer is of interest, also the time the test person needed to answer the question. This is described as “comprehension speed” or “cognitive effectiveness”.

“Another variable is ‘comprehension speed’, e.g. the number of seconds that the subjects look at the object (or maybe even ‘easy to remember’, i.e. the number of times that the subjects take a look at the objects [...]).” [68, p.80]

“Cognitive effectiveness is defined as the speed, ease and accuracy with which a representation can be processed by the human mind.” [54, p.757]

If such a test is created and performed for the specific modeling language, the quality of this language can be evaluated empirically. A high level of cognitive effectiveness is established, if the language is well accepted and understood by all participating parties (like language designer, modeler and user). This ensures that the information is transported between and comprehended by all stakeholders.

“The cognitive effectiveness of visual notations is one of the most widely held (and infrequently challenged) assumptions in the IT field. However, cognitive effectiveness is not an intrinsic property of visual representations but something that must be designed into them. Just putting information in graphical form does not guarantee that it will be worth a thousand of any set of words. There can be huge differences between effective and ineffective diagrams and ineffective diagrams can be less effective than text.” [54, p.757]

Card and Mackinlay [10] analyze the design space of different visualizations in order to map “the morphology of the design space of visualizations” [10, p.8]. The work is intended to create “a framework for designing new visualizations and augmenting existing designs” [10, p.1].

### 3.6 Guidelines

The ground rules on how to design the objects of a visual language are depicted in chapter 3.5 *Visual and Textual Languages*. In order to create a good visual notation, the principles are described there — the lower level of the model.

Sustained on this basis, guidelines should be developed in order to create models by the use of the elements of the visual notation. This is the higher level of the model where the elements are not only grouped into a graph but also the techniques gain cognitive effectiveness.

“Requiring Statecharts to be created with a uniform layout helps to reduce the time and effort, and in turn the cost, of Statechart design and maintenance.” [78, p.4]

Basically, two main tasks are involved. First, the view of the structure of the model needs to be established. This means that the techniques must be used in order to help the viewer accessing the structure behind the model. Second, the design of

the model needs to be described. According to the principles of the visual notation, design rules can be stated for graphs.

“Combining words, numbers and pictures often reveals a lot about the data, giving an explanation of the data to the viewer.” [15, p.160]

The goal is to design detailed models without losing the ability to maintain and understand them. Therefore, “graphics should be as intelligent and sophisticated as the accompanying text” [73, p.136].

“The general idea of understandable Statecharts is that they have to fulfill two criteria: Good structure and good layout.” [78, p.11]

The problem that too detailed models are turning into “visual puzzles” [73, p.136] must be avoided at all cases.

“The complexity of multifunctioning elements can sometimes turn data graphics into visual puzzles, crypto-graphical mysteries for the viewer to decode. A sure sign of a puzzle is that the graphic must be interpreted through a verbal rather than a visual process.” [73, p.153]

This does not mean that models should not be complex or detailed — as this is necessary in most cases. It means that techniques must be used in order to avoid bad design and emphasize the needed information for a specific task.

“Confusion and clutter are the products of bad design, not attributes of information. Layering and separation can solve this, but is difficult to do effectively.” [15, p.160]

Apart from the viewing techniques, the design of the graph is fundamental. Contemporaneously, the design is highly dependent on the content of display. From this follows that it is nearly impossible to state general rules for design which are applicable for every model. Instead, ground rules can be given which should help to create well designed models.

“Design is choice. The theory of the visual display of quantitative information consists of principles that generate design options and that guide choices among options. The principles should not be applied rigidly or in a peevish spirit; they are not logically or mathematically certain; and it is better to violate any principle than to place graceless or inelegant marks on paper. [...] What is to be sought in designs for the display of information is the clear portrayal of the complexity. Not

the complication of the simple; rather the task of the designer is to give visual access to the subtle and the difficult — that is, the revelation of the complex.” [73, p.191]

“What makes for such graphical elegance? [...] Good design has two key elements: Graphical elegance is often found in simplicity of design and complexity of data.” [73, p.177]

This means that models can have a high level of details. But there must be techniques to filter out or emphasize details of the model. Also, models should be designed for simplicity, meaning that the graphs should not be cluttered by objects but have a clear design.

### 3.6.1 Literature

Most DSLs are textual languages and their design is therefore mainly similar to textual programming languages. Hence, the design of these languages is well understood.

This is not the case for graphical languages. Here, guidance is rare on how to create a good language by designing graphical objects and representations from scratch. General guidelines for creating a DSL are mentioned by van Deursen et al. [75], [76] and Mernik et al. [48].

In case of a graphical DSL, guidelines for a good design — with respect to gather and process information as fast and accurate as possible — is mentioned by Tufte [72], [73].

### 3.6.2 Views

The size of a model increases with the complexity of the model. This means that it might be complicated to spot the necessary information immediately. Some aspects can be hidden and not visible at first sight. If models are used to create an understanding between different stakeholders, a shared view on a problem or aspect is necessary (see chapter 5.1 *Sub-Projects*). Also, the model might be used by different kinds of users. For instance, if the model is used for testing reasons and the model is the input for a simulation and a test data generator, the needed information for that user might differ. Nevertheless, both types of information must be included in the model. It is therefore necessary to create different views or to have the possibility to do so.

The model should be accessible from a high-level view, meaning that the overall

structure is displayed and an overview is given. Also, the details must be made visible. There are several possibilities to conduct this undertaking. An overview graph could be enriched by details or several sub-graphs could be created which are linked to the appropriate aspects of the overview graph. However, the best option might depend on the project and its environmental requirements.

“Graphics can be designed to have at least three viewing depths: (1) what is seen from a distance, an overall structure usually aggregated from an underlying microstructure; (2) what is seen up close and in detail, the fine structure of the data; and (3) what is seen implicitly, underlying the graphic — that what is behind the graphic.” [73, p.155]

According to Delport et al. [15], these three key views should be used in order to display and emphasize information if large datasets are used.

“Graphics should display an overview of the information first, giving the option to zoom and filter, thus allowing the user to get the details-on-demand.” [15, p.161]

“Information Visualization Mantra: Overview first, zoom and filter, then details on demand.” [57, p.18]

**Overview** The main and most important data is shown. A summary on the data is given. This is the starting point for all further displays of data.

The overview shall display the summary but the user should be encouraged to dive deeper into the data [15], [57].

The overview should “gain an overview of the entire collection” [63, p.339].

**Zooming and Filtering** Parts of the graph are explorable and provide more information. Two concepts can be used in order to display additional data. By zooming, parts are displayed in different resolutions thus providing different detail levels. For instance, zooming in can be achieved by displaying an object in greater resolution and showing the properties according to its size. The larger an object is drawn, the more information is shown and vice versa. Also, subgraphs can be used. Zooming into an object would result in a new graph where the details are shown.

Filtering means that some parts of the graph can be blended in or out in order to reduce or raise the level of details.

Users have an interest in only some parts of the data and want to zoom into it [63], [15].

**Details on Demand** means that the user can insert information by blending in overlay layers. Also, by blinding out certain details, data can be hidden in order to reduce the level of displayed information.

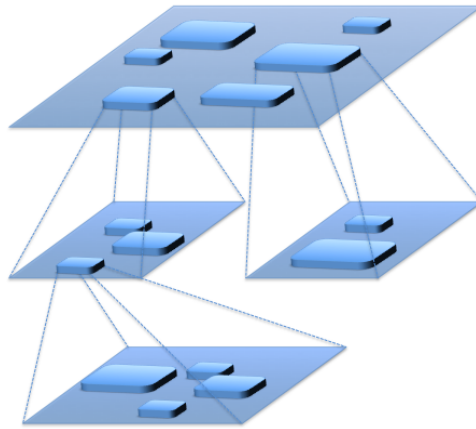


Figure 3.7: Hierarchies

These views can be implemented by creating layers of hierarchy in the model. Related parts of the model are compiled into compounds (see figure 3.7 *Hierarchies*). Regarding the viewing depths, the first layer contains the overall view. All needed parts are displayed in a way that the whole ensemble could be surveyed. Of course, not all information can be revealed here, this is why some objects can be opened and the content of these objects is shown: a new layer of hierarchy is established. This procedure can be further refined and new layers can be introduced if needed. Of course, the number of layers depends on the complexity of the model.

“Such an analysis of the viewing architecture of a graphic will help in creating and evaluating designs that organize complex information hierarchically.” [73, p.159]

In addition to the “overview to detail” views, another kind of view is necessary. Various roles might be used to look at the model: the test designer who is responsible for the test data might need different information than the designer who creates and maintains the simulations. Although, the needed information might differ, they also might need to have the same view on a certain aspect. This means, that they both look at the same graph but other details are shown depending on the role.

“Multiple layers of information are created by multiple viewing depths and multiple viewing angles.” [73, p.154]



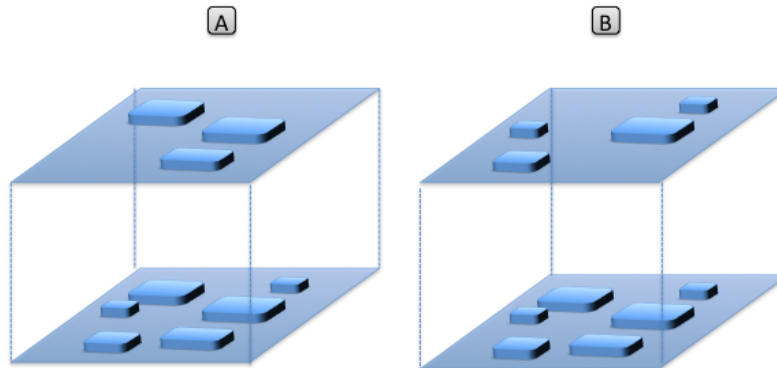


Figure 3.8: Views on the Model

In figure 3.8 *Views on the Model*, two different views are depicted. The underlying model is the same for both views. However, not all details are shown for view (A) and (B). These views can be regarded as projections of the model. Each projection displays a specified set of the underlying model, necessary for the given task.

“The different domain-specific views can then be viewed as projections of the underlying merged model.” [62, p.300]

By creating views, and therefore interactivity with the model, viewers become users of the graph as they “can now control the sequence, the pace, as well as what they want to see or wish to ignore” [15, p.161].

“When large amounts of data are displayed in a graphic it can however cause confusion. Introducing interactivity to the graphic has been shown to improve its effectiveness.” [15, p.161] [37]

By using these three different techniques (overview, zooming and filtering, and details on demand) of displaying data, the user can choose the richness of details and can fade in only the important data for a specific task.

### 3.6.3 General Guidelines

Despite the fact that there could be situations where specific guidelines cannot be applied rigidly, general guidelines can be developed. These guidelines help to ensure a clean design and emphasize the data of the model.

Information can be transported by different means like text, graphics or tables. Each of these means has its advantages and disadvantages. Therefore, the means should

be chosen according to the selected information which should be transported to the user.

“More information is better than less information, especially when the marginal costs of handling and interpreting additional information are low, as they are for most graphics. The simple things belong in tables or in the text: graphics can give a sense of large and complex data sets that cannot be managed in any other way.” [73, p.168]

For instance, a decision table is a well established form of displaying combinations and links between elements. This could also be described in textual form but if complex datasets are used, the text will get large und unclear. Also, a state or conclusion could be forgotten in the text and this might not be noticed. A decision table can collect all states in a clear and compact form. A missing state or conclusion can be detected easily.

On the other hand, this does not mean that only one formalization should be used. The combination of different representation forms supports the comprehension.

“Words and pictures belong together. Viewers need the help that words can provide. [...] Explanations that give access to the richness of the data make graphics more attractive to the viewer.” [73, p.179]

By combining these different means, the quality and the expressiveness of the model is increased. More information can be displayed in less space regarding the readability of the model.

“It is nearly always helpful to write little messages on the plotting field to explain the data, to label outliers and interesting data points, to write equations and sometimes tables on the graphic itself, and to integrate the caption and legend into the design so that the eye is not required to dart back and forth between textual material and the graphic.” [73, p.179]

In conclusion, several general guides are identified which support presenting information:

**“A story to tell about the data”** means that the graph should transport more than just trivial information [73, p.177].

**It should “have a properly chosen format and design”** in order to emphasize the presented information [73, p.177].

**“Words, numbers and drawing[s should be used] together”** because this increases the expressiveness of the model [73, p.177].

**“Display an accessible complexity of detail”** means to use the right balance between presenting too less or too much information [73, p.177].

**“Reflect a balance, a proportion, a sense of relevant scale”** within the displayed data. The objects and visual linkage between these objects should be in scale [73, p.177].

**“Avoid content-free decoration, including chartjunk”** because the decoration should emphasize the displayed data and not distract the viewer from it [73, p.177].

Specific policies can be deduced from these general guidelines. Tufte [73, p.183] distinguishes between two categories: friendly and unfriendly graphics. Each category consists of “dos and don’ts” which are meant to help to support or avoid certain realizations:

### 3.6.3.1 Friendly

The following guides should be regarded in order to create a “friendly graphic”:

“Words are spelled out, mysterious and elaborate encoding avoided

Words run from the left to the right, the usual direction for reading occidental languages

Little messages help explain the data

Elaborate encoded shadings, cross-hatching, and colors are avoided; instead labels are placed on the graphic itself, no legend is required

Graphic attracts viewer, provokes curiosity

Colors, if used, are chosen, so that the color-deficient and color-blind (5 to 10 percent of viewers) can make sense of the graphic (blue can be distinguished from other colors by most color-deficient people)

Type is clear, precise, modest; lettering may done by hand

Type is upper-and-lower case, with serifs” [73, p.183]

### 3.6.3.2 Unfriendly

The following items should be disregarded in order to avoid “unfriendly graphics”:

“Abbreviations abound, requiring the viewer to sort through text to decode abbreviations

Words run vertically, particularly along the Y-axis; words run in several directions

Graphic is cryptic, requires repeated references to scattered text

Obscure codings require going back and forth between legend and graphic

Graphic is repellent, filled with chartjunk

Design insensitive to color-deficient viewers; red and green used for essential contrasts

Type is clotted, overbearing

Type is all capitals, sans serif” [73, p.183]

### 3.6.4 Pattern

In order to help distinguishing between objects and raise the visual distance between them (see chapter 3.5.2.4 *Visual Distance*), color and shape are the most used variables. In some graphs, color is replaced by patterns in order to reduce the count of used colors. This is a common approach if a graph needs to be printed to paper, as a black-and-white print is usually cheaper than a colored page. However, if graphics are not meant to be printed, there is no reason not to use colors.<sup>5</sup>

The moiré effects can be regarded as “chartjunk” [73] and should therefore not be used.

“[The] moiré vibration is an undisciplined ambiguity, with an illusive, eye staining quality that contaminates the entire graphic. It has no place in graphical design.” [73, p.112]

Instead of using different kinds of patterns, Tufte suggests to replace these by different hues of gray. In this case, the graph does not become cluttered and easier to read.

---

<sup>5</sup>Of course, in order to increase the visual distance, colors should be used which can be distinguished by color-blind persons.

“Cross-hatching should be replaced with tint screens of gray. Specific areas on a graphic should be labeled with words rather than encoded with hatching.” [73, p.111]

### 3.6.5 Grid

The grid should be used if necessary but not as a style object. There are possible reasons for using grid lines: in order to emphasize data and for layout purposes.

A grid could help when the graph is designed and objects need to be arranged. For instance, if a state-chart like graph is created, the states should not be placed anywhere on the diagram. Instead, the objects should be placed regarding their relationship.

“Grids are mostly for the initial plotting of the data at home or office rather than for putting into print.” [73, p.112]

It should be prevented that a grid line competes with the data of the graph. Therefore, the grid should be “suppressed so that its presence is only implicit” [73, p.112].

“Dark grid lines are chartjunk.” [73, p.112]

Another reason to use a grid is for displaying tables. In this case, a light grey grid line could help reading the content but does not compete with the data.

The human eye can distinguish between a large amount of points within a rather small area. This can be used to emphasize data by placing a light grid which does not compete with the content but supports showing it.

“Our eyes can make a remarkable number of distinctions within a small area. With the use of very light grid lines, it is easy to locate 625 points in one square inch or, equivalently, 100 points in one square centimeter.” [73, p.161]

### 3.6.6 Layout

#### 3.6.6.1 Graph

Following the rule of the golden selection, a graph shall be orientated horizontally as our “eyes are practiced to the horizon and it is therefore easier to read graphs that are greater in length” [15, p.160].

“If graphics should tend toward the horizontal rather than the vertical, then how much so? A venerable (fifth-century B.C.) but dubious rule of aesthetic proportion is the Golden Section, a ‘divine division’ of a line.” [73, p.189]

In the western hemisphere, text is written from the left to the right. Therefore, if additional text needs to be displayed in order to describe the content of the graph, the text should be printed horizontally accordingly. This makes the graph easier to read and understand.

As the causal variable (see chapter 3.5.2.3 *Perceptual Discriminability*) is represented by the horizontal axis, this increases the detail of this variable [73].

### 3.6.6.2 Object

Tufte states that “visual preferences for rectangular proportions have been studied by psychologists since 1860” [73, p.190] but there is no clear outcome which could be taken as a basis for the design of graphical objects. The results vary but a “mild preference” for objects shaped according to the golden rectangle can be experienced [73]. The range (which also includes the square) is depicted in figure 3.9 *Preference Shape Ratio*.

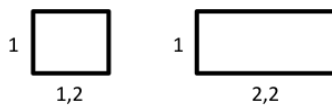


Figure 3.9: Preference Shape Ratio

“And, as is nearly always the case in experiments in graphical perception, viewer responses were found to be highly context-dependent.” [73, p.190]

According to Tufte, the conclusion which can be drawn from the experiments is two-fold. First, “if nature of the data suggests the shape of the graphic, follow that suggestion” [73, p.190], meaning that if the rule of the golden rectangle does not fit for a certain object then it should not be applied only because of this rule. The second conclusion is that objects shall be orientated horizontally and “about 50 percent wider than tall” [73, p.190].

## 3.7 Influence on Productivity

It is claimed by Kelly et al. [36], Tolvanen et al. [71], Wienands et al. [81], Weigert et al. [79], Mohagheghi et al. [53] and Staron et al. [67] that Domain-Specific Modeling increases productivity. Due to a higher level of abstraction, less time and resources are needed to create a product [36]. This is linked to the fact that the domain knowledge is built right into the domain specific model. Therefore, the creator of the models becomes an expert developer by implicitly using this knowledge [71]. The learning curve is flatter and goals could be reached in a shorter time.

“We are not talking here about squeezing another 20% out of our existing developers, programming languages, or development environments. Our industry is long overdue for a major increase in productivity: the last such advance was over 30 years ago, when the move from assemblers to compilers raised productivity by around 500%. Our experiences, and those of our customers, colleagues, and competitors, have shown that a similar or even larger increase is now possible, through what we call Domain-Specific Modeling. Indeed, the early adopters of DSM have been enjoying productivity increases of 500–1000% in production for over 10 years now.” [36, p.13]

Also, this covers time after the product is brought to market: the effort in the maintenance phase could be kept lower [36]. This is because the loop from adjusting a model, create target code and re-run the tests for the product can be highly automated [53]. Also, changes in the model are less complex as the level of abstraction is high [71]. This leads to cleaner models and therefore cleaner target code.

### 3.7.1 Detection of Faults

The detection and repair of a fault is costlier if this defect is discovered at a later time by Boehm et al. [5] and Weigert et al. [79]. Therefore, if a defect can be detected in an early stage of the development phase the costs of fixing it will remain low.

“Our internal data confirms Boehm’s observation that the cost of fixing defects increases exponentially with the distance between where a defect is sourced and where it is discovered.” [79, p.42]

If a fault is detected by tests, then this fault must be tracked back to the source — most of the time this will be a requirement of a specification. In a worst case scenario, requirements must be changed or varied. All steps of the process must be executed again (see chapter 2.2 *The V-Model*).

Weigert et al. [79] state that nearly half of all detected faults are in fact errors of requirements. As these defects are hard to detect (typically through testing at the end of the process) they remain hidden. Repairing them is expensive as the loop of fixing these requirements, implementing and testing needs to be re-run from the beginning.

“Approximately 50% of defects are requirements errors. These are typically the hardest errors to find, and thus also the costliest.” [79, p.41]

Model based development could find these type of errors in an earlier stage as checks could be applied to the model. In this case, these defects could be detected within the specification phase and not only in the testing phase. Figure<sup>6</sup> 3.10 *Cumulative fault discovery rate across the development life cycle* depicts the detection of faults in a model based approach and a traditional one.

“Working with models of requirements enables mathematical techniques to be applied to these models, which enables detection of these hard-to-find defects.” [79, p.41]

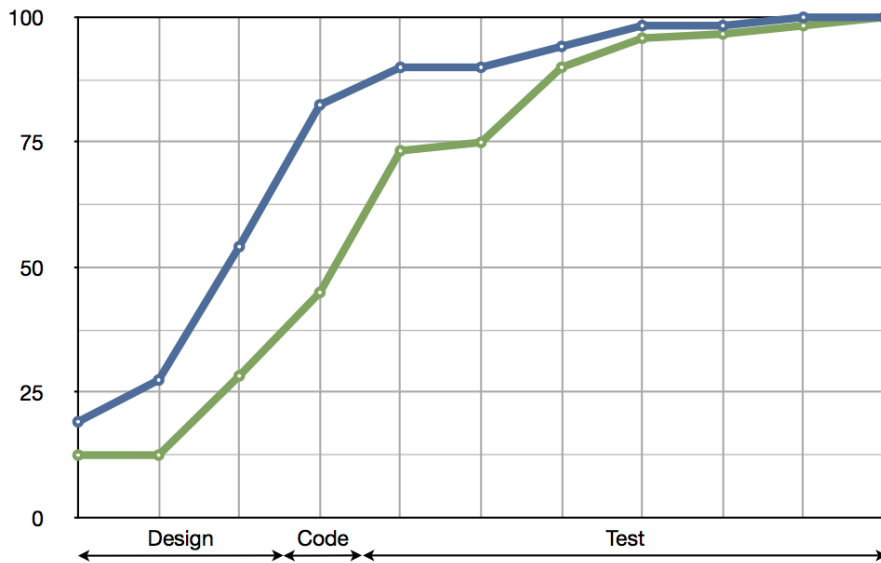


Figure 3.10: Cumulative fault discovery rate across the development life cycle

---

<sup>6</sup>Original image was published in [79].



### 3.7.2 Information and Inspection Cycles

Another observation is that too much irrelevant information is part of the design documents. This means that requirements are specified which are not necessary for the depiction of the system's functionality. This could have a negative impact on the productivity and quality. These superfluous requirements could not be needed in a model driven development as coding conventions and restrictions could be done automatically by mapping the model to code — without manual intervention.

“Following the conventional development process, much irrelevant information — irrelevant from the point of view of system functionality — had to be kept in the design document to ensure that it be considered during coding. This extraneous information negatively affects productivity and quality.” [79, p.40]

According to Weigert et al. [79], the reduction of inspection cycles is also a time saver. Developers tend to need only three cycles when a model based approach is used. Four cycles are needed for a traditional process. Each cycle which can be omitted saves time in the development process.

“On average, developers rely on three inspection cycles of the model instead of four cycles of the source code when compared to following the conventional process.” [79, p.38]

### 3.7.3 Studies

Furthermore, a gain or loss in productivity has consequences on the costs of a project. In Mohagheghi et al. [53] five examples for model based development were evaluated. Three of these projects had a positive outcome, one had no effect and one exertion had a negative impact.

For instance, Motorola could reduce the time for running test scripts by automating system tests. These tests are transformed from models into test scripts which is done by the use of tools and these steps need no (or less) manual intervention.

“In Motorola, by using TTCN scripts, 90% of the tests are automated which has led to a 30% reduction in box-test cycle time.” [53, p.436]

Furthermore, Motorola discovered a reduction in time for the test cycles. A test cycle consists of fixing a fault in the model, creating a test case, generating code from the model and executing the tests.

“[...] the test cycle across four releases of two network features [...] has been reduced from 25–70 days to 24 hours.” [79, p.40]

In a study from the Middelware Company, the task was to develop and implement the same application by two different teams. One of the teams was using a model based approach and the other team used a traditional process.

“The result of this study is the MDA team developed their application 35% faster than the traditional team. The MDA team finished in 330 hours, compared to 507.5 hours for the traditional IDE team.” [67, p.3]

This study was also evaluated by Mohagheghi et al. [53]. It is mentioned here that the used tool made transformation mappings available which might save some time.

“It is worth noting that the MDE team used a tool with pre-made transformation mappings, which relieved them of potential work. On the other hand, this was the developers’ first experience with MDE and related tools, which would presumably hamper their productivity. Issues like application performance and maintenance were not evaluated.” [53, p.438]

Another example for a successful implementation of a model based approach was the reimplementation of requirements. In this case, the productivity could be increased by 24%.

“In WM-Data (desktop business applications), two developers reimplemented a subset of requirements and the effort was compared to some baseline data. The productivity gain was on average 24% using MDE.” [53, p.438]

Not every usage of model based development led to a success. Within an experiment, only eight out of 24 developers could finish the given task. Because of problems with the tools not every task could be performed. Within this experiment, no difference between a model based development and a conventional approach could be measured.

“WesternGeco (oil and gas exploration) performed an experiment with 24 developers who were given four tasks — two involving a traditional development process and two involving MDE. Only eight subjects finished the experiment due to problems with the MDE tooling and complexity of the tasks. The results show no difference in productivity between the two approaches.” [53, p.438]

In another study, a downturn in productivity of 10% was determined. The ascertained decline was even much greater if the used tools are taken into account. Because of not usable tools, the loss of productivity was stated at 27%.

“A team of two developers from Enabler (specialist in creation and integration of IT solutions for retailers) developed a module twice over a period of approximately 300 hours. The results show an overall loss in productivity when using MDE by 27%. When discounting the problems with the use of immature tools, the loss in productivity was 10%.” [53, p.438]

On the other hand, a study by Wienands et al. [81] shows that a model based development increases productivity.

“The evaluation of our elevator controller case study supports previous findings that model-driven development and domain-specific languages increase productivity.” [81, p.466]

Thibault et al. [70] describe their approach of using DSLs in order to generate video display device drivers.

“This demonstrates some of the generally claimed benefits of using DSLs: increased productivity, higher-level abstraction, and easier verification.” [70, p363]

Other studies and experiences of model driven development are mentioned in [69], [60], [39], [21], [4], [2].

## 3.8 Code and Design Smells

In most cases, developed software changes continuously. This could happen if errors are detected and the software must be fixed. If new features are introduced or some parts are removed, the software needs to be changed. Also, if the interfaces of the external systems are revised, the software needs to be adjusted as well.

In these cases, the software code needs to be analyzed and customized according to the appropriate task. This leads either to small adjustments in the code or it might be the case that larger parts need to be restructured. In both cases, the code must be maintainable. This means that not only the code needs to be correct but also in a state that further steps can be processed: static checks might be applied after the code is written in order to reduce the possibility of stack overflows, etc.

“Following their development and delivery to the clients, software systems enter the maintenance phase. During this phase, systems are modified and adapted to support new user requirements and changing environments, which can lead to the introduction of ‘bad smells’ in the design and code. Indeed, during the maintenance of systems, their structure may deteriorate because the maintainer’s efforts are focused on bugs under cost and time pressure, thus with little time to keep the code and design ‘clean.’” [52, p.345]

These so called “code smells” and “design smells” should be avoided in order to reduce negative impacts on the software during its lifecycle. To ensure a good maintainable code basis, these high- and low-level problems must be taken into account when designing the system and developing the code. This means that this issue needs to be addressed in the beginning of the software development.

“Yet, code and design smells are costly because these ‘bad’ solutions to common and recurrent problems of design and implementation are a frequent cause of low maintainability and can impede the evolution of the systems.” [52, p.346]

Regarding that the code is generated from a model, design and code smells must be covered by the code generator. It might be the case, that some design smells can already be avoided in the model or the DSL. It seems unfeasible to state clear rules for the design of the model as the goal for a DSL is to be as close as possible to the problem domain without aiming at the lowest code level. This means that all problems which could occur on code level must be covered by the transition from model to code on the code generator level.

### 3.8.1 Smells and Design Practice

Code and design smells are defined as code which should be rewritten or reorganized. The code itself could be correct and working but might be hard to maintain because of bad structuring. Design smells can be regarded as high-level problems [52] (like antipatterns [8]) in contrast to code smells which are located on a lower level [20].

“Code smells are in general symptoms of design smells.” [52, p.346]

As this does not seem to be a major problem in the first place, it could be a huge drawback through the lifetime of the software.

If a software is delivered to the clients and there are bug reports or feature requests, the software needs to be updated. The code structure could then either support changes in the design or hinder it. Extra effort in methods needs to be invested in order to detect such smells [51]. In many cases, the update cycle could be more expensive if changes cannot be introduced easily. This is because either large parts of the code must be rewritten or the code needs to be changed at a lot of different positions. If tests have been designed to ensure a high level of software quality, it could be that a lot of them need to be redesigned as well. For instance, if branch coverage is necessary, test cases and test data might be updated. As this is a time consuming activity, the costs for the changes increase.

To avoid these problems, several smells have been identified, for instance:

**Large Class** This class includes methods and attributes which are unrelated to each other.

**Lazy Class** This class consists of only a few methods which have a low complexity.

**Long Method** This method has too many lines of code.

**Long Parameter List** This method has too many parameters. A method should have not more than four parameters.

**Speculative Generality** This class was added for future use but has not been regarded by now. This is an abstract class without any child classes.

Avoiding code and design smells is a common practice and it can be assumed that this is implemented in larger projects. A better maintainability of the code during the software lifetime could be expected.

### 3.8.2 Coding Style Guides

The outcome of the analysis of code smells are coding style guides. For instance, software for safety critical systems must get certified before it can be used in everyday life. These certifications consist of several procedures and maturity gates to ensure the product works as specified whatever happens. Then code smells are taken into account and therefore some language constructs are not allowed and others are entitled to be used. These style guides describe best practices and also forbid hazardous constructs.

To avoid common errors in code, different style guides have been established. For instance MISRA C or style guides for aerospace systems (a guideline would be defined

in an additional document (Plan for Software Aspects of Certification (PSAC)), required by DO178B).

Style guides should not only describe things like indentation but also contain conventions for the naming of variables, the positions of operators in joined conditions, the length of a method or the maximum of parameters for a function. This means that a style guide could forbid constructs which can be defined by the target language.

Also, the usage of low-level techniques can be defined here in order to ensure a consistent behavior of the software even if it is developed by more than one developer. For instance, the mechanism of interaction between software modules can be defined or the technique on how information is exchanged. A protocol for exchanging messages can be defined, meaning that a set of transmission routines must be used or avoided. For instance, it could be defined that a signal can only be transmitted by the use of a specified function call of the underlying operating system (for instance: ARINC 653 inter- and intra-partition communication) in order to prevent self-written additional functions.

For instance, the usage of pointers could be restricted or even forbidden. Another example is the dynamic allocation of memory within runtime. This should be avoided in embedded systems as the runtime behavior could change: the memory footprint and time constraints cannot be calculated in advance which is an important task for safety critical software.

Another important part of a style guide is the documentation of the code. A good style guide should define conditions where documentation is mandatory and which degree of information is necessary.

### 3.8.3 Why does code matter even if it is generated?

Even though the code could be generated by code generators from models, code and design smells must be taken into account. It is therefore important that the code follows a style guide. Different techniques of the target language could be permitted or forbidden. Within the aerospace industry, this could be the case if software is developed for aircraft on-board systems.

In many cases, software needs to be certified before going into service, especially when the software is safety critical. For instance, the standard for developing software for the aerospace industry is called “DO178B”. Of course, if the code is used later on for certification it must be compliant to these certification standards. In most cases, style guides are introduced to meet these standards.

Furthermore, it should be in perspective that the generated code is not the last part

in the tool chain and there could be more post-processing steps. The quality of the code must then meet the expectations for the following actions.

In most cases, code coverage must be ensured for safety critical systems in order to check for so called “dead code”<sup>7</sup> or “deactivated code”<sup>8</sup>. A bad code basis could hinder this task as it makes it more difficult to detect such parts in the code.

Regarding software for airborne systems which needs to be developed according to the standard DO178B, decision coverage needs to be ensured. This means that each branching of the software needs to be checked. In case of Design Assurance Level A (DAL A) — the highest level — Modified Condition/Decision Coverage (MC/DC) coverage must be proven. A bad code basis will likely increase the effort which is needed to fulfill this demand.

Code smells are resulting from design smells [52]. To prevent code smells, design smells must be avoided in the code generator. This means, that both smells must be covered by the code generator.

Some violations could be allowed within the coding standard, but each occurrence must be annotated within the code. A code generator must therefore detect such violations and act according to the style guide. For instance, if the style guide limits the length of a function to 200 lines of code and the generated code cannot be split into more than one part or there are good reasons not to try this, a comment must be placed at the beginning of the function. This comment must contain the violation and a reason for it.

Documentation of the code is an important step. If the code is generated, there should be trails to link the code back into the model. If a fault is detected while executing the code on the target, the source code is the first part which gets analyzed. If there are no possible traces into the model, the fault cannot be eliminated easily. The correction must take place in the model and new code must be generated afterwards.

Furthermore, the source code depends on the chosen target platform. For instance, if an embedded system with limited CPU power is used as target, the code is likely to be different as if a target with more resources is used. Efficient code needs to be generated [56].

From these examples, it could be concluded that there are good reasons to have a closer look at the generated code.

---

<sup>7</sup>Code which can never be reached unintentionally is called “dead code”.

<sup>8</sup>Code which was deactivated for a specific purpose is called “deactivated code”.

### 3.9 DSL Phases

Basically, there are three main approaches to create a domain specific language [62] and they differ significantly in effort. Of course, it depends on the circumstances which of the three possibilities is the right choice for the current work.

First, a yet existing DSL can be taken as a basis. This language can be refined and certain aspects are further specialized. Second, essential parts can be extracted from an existing language and taken as a foundation for the new language. Third, a domain specific language is designed from scratch. This means that there are no predefined constructs in this domain which could or want to be reused.

Furthermore, the usage of domain specific modeling could be divided into separate phases. The workload might differ within such a phase regarding the chosen approach but the task description remains the same. Generally speaking, the phases are partitioned into an analysis phase, a design phase and a usage phase [76].

Four key tasks take place in the analysis phase [76]. The problem domain must be identified first. Then, all knowledge which belongs to the domain must be collected and — in a third step — clustered so that specific concepts can be expressed. The fourth step is to use these concepts and define constructs: the domain specific language.

Within the implementation phase, the key task is to create mappings from the created DSL constructs to the target. However, a target must not necessarily be code. This is project specific and might differ. For instance, a DSL could be used to create target code for an application. Also, test data could be generated which will be used by other programs.

In the use phase, domain specific programs are written. The mappings from the implementation phase are used to create a target.

However, Mernik et al. [48] suggest that the analysis phase could also be split into two more parts: an analysis phase and a design phase. This is due to the fact that each of these phases is primarily used by different roles of users. In case of the analysis phase, not only a language designer but also a platform expert is involved. Regarding the design phase, the platform expert is not necessarily needed so the only required role is the language designer. The implementation phase is dealt by the transformation implementer who is responsible for creating the necessary mappings. In the end, the modeler uses the domain specific language to create the models.

In Tolvanen et al. [71] four major activities are described which are necessary for the process of language creation: identification, definition, validation, testing. Identification of problems and concepts are part of the analysis phase. The definition



of the domain specific constructs is the outcome of the analysis phase and takes place in the design phase. However, sanity checks must be done on two levels: the created constructs and also the created mappings must be validated and tested.

The four DSL phases could therefore be pictured like this:

**Analysis** Roles: Language Designer, Platform Expert

Task: Identification and specification of the key concepts

**Design** Role: Language Designer

Task: Creates and defines DSL constructs, validation and test of constructs

**Implementation** Role: Transformation Implementor

Task: Mapping of DSL constructs to target, validation and test of transformations

**Use** Role: Modeler

Task: Uses DSL constructs and creates model

Of course, these phases might not be processed in sequential steps. The outcome of one of the phases is the input for the next phase. If unforeseen problems occur, it might be necessary to re-analyze and go back to a preceding phase. For instance, it could occur that a construct is not well defined and this is noticed in the implementation phase as the transformation mapping cannot be done as intended. Therefore, at least the design of the language must be adjusted and in a worst case, also the analysis needs to be performed again.

### 3.9.1 Analysis Phase

The main task for the analysis phase is to extract the key concepts of the problem domain. In the end, the primary notions for these concepts must be defined.

“Domain analysis is a process by which information used in developing software systems is identified, captured, and organised with the purpose of making it reusable when creating new systems.” [52, p.347]

In the end, “the output of domain analysis consists basically of domain-specific terminology and semantics in more or less abstract form” [48, p.336].

If the modeler, the platform expert and the language designer are not the same person, which is the case for each larger project, the concepts of the problem domain

must be gathered through interviews and discussions. Existing documentation and tools must also be taken into account. The paper landscape method [6] could be used to gain a common understanding of all participating stakeholders.

“The data on DSM development (also know [sic] as method construction rationale) was gathered from interviews and discussions, mostly with the consultants or inhouse developers who created the DSM languages, but also with domain engineers and those responsible for the solution architecture and tool support.” [71, p.199]

“Typically in DSL projects, these abstractions can be elicited from domain experts, technical literature, or by analyzing existing software.” [81, p.456]

### 3.9.2 Design Phase

In case the language will be based on concepts which the domain experts specified already, the definition is considered an easy task. Semantics are specified and might be documented.

“Many of the modeling concepts could be derived directly from the domain model, as could some constraints.” [71, p.202]

In case the language is a graphical language, the language designer should also be aware of the principles of graphical design and capable of creating graphical elements. A solid knowledge of design and a graphical foundation is necessary. The key concepts for creating a graphical language are described in chapter 3.5.2 *Visual Notation*.

“Constraints specifically related to modeling often needed to be refined, or even created from scratch, together with the domain experts. This process was rather easy as testing of the language could easily be carried out by the domain experts themselves.” [71, p.202]

### 3.9.3 Implementation Phase

Regarding a DSL not only as an add-on to enrich a textual system specification but to create more value out of it, the model must be transformed into a target. This is done by manually implementing the model or by applying rules to the model which will transform the objects and relations between them (for instance, code is generated

and could be compiled and executed). Because the “manual transformation is a slow and error-prone task” [74], this should be done automatically.

“This has as advantage that engineers do not have to learn the syntax and semantics of different languages.” [74, p.49]

By applying a concept which will match the model to the target, the developers of the model do not need to know the underlying concepts and languages beyond the transformation. This has the advantage that “formal methods can be applied without having to know their details” [74, p.48]. Furthermore, “[...] users of these languages did not need to have a software development background [...]” [71, p.202], because they do not need to know how the transformation works.

“In collaborations with industrial partners, we observe that they find formal methods useful, but hard to use.” [74, p.49]

Of course, if this mapping from model to target is done automatically, the model needs to be well-formed (see chapter 3.11 *Syntax and Semantics*). In order to ensure this for the model, the visual notation must be formal as well (“[...] visual languages are no less formal than textual ones” [54, p.758]).

Through the higher level of abstraction, the mappings from the model to the target might also not be too close. This means that objects and relationships cannot be mapped simply to a programming language. In most cases there is no 1:1 equivalent from a model element to a language statement. This is not a drawback because this is the work of the generator framework which does the mapping. The generator framework hides this complexity in order to keep the model clean of constraints of the target language.

“Commonalities [of the DSL and the target platform] were usually hidden into the generator or framework in addition to complex issues which can be solved in an automated generator.” [71, p.206]

In addition to the abstraction introduced in the model, this concept should also be used in the generator framework. The objective is to establish similar layers in the generator in order to “make the modeling language and generators easy to extend, allowing the level of abstraction to be raised substantially now, and making it possible to maintain that level in the future” [71, p.206].

“This type of language raises the level of abstraction far beyond programming concepts. Because of this, the generated output could easily be changed to some other implementation language.” [71, p.202]

### 3.9.4 Use Phase

In the end, the DSL is defined and could be used to create models. The mappings are executed and the target is created.

If the target is executable code, it could then be passed to the next steps of the development process, testing for instance. If the produced target are input files for other tools, the next part of the tool chain could be processed (see chapter 4 *Workflow*).

## 3.10 Evolution of DSL

The four stages of DSL development and usage (see chapter 3.9 *DSL Phases*) might not be processed sequentially and only once. Of course, while designing and implementing the language, the roles and phases depend on each other. The outcome of one phase is the input for the preceding phase. Therefore, loops between these phases are common events.

“The design of our DSL has been influenced by a number of roles. Although each role has its own separate tasks, these tasks greatly depend on each other, which leads to interaction between the roles.” [74, p.53]

Furthermore, in most approaches, the first attempt might have some flaws and so the DSL needs to be refined. This could be because problems occur in one of the phases and the work of preceding steps must be adjusted. Also, new features might be needed and should be introduced in the language. In these cases, at least some of the phases must be traversed again.

“Over time, our DSL and the accompanying transformations have evolved. The evolution of the language and the transformations has been influenced by a number of roles, each of which is responsible for performing certain activities that belong to the four phases in the development of a DSL [...]” [74, p.53]

Another example for the evolution of a DSL is simply the point that deficiencies might be noticed when the language is actually used and models are created. It could be that the representation should be adjusted or revised. For instance, if a graphical DSL is used, the representation is an essential part of the language as this is the interface for the user. The interface might need some tweaking and evolves from the first approach to the later used version.

“Graphics are almost always going to improve as they go through editing, revision, and testing against different design options.” [73, p.136]

If new features must be introduced in the final product, the language might be extended as well. Of course, the underlying layers must also be checked and extended if necessary.

“A common solution for [this case] was to make the modeling language and generators easy to extend, allowing the level of abstraction to be raised substantially now, and making it possible to maintain that level in the future.” [71, p.206]

“The difficulty lay in behavioral variability and coming up with a language that supported building almost any new feature based on the common services of the platform.” [71, p.205]

Of course, by introducing too many generic language constructs, the specific properties of the problem domain shall not be put in the background. The language must be extensible but also specific to the domain at the same time. This tradeoff must be balanced while the language is evolving.

Another approach is to build convertors between different DSL versions. The meta-model is analyzed and mappings between both versions is reverse engineered. This can be applied to migrate a model [14].

Furthermore, it is important that the mapping of symbols to concepts is still well-formed if new concepts are introduced. The problem of symbol redundancy, overloading, excess and deficiency must be taken into account (see chapter 3.5.2.2 *Semiotic Clarity*).

## 3.11 Syntax and Semantics

In order to evolve from being “just pictures”, models need a defined understanding between stakeholders (which could be either persons (see chapter 3.9 *DSL Phases*) or other tasks in a workflow (see chapter 4 *Workflow*). Therefore, the description of the behavior and the meaning of objects and their relationships must be downright so that no ambiguities are present in the model. Hence, syntax and semantics — the language concepts and relationships — must be defined in detail. If such a level of detail is specified and the model conforms to this specification, the model is considered to be “well-formed”. The formal specification is called a meta-model.

“A visual notation (or visual language, graphical notation, diagramming notation) consists of a set of graphical symbols (visual vocabulary), a set of compositional rules (visual grammar), and definitions of the meaning of each symbol (visual semantics).” [54, p.756]

The grammatical rules of the language and its structure is called abstract syntax. The constructs and their relationships are described here. The concrete syntax is formed by “the visual vocabulary and the visual grammar [...] of the notation” [54, p.756].

“On the syntax side, we can further distinguish between abstract and concrete syntax. The former denotes the structure and grammatical rules of a language. The latter deals with notational symbols and the representational form the language uses.” [36, p.68]

Also, semantics can be distinguished between static semantics and dynamic semantics. Static semantics describes if a model is well-formed, i.e. if it conforms to the specified syntax. If such a model is translated into an executable program, the execution on the target needs to be taken into account (i.e. how the model is transformed into a programming language and characteristics of the target hardware). This is described by dynamic semantics.

### 3.11.1 Well-Formed Model

A model must be consistent and without errors, if it is used for further processing and not only for a quick overview where not all details must be present.

Vocabulary and grammar must be defined without ambiguities. Also, no lacks within this definition must be present. Each construct of this language should be defined so that a misuse of objects is prevented. A model is called well-formed, if visual vocabulary and grammar are applied to the model.

Such a well-formed model is necessary in order to reduce failures in the subsequent steps of the tool chain (see chapter 4 *Workflow*). This would be the case, if the model lacks information which might be overseen in the model but is necessary later on. For instance, the model could contain not fully specified objects or a transition of the wrong type is used to connect two objects. Not fully specified means, that there are objects with empty but mandatory properties. Also, there could be objects which need to be connected to other objects but the transitions are not present. Another example is guard conditions for transitions which could not be specified correctly. For instance — regarding state-chart semantics — if two transitions have

the same source and destination object and also the same guard condition but a different action defined, no valid code could be produced for a deterministic hardware platform.

If the model is not checked against being well-formed, the incomplete output is then transferred to the test bench. In order to create the target, the output needs either to be processed further or it will be used later on (by another part of the tool chain).

For instance, for running a test, a test procedure needs to be created. As this test step should be generated from the model it depends on the correctness of the generated output. If the model is incorrect or faulty, then the test step will be incorrect as well. If the produced code from the generator is faulty, it is most likely that it might even not compile on the test bench.

If this is the case, the test designer must identify the error and localize it in the model. Therefore, he must understand how the code is compiled, where to find the output from the compiler and how to interpret the logs of the compilation attempt. Then, the error must be interpreted and isolated from the log file. In the next step it must be mapped manually to either the model or the generators and be corrected there. It could also be the case that the compile error can be mapped to different parts of the model or different parts of the code generator. Knowledge of the test bench, the tool chain and compilers is necessary. Hence, the test designer must also be a specialist of the test bench technology, the meta model and the generators. Also, it is a time consuming (and most likely a frustrating) activity. In most cases, the creator of the test is specialized in his test, not the test bench domain.

To avoid these time consuming and costly tasks, a well-formed model must be established before the following steps of the tool chain can be further processed. The conclusion is to restrict the usage of the model elements so that only a valid model can be produced. If this is assured, the described problems cannot occur.

### 3.11.2 Syntax

The syntax describes the rules on how a language is used. This ensures that the structure of the language is used as intended.

“The syntax of a modeling language means more than just reserved words. It is commonly seen as also covering grammatical rules that need to be followed while specifying models.” [36, p.68]

Syntax consists of two parts: the abstract and the concrete syntax. The abstract syntax describes the elements (objects and their relationships) of the language. The concrete syntax is the graphical representation (symbols) of the abstract syntax.

### 3.11.2.1 Abstract Syntax

The abstract syntax describes the elements of the language. Basically, these elements are the objects and their relationships. The term “GOPRR” stands for Graph, Object, Property, Roles and Relationships and is described by Kelly et al. [36]. These are the key elements which are needed to specify a meta-model. Therefore, the complete abstract syntax consists of

- properties,
- objects,
- relationships,
- roles,
- ports and
- graphs

which will be explained in the following paragraphs.

#### 3.11.2.1.1 Properties

All objects have properties. These properties can be a string, some text, numbers, a boolean, a timestamp or a derived element like an object, a relation, a role, a port or a graph.

#### 3.11.2.1.2 Objects

A name is mandatory for an object. It must be assigned to identify it later on in the graph. A textual description can be added if wanted. Properties are then assigned to the object.

#### 3.11.2.1.3 Relationships

With the relationship tool, relations between objects can be defined. In contrast to other modeling tools, different meanings can be assigned to these relationships. For instance, two objects might have a relationship of a type “network” or of a type “power”. This restricts incorrect relations between objects right in the tool and prevents improper usage of objects. This is a very important issue because the test data and the test cases are automatically derived from the model. Therefore the model must be well-formed.



#### 3.11.2.1.4 Roles

Roles are the endpoints of relations. With the role tool, the source and target of the created transition is defined. Like in the relationship tool, a name must be inserted. A description might be added. Within the role symbol editor, the appearance of a role within a graph is defined.

#### 3.11.2.1.5 Ports

With the ports tool, it can be assured that different relationships are bound to the right objects. For instance, if an object does not have a port network, no network relation can be assigned to this object. Also, a power relation cannot be assigned to a network port. This ensures a well-formed graph without illegal relations between objects.

#### 3.11.2.1.6 Graphs

Within the created graph, all defined objects are accessible through the menu bar. Also, all transitions for this graph can be found there. The graph is now filled by drag and drop of the created objects. Relations between these objects are created by connecting the objects. For every test scenario a graph needs to be created.

Graphs are restricted to use only a set of specified objects. This ensures that not related objects are placed within a graph.

### 3.11.2.2 Concrete Syntax

The concrete syntax is the graphical representation of the abstract syntax. The defined elements in the meta-model need to be accessible by a visual formalism. The visual notation is mapped to the abstract syntax in such a way that ambiguities and misuse is avoided. The principles for visual notations must be taken into account [55] in order to ensure semiotic clarity (mapping of concepts to representation), perceptual discriminability (in order to distinguish between objects) and semantic transparency so that symbols reflect their meaning (see chapter 3.5.2 *Visual Notation*).

In addition to the mapping of the concept to the representation, additional information can be introduced into the concrete syntax in order to support the usage. For instance, a transition between two states can be visualized by connecting them by the usage of a line. If this transition is one-directional, an arrow can be drawn at the target end of this line. If more than one type of transition is needed, the lines

could be drawn in different colors, thickness or style, e.g. dotted or continuous (see chapter 3.5.2.4 *Visual Distance*).

“Unlike textual concrete syntaxes, visual concrete syntaxes provide a much richer feature set that can be used to design a usable, efficient, intuitive modeling language. Basic feature include nodes (rectangles, circles, icons, etc.), edges (lines, arrows), regions (complex shapes made up of primitives), decorators (icons, labels, etc.). Many DSLs require customization of visual elements and behavioral adaptation (e.g. conditional display of decorators). Features for concrete syntax customization provided by DSL workbenches are complementary to the structural features described earlier.” [81, p.460]

Other examples of a structural feature for concrete syntax are the concepts of “explosion” and “decomposition”. This means that an object can be partitioned into subgraphs. For instance, this could be used to create an overview graph and — if more information is needed — subsequently exploring the details of the objects by opening a subgraph in order to dive into the details (decomposition). Using explosions means that links are created between objects and other graphs. For instance, a class diagram can be linked to a state-chart (and vice versa) in order to describe the static properties and the dynamic behavior.

“The most favored concrete syntax candidate used decomposition, where a double click on a certain element type in a diagram would open up a subdiagram of a different diagram type.” [81, p.459]

This supports the two principles “Complexity Management” and “Graphic Economy” (see chapter 3.5.2.1 *Principles*). Also the concept of views on the data is supported. By using decomposition and explosions the three viewing depths “distance”, “close up” and “implicit” can be ensured (see chapter 3.6.2 *Views*).

### 3.11.3 Semantics

Semantics describes the meaning of a symbol representation and is therefore an important part of the specification of the model.

“The formal specification of the semantics of a modelling language is a key current issue for model-based engineering.” [23, p.418]

In general, semantics can be divided into static and dynamic semantics.

By using static semantics, it can be ensured that a model is well-formed. This means that a visual vocabulary and grammar are applied to the model (see chapter 3.11.1 *Well-Formed Model*). The dynamic semantics describes the meaning of the execution of a program, the “computational model” [36]. If a model is translated into code which is executed later on, the dynamic semantics is defined in the code generator. As this layer is transparent to the modeler, this might be hidden and not visible in the model. This knowledge is important for the modeler as the concrete usage of symbols and representations will have an effect on the executed software.

For instance, if an array of objects is created in the model and each object acts as an independent actor, several different execution semantics can be applied. One method is to include all objects into one executable and the scheduling is done by a loop within this program. Also, each object could be translated to one executable and the scheduling is done by the operating system. A third common practice would be the usage of threads. If large arrays of objects are used, a combination of these three possibilities would also be valid.

Each solution translates the model correctly into executable code but the semantics slightly differ. Depending on the translation, the characteristic of the execution changes. For instance, if information is transferred between these objects, different kinds of techniques (shared memory, pipes, CORBA, etc.) can be used depending on the execution type (one or more executables). This affects the execution as the different methods differ in memory and time consumption (task switching or operations within the same address space). The scheduling by the operating system or internal scheduling by the executable also affects the memory and CPU footprint.

In the end, at least parts of this knowledge must be present while developing the model. In order to support the modeler, this knowledge should be made visible in the abstract syntax [23]. This means that the modeler can choose which execution semantics is applied in the code generator.

#### 3.11.4 Constraints

Constraints are needed to prevent illegal utilization of constructs and to enforce the intended usage of model elements. Therefore, constraints are part of the syntax and semantics of the model.

Constraints can be established by applying different kinds of concepts. These concepts are:

- regular expressions
- roles and relationships

- templates
- predefined lists (i.e. predefined properties for objects)
- derivation of information from other items (i.e. graphs, objects, properties)

### 3.11.4.1 Regular Expressions

Regular expressions provide a precise means for matching patterns. If the input does not match the pattern, then the input will be rejected. Otherwise, if the input does fit to the specified pattern, the input will be accepted.

For instance, regular expressions can be used to test the properties of a network participant by checking the Internet Protocol (IP) address and port. IP addresses<sup>9</sup> are specified as a compound structure with four numbers, each within the range from 0 to 255<sup>10</sup> [12]. The regular expression for one number is assembled of a set of sub-expressions.

Table 3.1: Regular Expressions: IP Ranges

Range	Regular Expression
0–9	[0–9]
10–99	[1–9][0–9]
100–199	1[0–9][0–9]
200–249	2[0–4][0–9]
250–255	25[0–5]

The regular expression for each part of an IP address can be defined as a compound of these sub-expressions:  $(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9]).(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9]).(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9]).(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])$ .

Also, the check of a port number is specified by a regular expression. Each port number must be in the range of values<sup>11</sup> from 0 to 65535 [13]. Again, the expression is assembled of a set of sub-expressions:

The regular expression for the check of the port can be defined as a compound of these sub-expressions:  $(6553[0-5]|655[0-2][0-9]|65[0-4][0-9][0-9]|6[0-4][0-9][0-9][0-9]|[1-5][0-9][0-9][0-9][0-9]|[1-9][0-9][0-9][0-9]|[1-9][0-9][0-9][1-9][0-9]|[1-9][0-9][0-9])$ .

---

<sup>9</sup>In this case, only IP Version 4 is regarded.

<sup>10</sup>No distinction is made regarding the special IP address spaces described in RFC 3330.

<sup>11</sup>No distinction is made regarding ports below 1023 (like described in RFC 6335) as internet services like DNS, etc. are not needed in a testing environment or are described in the test bench manuals.

Table 3.2: Regular Expressions: Port Ranges

Range	Regular Expression
0-9	[0-9]
10-99	[1-9][0-9]
100-999	[1-9][0-9][0-9]
1000-9999	[1-9][0-9][0-9][0-9]
10000-59999	[1-5][0-9][0-9][0-9][0-9]
60000-64999	6[0-4][0-9][0-9][0-9]
65000-65499	65[0-4][0-9][0-9]
65500-65529	655[0-2][0-9]
65530-65535	6553[0-5]

### 3.11.4.2 Roles and Relationships

Relationships specify the context between objects. Of course, different objects might have different relations to each other. These must be expressed by different kinds of relationships.

Each relationship describes a certain aspect on how the objects are related to each other. By specifying relationships the concrete context between objects is defined. Each context should be presented by a relationship. This means that there could be different connections between the same objects and each of these connections is specified as a relationship.

A role describes a sink or source of a relationship. This is important as not all relations between objects are bi-directional. For instance, a “child-of” relation consists of a “parent-role” and a “child-role” so that it cannot be mixed up by mistake.

### 3.11.4.3 Templates

A template specifies a frame but leaves the particular features to the modeler. By using such a template, the realm of an object is defined. For instance, all properties are already added to this object and default values might have been defined as well. Also, in case of a template for a network message, the items of the header and payload (like type, length and position) are specified. This eases the work of the modeler by applying predefined templates instead of defining everything over and over again. Also, this ensures that an object or message meets certain standards and reduces misconfigurations.

#### 3.11.4.4 Predefined Lists

If possible, predefined lists could be used. In some cases, an editable text box can be exchanged to a drop-down list with a fixed list of entries. This minimizes the usage of free text input.

Using text boxes leads to the need of syntactical checks of the entered text. For instance, the entered characters need to be checked if lower and upper cases are mixed. The characters must be converted before the input can be processed. Also, it could be that the user entered text with a typo so that the input cannot be matched.

For instance, a graph contains objects and there are transitions between these objects. A message type is attached to these transition links, so that the succeeding state will take the message type as an input for further processing. As all possible message types are specified as templates, there is no need to let the user enter the name of the message type. Instead, a predefined list is presented where the appropriate type could be selected. This eases the work for the modeler because no wrong input can be given. It also reduces the effort to check and fix the model.

Another example is to predefine the properties of an object. For instance, the sex of a passenger is reduced to the choice of being “female”, “male” or “other”. There is no need for the modeler to enter the same text over and over again. Instead, the right property is chosen from the drop-down menu entry.

#### 3.11.4.5 Derivation Of Information

As information should be defined only in one place, this does not limit to show data in several places. This explicitly means that the shown data should be either read-only or it needs to reflect changes back to the original position (or all other positions where this piece of information is present).

For instance, several objects are present on one type of graph. Values for the properties have been assigned so that this graph is well-formed. Some parts of this data is needed on another graph as well. Instead of entering the same data on a different object on the second graph, the data from the first graph is re-used and displayed instead. This ensures that the same data is used later on. For example, if data on one graph is altered, it is reflected on the other graph instantly. If it was possible to enter the data in both graphs, the user must ensure that he will keep the data synchronous. As this is a manual and therefore error prone task, this should be avoided.

### 3.11.5 Syntax and Semantics Checks

The earlier a software fault is detected, the less costly it is to fix it. If a fault is detected during a system test, a lot of effort needs to be put into repairing it. In most cases, not only the software must be adjusted but also the according documents. Hence, each fault which is detected at the earliest possible state in the development saves time and money (see chapter 3.7.1 *Detection of Faults*).

The first step where faults can be introduced is the model itself as this is the basis for all further activities. It is therefore important to identify all — or at least as many as — possible faults in the model.

“Model analysis attempts to uncover model defects by looking for violations of the semantics of the underlying modeling language. While such analysis techniques are independent of what the model should do, they are quite fruitful at detecting situations where the model will do something that it should not do.” [80, p.167]

“Our experience has shown that testing done on the models during the design phase is typically done to various levels of completeness, and that the more model testing that is done, the more defects that are found.” [80, p.169]

Not only can a fault be detected within the testing phase, errors could also occur at the compiling stage. For instance, if the generated code contains errors the compiler will discontinue the compiling process and display an error. This error then needs to be mapped to the appropriate part in the model. If the code generator is well-known this might be a tiresome task but manageable. If the code generator is not known, chances are that this error could not be mapped to the model. Therefore, the generated code must — at least — be able to compile. For this to happen, the code generator needs to check the input (in this case a model) to ensure this task.

Assuming the code generator would check the model for its syntax and semantics, the model would only be checked when the code needs to be generated. If the generator would detect an error in its processing steps, the user would need to fix the model and then repeat the compiling step. This could lead to some reiterations until the model can be processed by the generator. In the worst case, large parts of the model need to be adjusted in order to satisfy the transformation rules.

Therefore, it would be helpful if these checks are introduced even before the generator process. The model should be checked while it is created. The validation is done at the earliest level.

As the syntax and the semantics are defined, a check can be implemented so that

the model is examined constantly. This static analysis performs checks on the model without executing it (in contrast to testing).

In order to perform such a static analysis of the model, two main possibilities exist. Either the used modeling tool supports rules which can be applied so that a violation is detected. The tool must then be able to import model specific rules in order to support each kind of model. Another solution is to use some kind of “test bench ready” object. This means that a special object is created which will run the checks for one or more graphs.

In both cases, it needs to be checked if all constraints are valid, if all objects are used according to the syntax and semantics, if configurations are valid, and so on. Regarding syntax checks, each state in a state-chart like diagram is verified if it has an outgoing transition (except for the end state).

Also, a combination of syntax and semantic checks could be run. For instance, a graph contains templates for messages, message sections and databases. A syntax check would validate the message if the subsections of the message are jointed properly. A semantic check can prove that this message could be stored in an attached database. If both checks are successful, the graph can be regarded as well-formed.

These violations must then be made visible within the model. This could be done by highlighting the violations. For instance, a semantic inspection could check if an object contains invalid properties. If such a violation is detected, like the misuse of an object or inconsistent properties, it should be “flagged” [62] in such a way that the modeler can fix this issue immediately.

Of course, if a model is created from scratch or large parts are changed, there could be a lot of violations. In this case, the modeler is aware of this and it would hinder him if the checker would flag everything. Therefore, a global switch should be present in order to disable the constant checks for some time. The development is therefore supported but not slowed down.

## 3.12 Versioning

Versioning is a very important aspect as all test results must be able to be mapped against specified requirements and/or code base. Of course, a test result is useless, if the state of the target cannot be determined.

As each model consists of several parts, for instance the meta-model, the model itself, the code generators and maybe more. There must be either one global version for all parts or each part needs its own version. It could be the case, that the meta-model is extended due to new functionalities but the model remains the same, also the



model could be extended using the defined meta-model. Therefore, different versions for model and meta-model must be possible.

For instance, for regression tests it can be very useful to re-run or examine the previous test suite. Therefore, the exact same model and the exact same meta-model needs to be present. If a test was run or documentation was created, the version tags for both types of models must be mentioned.

### 3.12.1 Versioning of Meta-Models

The meta-model defines all aspects of the used language which is used to express the problem domain. Here, the mechanisms of the model creation are defined. Hence, it is important to version the language constructs to ensure that the used methods and objects have the same meaning. For instance, the semantic must be well-defined and unambiguous.

It is very important that the created models depend on the same meta-model. Else, due to possible different semantics, the models could look the same but have a complete different meaning — and therefore different target code and a different execution model.

### 3.12.2 Versioning of Models

It is quite straight forward that all created models need to be stored with an attached version identification. The tool chain starts with the models, so the following steps depend deeply on the first one.

Everything that is created at the end of the tool-chain is extracted information from the model. Therefore, all test cases, test steps, test data and all abstract machines are derived from the model. To ensure re-runs of tests, the produced code must be exactly the produced code which was used the last time.

### 3.12.3 Versioning of Code Generators

The code generator extracts the necessary parts from the model and transforms the information to text (model-to-text transformation). The produced code shall be product quality like. No code needs to be edited. If the generated code does not fit exactly the needs, the code generator should be edited. Therefore, it is an important task to version the code generators as the code depends directly on the quality of the code generator.

#### 3.12.4 Versioning of Code

As the code is produced directly from the models through code generators, there is no reason to version the produced code. Instead, the models and the generators need to be versioned. Hence, the version of the code depends on the version of the code generator (and of course of the version of the model and meta-model).

The code itself depends on the quality of the code generator. It is produced completely from the model. No manual interaction during the compile process is necessary and should not be made.

#### 3.12.5 Achieving Versioning

As MetaEdit+<sup>12</sup> does not store the items of the models in single files but some kind of internal database, the whole project could be checked in and out using a standard repository tool. As the project consists of the meta-model, model and generators, all parts are in sync because there is no possible way to mix a non-fitting meta-model with a model or generator. This ensures that each part of the checked out model has got the same version as the previous used model. If a scenario is set up and every part of the model is defined, the whole model is checked into the repository tool as such. This state needs to be tagged so that it can easily be retrieved and restored. This procedure makes it very easy for managing whole models. If it was ensured, that only fitting parts of the model are checked in, this solution could be easily applied.

On the other hand, it makes managing the model and meta-model more complex. The only possible trace of meta-models and models is to check the comments and tags in the repository. This is because the meta-model is tightly integrated into the model and cannot be managed individually.

For instance, one scenario is set up and stored into the repository tool. The tag for this check-in is set, so the combination of model and meta-model can be retrieved by a check-out. A couple of days later, the meta-model is refined. The model itself has not changed and stays the same. The whole project is checked in again and another tag might be applied. Another couple of days later, the model is refined but the meta-model has not changed. Everything is put back into the repository by checking in the project. The meta-model and the model are perfectly in sync and tagged correctly.

A problem occurs if the project needs to be split into different branches. For instance, in larger projects it is quite common to have a stable branch for the customers and

---

<sup>12</sup>For this work, the single user version of MetaEdit+ was used. However, a multiuser version is offered by MetaCase but is not evaluated here.

a development branch where new features are tested. If the meta-model and model are stored together, there is no chance to create a second branch and mix together parts of both branches later on.

A solution for this problem is to try to separate the meta-model from the model. In this case, the meta-model can be changed and applied to the model later on (and vice versa). The key is to extract the meta-model into a patch file and store this into the repository tool. This must be done as a separate project so that different tags can be specified. This needs to be done every time the meta-model changes. If the new meta-model needs to be applied it can be checked out from the repository. The last step is to import the file via the internal patch command.

Of course, it can be tricky to import a new meta-model to an existing model. If the meta-model is only extended by new features, the model itself must not be changed necessarily. Handling extended objects is a simple task. A problem could occur, if a feature of the meta-model was removed (for instance, a property was moved from one object to another one). Applying a reduced meta-model is a crucial task. Code generators or other objects might rely on the previously stored information. Some parts of the model could behave differently than expected. These problems cannot be solved automatically as there is no standard workflow to handle this kind of task. The model needs to be checked manually for these changes. Instead, the developer of the models and meta-models needs to be very careful to ensure a model without ambiguities.

Storing the meta-model and the model by its own is needed for two reasons. First of all, a modified version of a meta-model could be needed on a former model. Instead of adapting all changes of the meta-model manually (which could be an error prone task), the patch is applied and the new features of the meta-model can be used immediately. Also, if the meta-model shall be used in more than one project or more than one developer is working on a model, the meta-model can be transferred between two different projects. The second important part for versioning a meta-model as well as the model is the documentation. For instance, each test run needs to be documented. Also, the components of the test and the test bed need to be stated (e.g. compiler and generator). The versions of the meta-model and the model are an important information (because this is the root of the test and everything else depends on this versions) and needs to be printed in the documentation as well.

The versions of the meta-model and the model should be made visible within the model. The developer can ensure that the right versions are used by simply checking this type of information. There are several possible solutions and the usage depends on the setup of the used configuration setup and repository tool. If the repository tool stores the current checked out version in a file which can be parsed, a “graph information object” can display this information (see chapter 6.11.3 *Graph Information Object*). This object can be placed onto a graph and the user can read

the actual version identifier for the meta-model and the model.

If there is no way to retrieve the version identifier automatically, both values need to be stored manually. This can be done by adding two more attributes to each graph or “main” graph, if it is possible. These values need to be adjusted by the developer each time a new meta-model is applied or the model is changed and stored into the repository. Maintaining these baseline strings manually is an error prone task because it must be ensured that this update is done each time the repository is used.

### 3.12.6 Model Compare

As described in chapter 3.12.1 *Versioning of Meta-Models*, versions of meta-model and model are essential to keep track of the current development and the test campaign later on. Unfortunately, viewing the differences between versions is not a trivial task and tool support is limited — in best cases.

“Furthermore, it is generally more difficult and expensive to develop tools for modeling languages than for programming languages, due to the usually more sophisticated semantics behind many modeling language constructs. For example, a tool that compares two different versions of a state machine model must ‘understand’ the semantics of state machines.” [62, p.299]

The tool can only support a comparison between models if the semantics are also comparable. This means that the model does not only consist of its own meta-model but also a ruleset that describes which properties of an object can be compared.

“In contrast, because programming languages are mostly textual, the differences are usually expressed in terms of lines of text that have been added or changed, without any concern for semantic constructs, such as states or transitions. The semantic interpretation of such differences is left to the programmer. Unfortunately, this kind of semantics-free differencing is not practical for graphical languages.” [62, p.299]

Brosch et al. [7] propose a “conflict-tolerant model versioning”. This means that conflicts can occur during modeling phase and are not corrected immediately. These issues are solved “later in a collaborative setting”.

The concept of using abstraction so that “a large model can be decomposed into a set of related units” [42, p.336] is described to be a solution in order to enable concurrent development. These units are then versioned.

The following two examples indicate that it seems to be difficult to have a generic workflow to display the important differences between two diagrams. Each DSL must have some kind of “ruleset” which defines semantic changes between two versions. Furthermore, this ruleset must be supported by the modeling tool.

### 3.12.6.1 First Scenario

In this case, there is a UML state-chart like modeling language which contains only states and transitions between those states. As there is no rule if a start state must be located on a predefined position in the graph, this object could be placed everywhere. Its position has got nothing to do with the semantics of this state. In fact, as there might only be a rule of thumb, the start state can be placed in such a way that the diagram is more readable. Prochnow et al. [58] showed that a diagram can be understood more easily if a start state is located on the top left corner and the end state should be located in the lower right corner. The states in between should be placed from top to down and from left to right — like the written natural language<sup>13</sup>. Subsuming, the placement of objects is completely up to the developer.

Also, the size and color of the objects and transitions do not carry any information. Size can be used to emphasize a part of the diagram but this does not change the semantics. The length of a transition is also irrelevant. The transition should be placed in such a way that the diagram is “readable”, this means that they need to be distinguishable between each other and should be visible and traceable.

As the placement of objects has got nothing to do with the semantics, the difference between two diagrams does not depend on the layout. It depends on the semantic changes like changes of objects and transitions. For instance, it does matter if a guard condition or action is changed on a transition, if a transition is removed or added or if an attribute of the object is changed.

These changes must be made visible so that the developer can view the differences.

### 3.12.6.2 Second Scenario

In this case, a DSL defines a layout for a circuit board. Here, the layout is very important. The length of a transition between two objects (Integrated Circuit (IC)) defines the length of the real connections between two ICs. Therefore, the time a signal needs to be transmitted from one point to another depends only on the length of the transition.

---

<sup>13</sup>Of course, this seems to depend on the region. This could be different in other than the so called western part of the world.

The size of the IC is also relevant. If an IC is large and the connections of two other ICs must be routed around this IC, this affects the length of these connections. It could be that these two ICs must be placed elsewhere on the circuit board or another layer must be introduced so that the connections could be placed below the larger IC.

As the semantics of the model depends on the layout, the view at the differences of two diagrams must show exactly the change of positions.

---

---

## CHAPTER 4

---

# Workflow

In this chapter, the workflow and the necessary steps to import the approach of End-To-End Chain Testing into an existing development process are described. First, a basic view of the workflow on how a domain specific language is used is presented. In the next step, this workflow is extended in order to establish a more complex tool chain. The used tools enrich the process of generating the target. Not only the binary target is created but also the complete environment like test procedures, test data and simulations.

In order to create larger or safety critical software projects, each company must ensure that the software is created in compliance to a predefined process. In most cases, this process is a well established procedure within the company. Hence, the new workflow must be embedded into these existing development processes. If this can be guaranteed, the workflow can be applied. It is not likely that an established development process is changed significantly in order to implement a completely new method.

### 4.1 Generic View

The generic view is depicted in figure 4.1 *Generic Workflow*. Graphs with embedded objects and their relations are specified and implemented. In the end, a model is designed. This model is then transformed by one or more generators (model-to-text generator). The graphs and objects are parsed and translated. This output results in one or more files. In order to create the target, a framework might be used. This framework could consist of libraries (for instance, standard functions or already developed functions for an earlier project) which are used in conjunction with the created files in order to compile a target software.

This workflow does not contrast to the DSL phases (see chapter 3.9 *DSL Phases*). It focuses on the utilization as it describes the “Use Part” of the DSL phases. The analysis, design and implementation is subsumed in the layer “DSL”. The constructs are already defined and implemented. The transformations are specified and integrated.

The framework layer may contain additional tools and processing steps which help

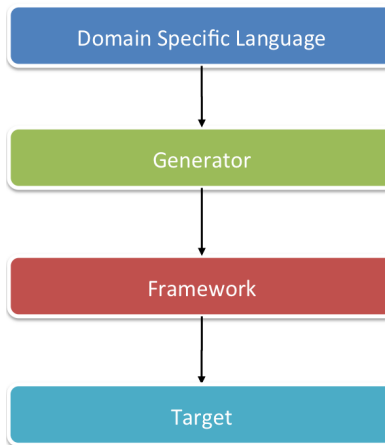


Figure 4.1: Generic Workflow

to enrich the target. In most cases, an additional library can be used to compile the target. The concrete characteristic might vary within different projects.

## 4.2 Detailed View

In this chapter the workflow is explained in detail. The general workflow is described in figure 4.1 *Generic Workflow*. Starting from a defined Domain Specific Language (DSL), generators extract the information of a model and transform it — with the help of a framework — into a target. This can then be executed or further processed.

The objective of the workflow is to establish a process which adapts the right tools for the specific jobs, enable a switch between tools, attach the necessary jobs to the tool chain and automate this procedure. This means that all necessary steps should be processed by a tool chain which is not visible for users as everything is processed in the background. Manual intervention of a user is neither needed nor wanted.

Each of the layers of the general workflow have been extended to suite the needs of this goal. As the model is processed automatically within the tool chain, it is necessary that it is well-formed, meaning that the model is syntactically and semantically correct (see chapter 3.11 *Syntax and Semantics*). This is done by the use of checkers within the modeling tool (see chapter 6.12.4 *Check Model*). They ensure that the model can be processed further and will not create a fault within the next steps of the tool chain.

The DSL parser traverses the model and extracts the necessary output for the specific jobs. In this case, several different targets are needed. A set of generators is used, each for a different purpose (see figure 4.2 *Detailed Workflow*).



For instance, test procedures and test cases are created. In order to execute these tests, simulations might be necessary to stimulate the tests. As the necessary information for these simulations are already specified within the model, it needs to be extracted as well and transformed into a simulation. Also, an oracle for the tests can be extracted from the model as the basis for a simulation and the oracle are the same (because the model is the specification for a specific system). The check conditions need to be extracted and applied to the oracle instance. Another job is to extract test data which are used by the simulations and the oracles (see chapter 8 *Test Data Generation* and chapter 9 *Creation of the Target*). The test data are read by the simulations as an input and they generate stimuli for the system under test. Also, the oracles can calculate the expected behavior by applying the test data to the specified behavior which is described in the model. At last, so called Interface Modules (IFMs) are created which are part of the test bed. IFMs are used as an abstraction to the underlying hardware, meaning that a unified access for interface cards like AFDX or Controller Area Network (CAN) is established. As the configuration of these interfaces are also stored within the model, these parts of the test bed can be extracted as well.

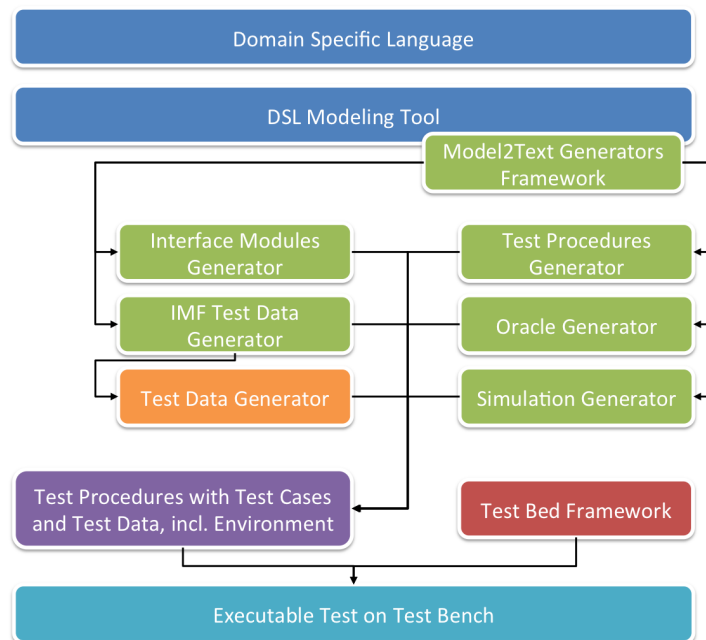


Figure 4.2: Detailed Workflow

A very important abstraction layer is the usage of signals. A signal represents a piece of information which is routed between different systems. All these systems might be connected by different links (see chapter 5.3.2 *Test Bench*). The information itself shall be defined independently of the link within the model, meaning that only

the type and value of the signal need to be specified. The transport of the signal is invisible and transparent. No hard-wiring is done within the model.

For instance, a passenger (see chapter 6.3.1 *Passenger Object*) has a name, an ID and a preferred language as properties. Each property is coded with one unique signal. The signal name itself is compounded of the unique internal model ID of the passenger and the name of the property. If this information is used by a system which expects an XML payload within a SOAP message, the signal will be mapped into an XML template and routed via the connected link (ethernet, for example). Another system might need only the ID in a 10bit Integer format. The signal is then mapped into the target datatype and maybe sent via another link (WiFi or bluetooth, for example).

The advantage of this approach is to quickly create different test scenarios without configuring the test bench manually. Also, if the protocol for a specific link is changed, only the signal definition must be adapted. The rest of the model remains unchanged. This increases the amount of executable test steps per time significantly.

Also, taking into account that the environment might be under development, changes can occur in these systems. By only changing the interface description through manipulating the signal attributes, the adaption can be made quickly without changing major parts of the test bed or the test environment.

If the model is created, the testing phase can be started. The model is parsed and the necessary parts are translated to target or intermediate files. These files are checked into a repository and are tagged (as “source”) afterwards.

On the test bench, the files are checked out and the files are placed in a temporary directory. A tool chain takes care that all necessary additional files are created, for instance the test data (see chapter 8 *Test Data Generation*). The created files are then placed into the test bed. Also, the test procedure is compiled and the environment is set up by using the configuration from model via the temporary files. In the end, all files are checked in and tagged (as “final”) again.

These two tags ensure that two states exist in the repository. First, all sources which are extracted from the model are tagged as “source”. Second, the complete test campaign, consisting of all test procedures, test cases, test data, simulations and all other parts of the test bed, is tagged as “final”.

This ensures that a certain test campaign can be re-run at any time. In such a case, the “final” tag is used to extract all necessary files from the repository and have the exact same state as in the previous run.

While the tests are executed, the actual state can be lifted up in the model again by using the interface of the modeling tool (in this case the SOAP stack of MetaEdit+).

If the test campaign is finished, logs and test reports are created. Also, the traceback of the covered requirements needs to be completed. There are two possible ways to assemble this kind of data. First of all, the data is collected while the test is running and pushed back directly into the model through the interface of the modeling tool. The second solution is to parse the log files and use the requirements coverage data to fill the properties of the model. In case of this work, both methods are used.

These documents are stored — together with the test campaign and all its associated files — in the repository with the tag “test complete”. Therefore, all needed test files and test results are now stored in the same place in order to document the run of a test campaign.

## 4.3 Development Process

Generally, when systems and applications are developed, they are created according to a well-formed process within the company. For instance, the procedures of Airbus are documented in two major documents, the Airbus Directives ABD100/ABD200 which conform with the standard DO178B. Within these documents, the process is described in detail.

Introducing a new approach of testing or developing cannot be done without such a process in mind. In most companies, a development process is a well established procedure which was developed over a long time period. Therefore, such a process is unlikely to be changed quickly in order to introduce a new approach. A process framework for adopting model driven development to achieve a clear system development process is described by Mansell et al. [46].

Hence, for acceptance reasons, the new approach must be embedded into all necessary steps of the existing process.

Speaking of safety critical systems, the tracing of requirements is an essential part of such a process. It must be ensured that all requirements are implemented and tested in the final product.

### 4.3.1 The “Standard” Development Process

As the method of tracing requirements is mandatory for developing certain critical safety systems, this procedure cannot be questioned but must be applied. The first step for such a development is to create a development plan (PSAC in DO178B). In this plan all necessary tools and planned steps are described and each deviation from this plan must be explained in detail in order to gain certification.

The development consists of four main tasks: the requirements phase, the design phase, the implementation phase and the verification phase. These phases are depicted in figure 2.1 *The V-Model*.

Each of these phases is coupled by a process. In order to switch from one process to the next one, a gateway point must be passed. This maturity gate can only be passed if certain, predefined transition criteria are met. With each step of refinement of the requirements, so called maturity gates are established. These gates should ensure that each phase is controlled and handed over properly to the next phase. In many cases, the transition can be done if reviews of documents or code are completed successfully. Such a procedure is described in the Airbus Directives ABD100/ABD200 and DO178B.

It starts with the high-level requirements down to low-level requirements which are the basis for the final product. The low-level requirements are a refinement from the high-level requirements. Also, the high-level requirements are the basis for the high-level test cases. These tests need to fulfill that all requirements are present and work like intended in the final product. In addition, low-level tests — which are derived from the low-level requirements — ensure the functionality on a lower basis.

Traceability matrices need to be created to ensure that the mapping between high-level and low-level requirements is complete. Also, the mapping of the requirements to test cases must be ensured.

## 4.3.2 Documents and Requirements

In order to record all results of the implementation and test activities, documents must be produced. In most cases, the documents are not written at the end but between steps of the process. Therefore, it is an important task to take care that these documents are synchronized with the current work as they document the actual state of the project.

### 4.3.2.1 Inconsistency between Documents

In most cases, documents are not consistent at the end of a project. This is a common problem. It results of the fact that information is specified and documented at a lot of different places. For instance, if a high-level requirement is refined, the original requirement is still valid in the original document. However, the refined requirement is stated in another document. Sometimes models are used to clarify the written specification. Although, it is rarely the case that these models are much more than images, created by a modeling tool, the information must stay synchronized. Therefore, if a requirement is changed, the models must be adjusted accordingly.

Due to time considerations, these changes are not transferred into all applicable documents. This leads to a lot of fragments of information which depict the final product.

“During implementation, models are no longer updated and are often discarded once the coding is done. [...] The cost of maintaining the same information in two places, code and models, is high because it is a manual process, tedious, and error prone.” [36, p.4]

“Finally, defects are repaired at the level of the hand-written code resulting in design documents that become hopelessly out of synch with the code and become incomplete or, worse, misleading.” [79, p.36]

Each of the originated fragments depicts a part of the final product. This leads to the fact that there are discrepancies between these artifacts. As there is no instance which could expose this problem, the inconsistencies remain hidden.

#### 4.3.2.2 Relations between Model and Code

Five different types of relations between a model and the code are described by Kelly et al. [36]. The first type depicts a smaller project where the functionality is not too complex (see (1) in figure 4.3 *Relations between Model and Code*). As everything can be done directly in code, there seems to be no need to create models to express the functionality. Therefore, the instance “model” is omitted here.

If the project is more complex, developers tend to create models (2). However, these models are not treated as part of the code because code and model are regarded as separate instances. Models are used to explain concepts of the system design at a higher level of abstraction. If the code needs to be changed or extended, the models might need to be changed as well. As there is no direct link between both entities, this task is discarded as soon as the coding is finished.

Models can be used for documentation reasons (3). In this case, the development phase might be finished already and the models are created afterwards. Models are then used to visualize the code. Kelly et al. [36] point out that these models are typically not used for further processing as testing and debugging would take place on the code level.

Relation (4) describes a round-tripping process. In this case, information is stored in two different places. The information can be translated from one specification into the other one. Of course, this is only the case if these two mappings could be established. It is stated that common procedure could be the creation of a database schema from a model or the creation of a schema model from an existing

database. Kelly et al. [36] explain that this procedure is rather limited and only static information could be translated. Dynamic actions like interactions and behavior could not be covered and this needs to remain in code. Therefore, the connection between the model and the code is represented in dotted lines (4).

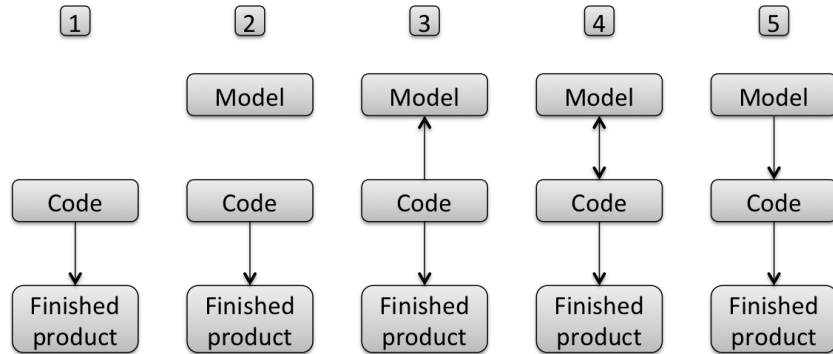


Figure 4.3: Relations between Model and Code

The last type covers a process where the model itself is the only source (5). Through mapping the model to code, the finished product is generated. In this case, no manual interaction is needed. As the mapping is done automatically, there is only one document which needs to be changed: the model. For instance, if the model was extended the code does not need to be modified. The model is first compiled into code and then into the target.

Because of this mapping and the concept of a single source, there is no need to sync different kinds of documents. The additional step of keeping the documents in shape can be omitted.

#### 4.3.2.3 Relations between Model and Requirements

In most projects, there are three possible approaches on how models are used for requirements. These possibilities are depicted in figure 4.4 *Relations between Model and Requirements*.

“Model-based development (MBD) is a specification-driven approach to engineering complex software systems. It relies on design artifacts called models, which are specifications interpreted against a fixed and formalized context.” [35, p.72]

Models can be used as an add-on to requirements. In this case, the model clarifies one or more requirements. There is only a loose connection between the model and

the textual requirements. If one of the requirements needs to be changed, the model must be updated accordingly. This is a manual task and needs to be performed where appropriate. This approach is depicted as (1) and (2) in figure 4.4 *Relations between Model and Requirements*. The difference between (1) and (2) is the coupling between the model and the requirements. In case of (1), the model is included as an image. It was drawn in a modeling tool and then exported as a picture. This picture was then imported into the requirements. Of course, the image could be used to explain the system — but no more use could be extracted. In case of (2), the model is not only an imported image. Here, the model is an embedded object and links could be attached from the requirements into the model and vice versa. The drawback in this case is that these links have to be created manually. This means that this task needs to be done every time a requirement or the model changes. Both methods lead to a better understanding of the system but the task to keep the documents updated is time consuming and hence expensive.

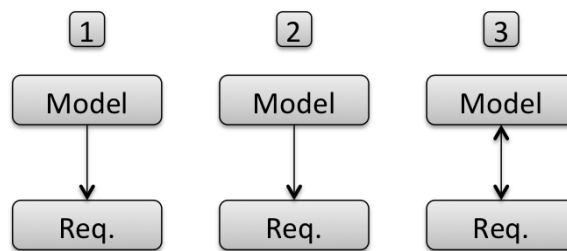


Figure 4.4: Relations between Model and Requirements

The best solution would be if the model and the requirements were equivalent. This is depicted as (3) in figure 4.4 *Relations between Model and Requirements*. Two possible scenarios are applicable here: either, there is an automated mapping between the textual requirements or the model itself consists of all requirements and no additional textual requirements are necessary: the model itself is the requirements document [9]. In this case, no additional textual document is needed, as all information is stored within the model and could be retrieved from there. Of course, in both cases, no manual interaction to keep different documents in sync is necessary. Either this will be done automatically or it is not necessary as only one document exists.

### 4.3.3 Solution

If a new method is supposed to be introduced, it must fit into the development process of the company. Also, a benefit for the user should exist, if the new method is applied. This is a very important step for acceptance and for the certification of the product at the end of the development process.

If a model is used as the basis for the development, the requirements might differ from a traditional approach. The focus must be put to the questions on where the requirements are stored, how the traceability is achieved and in which way the documentation is created.

The solution is to achieve two main tasks. First, all the requirements must be traceable — from the high-level and low-level requirements to the test verdict. Second, it must be possible to create the necessary documents and record the development phases (see chapter 2.2 *The V-Model*).

The most important task is to ensure that all the requirements are traceable. It needs to be assured that all applicable requirements are implemented and tested in the target. Also, if a requirement is deleted, the associated part in the target needs to be removed and the tests need to be adjusted. Both scenarios must be traceable, meaning that a mapping from the requirements to the target and also back from the target to the requirements must be established.

In most cases, two main types of requirements exist: high-level requirements and low-level requirements. Regarding the V-Model (see figure 2.1 *The V-Model*), high-level requirements are created and the low-level requirements are refined from these high-level requirements. Such requirements must also be traceable, meaning that a mapping from any type of requirement to the other must be established.

This must be done for certification reasons, as a gapless tracing is essential.

#### 4.3.3.1 Mapping of Requirements

Regarding model-based development, the requirements might not be written down in natural language. This means that the high-level requirements can also be specified as a high-level model. Each element in the model has properties which describe a certain aspect of the system. Additional properties are used for maintaining the model and its requirements. For instance, a property which holds the high-level requirement ID is added to all objects.

Another property is added which specifies the low-level requirement ID. This ID is added in the low-level model. While refining the high-level model the high-level requirement IDs are copied. This means that a tracking from high-level to low-level requirement ID is a trivial task as both IDs are located in the same model (low-level model). Also the tracking back from the low-level to the high-level requirements can be accomplished by extracting the IDs and create a tracing matrix. These tasks can be automated so that no manual tracking is necessary.

The tracking, if all low-level requirements are located in the target code, is a trivial



task as the code is generated from the model. This means that the code generator needs to extract the requirement IDs and place them into the code. In most cases, this can be accomplished by adding comments to the code with a predefined tag so that a filtering for IDs in the code can be done without great effort.

When the target is generated and the environment is set, the test campaign is started. As the software integration tests (SWI) and hardware-software integration tests (HSI) are executed automatically (no manual test steps), the test verdict can be computed for each test step. If the test procedures are coded manually, the low-level requirement IDs must be added into the test scripts manually. If the test procedures are derived from the mode, the generator will extract the IDs and write them into the test scripts. In both cases, a tracking from test step to low-level requirement is possible and the matrix can be created automatically.

In the next step, the test results can be pushed into the model in order to create the documentation for this test campaign. The test verdicts are parsed and matched to the low-level requirement IDs. Furthermore, the model is updated and each element in the model will gain a flag if the requirement was checked successfully by the tests. An automated checker can parse the whole model and verifies that all requirements are marked as tested successfully. If a mismatch is detected, a list is created which contains all requirement IDs and model elements which are not covered by the tests.

#### 4.3.3.2 Documentation

In the last step, the documentation is created. The model and the test verdict are used and the results are published. Also, this document is created automatically, as all necessary parts exist in electronic form which can be processed by the help of tools.

These parts are distributed over two main sources which need to be compiled into the necessary documents. First of all, as the requirements are stored in the model, they need to be extracted and transferred into a requirements document. The traceability of the high-level requirements to the low-level requirements and vice versa is also calculated from the model.

The test verdict is stored in the test result documents which are created by the test tools. The requirements coverage can be calculated by comparing the test results with the model, as the requirement IDs are stored in the test result documents. This means that the traceability of requirements to the appropriate test procedures and test cases can be established and proved.

Regarding the change management, the evolution of the model (because of an evolution of requirements) must be documented as well. There are two possible

methods to capture all changes: either the addition, deletion or change of objects are stored in the model, for instance with a “not applicable” flag in case of a deletion or a “notes” property which reflects a change comment. The second method is to use a repository. It can be used to retrieve the changes. If the model was changed it must be checked into the repository. The check-in comments must then reflect the changes. In order to create the documentation, the changes can be retrieved either from the model or from the repository.

In the end, all necessary chains of information flow and the traceability can be assured:

- From high-level requirements to low-level requirements
- From high-level requirements to high-level test procedures
- From high-level test results to high-level requirements
- From low-level requirements to high-level requirements
- From low-level requirements to low-level test procedures
- From low-level test results to low-level requirements

#### 4.3.4 Requirements

A requirement is a tuple which consists of (at least) three items:

- Identification - Requirement ID
- Description - the requirement itself, a textual description
- Rationale - description why this requirement is necessary

In most cases, the requirements are not written by developers but by system designers. Also, the requirement document must contain all information which is necessary to describe the product. Furthermore, each requirement must be complete, accurate, and unambiguous (formal methods might be used to prove requirements [50]). A different approach is to use a so called “Controlled Natural Language” which consists of a well-defined sub-set of the natural language and “avoid[s] textual complexity and ambiguity” [11].

There might be requirements which cannot be implemented on certain levels. For instance, if there is a system specification and a requirement states that the system should be shielded against electromagnetic waves, this requirement cannot be implemented in code which will run on this system.

“Following the conventional development process, much irrelevant information — irrelevant from the point of view of system functionality — had to be kept in the design document to ensure that it be considered during coding. This extraneous information negatively affects productivity and quality.” [79, p.40]

However, the visual notation might not only focus on functional aspects but also non-functional details [19]. As this information can be stored using a second notation, it is not getting lost but also doesn't have a negative impact.

The more a system communicates with other systems, the more exception handling needs to be introduced into the system specification. In order to make the system robust, all exceptions need to be caught, for instance if a remote system does not answer anymore or sends invalid messages. Therefore, a system specification must contain two sections: the description of the normal behavior and a part where all the exceptions are described. It is a sophisticated task not to introduce contradictions between these requirements. For instance, if an error routine is executed, it might not switch back to the calling state of the normal system behavior.

If a model is used as the specification, the requirements stored within the model might differ from traditional textual ones. Regarding the transition from one state to other states, it might be easier and more clear to use a model because only the guard conditions and the actions need to be described. In case of a textual description, all possibilities must be written down in detail to avoid confusion. This is explicitly done in a model.

#### 4.3.4.1 Door-Status-Controller Example

In order to explain the issues with traditional textual description, a real-world example is given here. A door controller sends the current status of a door of an A/C to a display controller.

The door controller itself consists of two interfaces and a processing unit. One interface collects the data from the sensors of the door (discrete signals). The processing unit computes the door status and composes a message. The second interface transfers the message via a bus system in a fixed-preset interval (e.g. transmission every 25 ms).

The display controller receives the message and displays the current status. Like the door controller, the display controller consists of two interfaces and a processing unit. The bus interface receives the message which is processed and the indications are switched via the second interface.

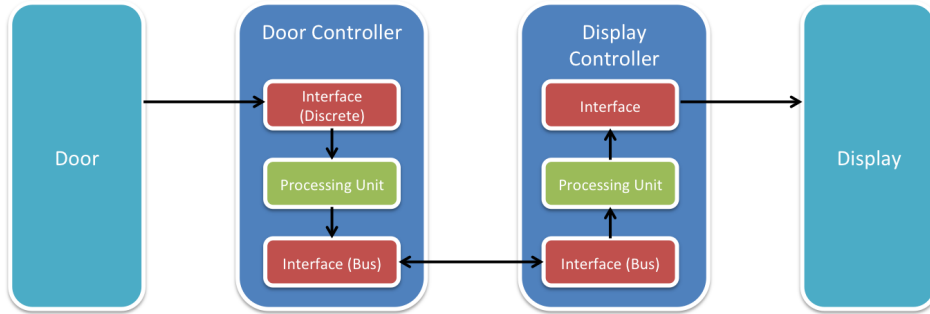


Figure 4.5: Example: Doors Controller

There are five different bits which are transferred from the door controller to the display controller which will set four indications. Three indications show if the door is open, closed or in-transition (currently opened or closed). A fourth indication displays a fault if an error occurred.

Three bits are used for the current door status. Each one describes if the door is open, closed or in-transition. Furthermore, a failure bit is sent by the door controller to indicate if a fault was detected (for instance, the door can't be moved).

It is common practice in the avionics domain to use redundant information. In this case, several bits are used to transfer one door status. It could have also been accomplished by using an enumeration type but then a fault cannot be detected. By using several bits for one status, a mismatch (e.g. door is open and closed at the same time according to the bits in the received message) can be detected easily.

Table 4.1: Door Status

Bit	Description
1	Open - the door is fully opened
2	Closed - the door is fully closed
3	In-Transition - the door is operated at the moment and therefore neither opened or closed
4	Fault - the door is inoperative or stuck and cannot be moved. It cannot be determined if the door is opened, closed or in-transition
5	Validity - the data is "valid and fresh"

A fifth bit is added by the bus interface of the door controller. This validity bit is set to ensure that the transferred data is "valid and fresh". This is necessary in order to inform all receiving controllers that the transferred data might not be valid as the status was not updated by the door controller for some time (in contrast to

non-aircraft systems, most — if not all — A/C systems are sending their status in fixed time frames so that no timeouts can occur).

There are four requirements which shall specify the corresponding behavior. As indicated by requirement IDs, they are located in different parts (general behavior, error management and data integrity) of the requirements document so that a comprehensive view is not given.

Table 4.2: Requirements Example

ID	Description	Rationale
100	Door is closed indication is set to on if closed bit is true else it is set to off.	Door status needs to be displayed.
101	Door is open indication is set to on if open bit is set to true, else it is set to off.	Door status needs to be displayed.
...	...	...
670	Fault Indication is set to on if fault is true or open and closed or open and in transition or closed and in transition is true. Else it is set to off.	Faults of the door needs to be displayed.
...	...	...
890	If the validity bit is set to true, the received signals are valid and should be regarded. If the bit is set to false, they should be considered as invalid and should not be regarded.	The faulty status is used as a redundant way to signal a fault in the door controller.

Some issues arise regarding these requirements. It is questionable if these behaviors are intended:

1. If the closed bit and the fault bit are set to *true*, the fault indication must be switched on (ID 670). According to requirement ID 101, the closed indication needs to be switched on as well — even though this would no make much sense as the actual door status cannot be retrieved.
2. According to requirement ID 670, if all three bits for closed, opened, and in-transition are set to *true* and the validity bit is also *true* (ID 890), the fault indication is set to off while the indications for the signals are set to on.
3. If the validity bit is then set to *false* (ID 890), all indications are set to off.

In order to specify these flaws, more requirements are necessary to fill the gap. However, this is a complex task as all combinations of requirements must be taken into account. Furthermore, in most cases the requirements are scattered in the document and some might not be regarded in order to fix the discrepancy.

Simple test cases would check if each requirement is fulfilled. This means that for requirement ID 100 the signal would toggle the values *true* and *false* and the indication would be evaluated. This would also be done for all other requirements. Eventually, all tests would evaluate to pass and the software is regarded as tested or “bug-free”.

As shown by this small-sized example, some gaps are not discovered and therefore not tested. The later such flaws are discovered, the more expensive it gets to fix these omissions (see chapter 3.7.1 *Detection of Faults*).

---

---

## CHAPTER 5

---

# E-Cab Project

E-Cab was a project dedicated to “E-enabled Cabin and Associated Logistics for Improved Passenger Services and Operational Efficiency”. This project shall provide airports and airlines with an environment that enables the potential offering of a step change in service concept. The project consists of over 30 partners from 13 nations. All components are developed and tested. The testing phase was finished by the end of June 2009.

The overall outcome of this project was to create a seamless flow for all passenger activities from booking a flight up to leaving the airport after landing. Due to handling all the different goals of this project, it was divided into six main sub-projects which are shortly introduced here.

### 5.1 Sub-Projects

One of the sub-projects took care of the passenger services on board, including the integration and usage of the personal digital devices of the passengers into the internal cabin systems, like the In-Flight Entertainment System (IFE). Information can be exchanged via the on-board network, like the preferred time slots for meals. The workflow could then be optimized to the passengers’ needs and the crew would have less pressure.

Another sub-project was dedicated to people moving at the airport. If the passenger enables this service, the system can track the position and offers guidance, for instance if the airport is unknown. This helps a passenger to find his way out of the airport, to the next taxi station, an exchange bank, the next hotel, a meeting place or the way to the gate. Furthermore, it can remind the passenger to get to his flight in time. For example, if a passenger has activated this function, he will receive a short message on his mobile phone that he has to leave the duty free shop and proceed to the gate. If the passenger is about to run late, the system will route him via the shortest and quickest path through the airport area.

Cargo handling was the topic of a further sub-project. Each piece of luggage is provided with an Radio-Frequency Identification (RFID) tag in order to locate it quickly at the airport or within the aircraft. RFID Readers are located at the Unified

Loading Device (ULD). Each ULD can then report a list of baggage to the on-board database. If a passenger does not show up, his luggage needs to be removed from the plane. By checking the on-board database, the container can be found easily and the luggage can be removed quickly. The delay is kept at a minimum, which can save money for the airline. These services — which are offered to the passengers — are created and maintained by several systems located on-ground and on-board. Long service chains are needed to pass the right information to all related systems.

Apart from selecting the food while booking the flight, the food could be ordered via the IFE and also the time slot in which the meal will be served could be defined. Every request from passengers is transferred directly to a mobile device, which belongs to the standard equipment of every member of the crew.

Trolleys are used to bring the already prepared meals to the plane. These are then stored in the galley. These trolleys are equipped with RFID readers. Each meal is tagged with an RFID chip. The trolley can report to the on-board database which meals are stored in which location. With the seat information and the preselected mealtime slot, the crew is informed to heat up the food and serve it to the right passenger just in time. This was part of the sub-project “E-Galley”.

The topic of the connecting networks was part of the fifth sub-project. Here, the network specifics were defined. This includes all on-board and airport networks and technologies (like Third Generation Cellular Mobile Network (3G), Worldwide Interoperability for Microwave Access (WIMAX), Bluetooth, Virtual Private Networks (VPNs) and so on). Also the integration of the different databases into the network was an important part.

The last sub-project was responsible for the integration of all other sub-projects, ensuring the interoperability and testing of all new developed and preexisting systems. The validation and verification activities were accomplished within this part of the project.

All of these sub-projects work closely together in order to provide a seamless flow of information. The passenger is supported by guiding him through the airport and he’s provided with more convenience functions in the aircraft. Furthermore, the airlines can plan more aspects of the flights in advance. For instance, the more passengers are booking their meals in advance, the less additional meals they have to provide in the aircraft.

## 5.2 Location

The project demonstrator was located at two different Airbus sites. The cabin and passenger functions demonstrator was installed in Hamburg. The demonstrator



for the freight handling was located in Bremen. Both sites were connected by an inter-site link so that the concerned systems could communicate within a logic network without noticing the distance between both sites.

The main test site was located at Airbus in Hamburg. There, a several hundreds square meter area was reserved for this project. A cabin mock-up of an original A300 with four seat rows in standard lining was located there (see figure 5.1 *A300 Mock-Up*). Each sub-project installed their own equipment here.

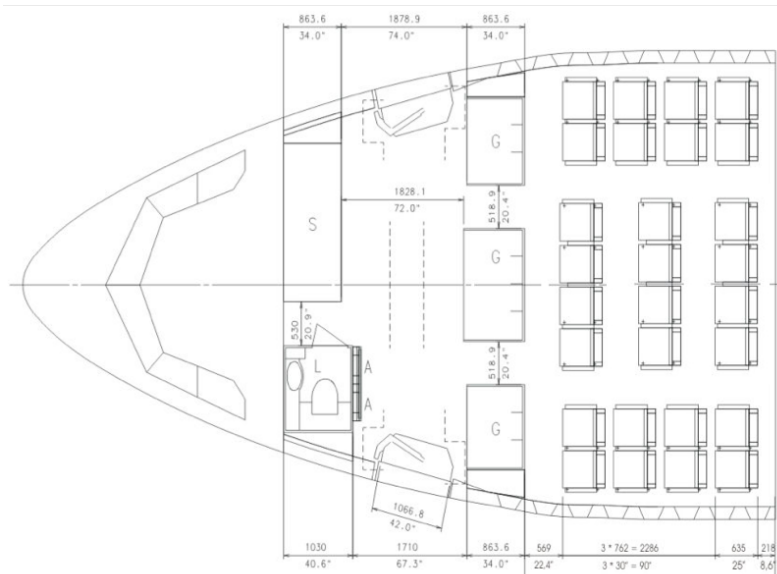


Figure 5.1: A300 Mock-Up

The area was divided into three different sub-areas in order to simulate three different airports (see chapter 5.3.1 *Test Scenario*, see figure 5.2 *Floor Plan*). In order to distinguish between these areas, they were taped and marked in different colors.

The cargo mock-up was located at the second site at Airbus in Bremen. The cargo compartment of a A340 was equipped with RFID reader. Furthermore, a high-loader which lifted the RFID equipped ULD into the aircraft was installed (see figure 5.3 *Cargo Mock-Up*).

### 5.3 Testing Activities

Many systems were under development during the integration phase. The availability and full functionality of all systems could therefore not be taken for granted. This means that there was a variety of possible combinations of original equipment, development equipment and simulated equipment.

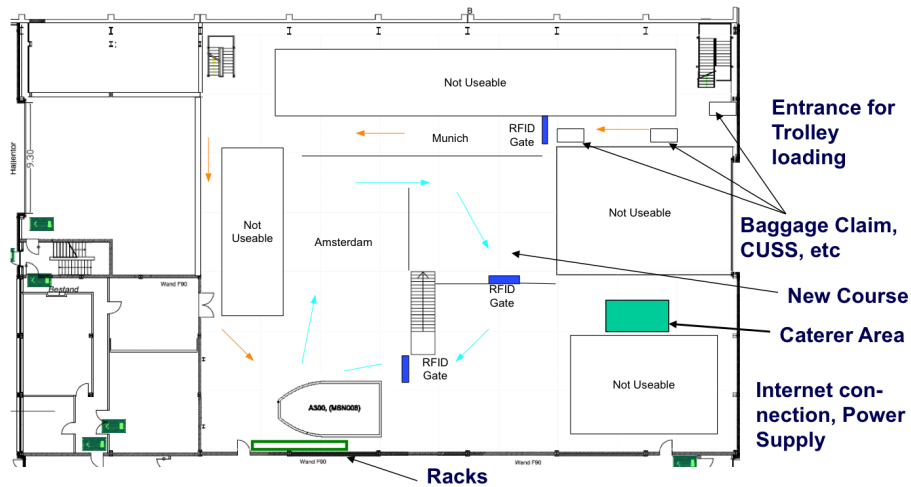


Figure 5.2: Floor Plan

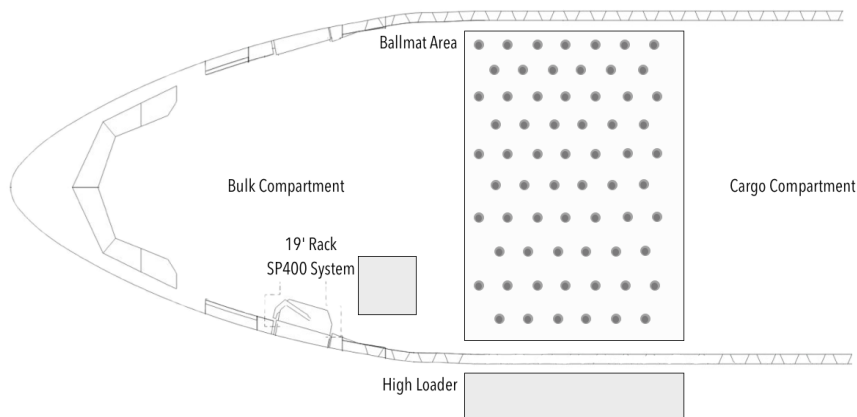


Figure 5.3: Cargo Mock-Up

Lots of information must be exchanged between the various E-Cab systems. To ensure privacy and to protect personal data, each system is only provided with data on a “need to know” basis. No system should be able to create a profile by collecting various data. Therefore, only temporary IDs which won’t provide any conclusions about a single passenger are used.

All mentioned functions and systems are connected through a variety of networks (also connecting different testing sites), links and associated protocols. Regarding the testing activities, the interoperability test bench needed to be connected to as many systems as possible in order to record and stimulate network traffic within different kinds of networks. Of course, checking of recorded values against test oracles can only be performed if all needed information could be gathered.

### 5.3.1 Test Scenario

In order to test and simulate all systems, a test scenario was created [28] [29]. In this scenario, all functions from all attached systems are involved.

Within the scenario, a common journey of a team and their freight shipping is depicted. They use the services offered by the E-Cab project, both on ground and during the defined flights.

The story starts with a team on a trip to a trade fair at Singapore. They travel from Munich via Schiphol to Singapore airport. They use different methods to gain their flight ticket (check-in desks and kiosk) and some book their meals in advance. Also, some of the team use the guiding system at the airport Schiphol. Their luggage is tracked by RFID so that the baggage system is able to determine the actual position (at the airport or in the aircraft). At Schiphol, one of the team members is called back to Munich. His luggage is located in the aircraft and the ULD which contains his baggage is unloaded, the baggage is removed and the ULD is loaded into the aircraft again. While being in the air, certain cabin functions are used. For instance, passengers change seats, meal slots are revised and orders of the duty free shop are placed which are directly passed to his account.

The exhibition demonstrator for this trade fair is produced by a supplier next to Frankfurt. It shall be shipped directly from Frankfurt to Singapore fair ground. Parallel to the flight of the team, the contracted freight forwarder sends the exhibition demonstrator so that it arrives just in time.

### 5.3.2 Test Bench

The test bench itself is shown in figure 5.4 *Test Bench Architecture*. The interfaces are located on the right side of the figure. On the left side, the five cluster nodes are located. They are connected via a real-time protocol with each other. Also, the cPCI-Module (compact PCI) is connected with the cluster nodes via this link. Each module can be accessed via the built-in console. It is also possible to connect laptops or PCs via the standard ethernet switch. They are used as additional consoles for managing the tests.

Various interfaces are used within E-Cab, like WIMAX, 3G, Ethernet, WiFi, discrettes Lines, RFID, Bluetooth and SATCOM. Two programmable switches are needed to connect the test bench to the on-ground and the on-board network. A third switch is needed for intercommunication of the cluster nodes.

The test bench has been chosen to be powerful enough to stimulate and record all kinds of signal on various interfaces.

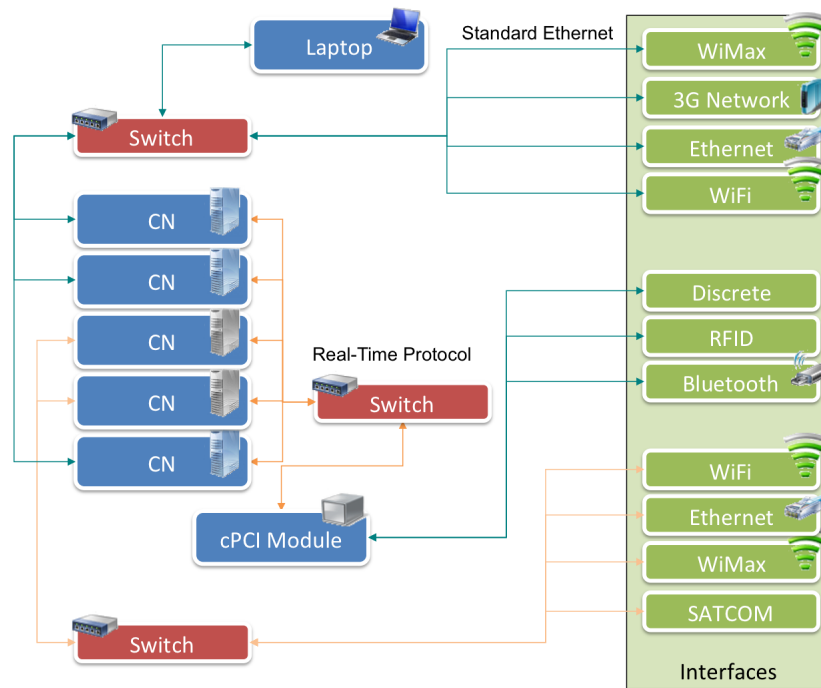


Figure 5.4: Test Bench Architecture

### 5.3.3 Testing

Special effort was put on creating End-To-End Chains within the mentioned scenario. From the testing perspective, this means that data-flows are stimulated at various input systems and must be recorded at various output systems (see chapter 7 *Testing Scenario*).

A new approach is needed to test these service chains. Not only one system is systematically tested alone, but many interacting systems. Checkers, oracles and stimulants are needed at several places, at different airports and on different aircrafts.

This will require different systems and logical entities of the simulated world to share the same physical location. This will impact the network architecture because it should be possible to simulate all the different locations using the same equipments (or, at least, a high percentage of them).

#### 5.3.3.1 Objectives

For the testing activities, several objectives have been defined [30]. The tests and the corresponding test steps shall be derived from a predefined model and with defined

semantics. Also, the test data shall be derived from the model. The model itself shall be expressed using a high level language. Quick changes shall be possible in case of design changes of the System Under Test (SUT) and/or environment. The model shall check for inconsistencies by using automatically performed checks, metrics and audits before test cases are generated.

One important point is that this high level language can be understood by all stakeholders, so that a common understanding about all systems of all involved parties is established in a very short time.

If original systems are not available, simulations should take over the functionality. These simulations should also be generated from the model. The creation and applying of these simulations into the network should be able to be created on the fly and fully automatically.

In order to check if all requirements are met, the requirements must be tracked in the model and during the execution. The needed reports and logs shall be generated automatically. Also, this coverage data should be redacted so that it can be stored directly in the requirements database.

### 5.3.3.2 Model based Solution

In order to gain the same understanding between all stakeholders, effort needs to be put on the visual notation (see chapter 3.5.2 *Visual Notation*). The objects and their relationships in the model should look similar to the used equipment (see chapter 6 *E-Cab Domain Specific Language*). Each stakeholder will then recognize their own parts and can contribute information more specific to the current topic.

“The used DSL, which was developed for the E-Cab project, was designed to look similar to the used components and systems. The structure and wiring of the systems are done intuitive.” [30, p.13]

Also, all input variables (for instance, a passenger name or RFID tag) is stored at a minimum count of places in the model. In order to discuss a certain test aspect with the stakeholders, all variables are visible without switching to (lots of) different graphs in the model. This eases the understanding of the test scope.

“Not all the data is used for every test case but all data shall be kept in one place. Defining the values in different graphs lead to unclear and confusing test data.” [30, p.16]

Each test step is marked with a requirements tag. If such a tag is successfully reached within a test, it is marked as covered. This list is later on exported in such a format

that it can be imported in the used tracking tool (see chapters 3.12 *Versioning*, 4.3.2 *Documents and Requirements* and 9 *Creation of the Target*).

All systems are modeled and present in the test scenario model. Each model of these systems can be used as an oracle in order to check the recorded data. Also, the model object can be used to generate a simulation on the fly. As the configuration of the test bench is also generated from the model, this created instance is integrated into the scenario automatically.

## 5.4 E-Cab Systems

Each of the E-Cab systems are provided by different companies and are interconnected at the Airbus site in Hamburg. The functional description of these systems is given in this chapter.

### 5.4.1 Check-In System

The check-in system is the start of most activities as the collected data is transferred into the other connected systems. The passenger will interact with check-in terminals which are the front-end for the check-in system. Also, the passenger can use a web-browser to interact with the system. The web-browser front-end can be used for check-in from home and also to switch settings while being at the airport.

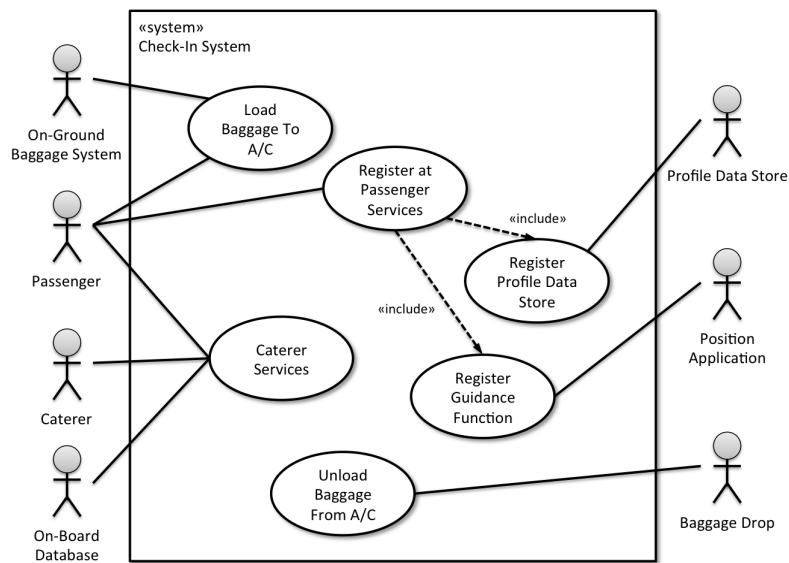


Figure 5.5: Use Case - Check-In System

The check-in systems utilizes the provided passenger data (like contact data and privacy settings) and registers this passenger at the data profile store and possibly at the position application (depending on the privacy preferences). All information of the passenger is stored at the profile data store. All other systems receive their needed information from this application on a need-to-know basis. The guidance function receives anonymized contact information so that it is not possible to identify a passenger, collect data and create profiles. This applies for all other systems which rely on passenger data.

In case of loading to or unloading baggage from the aircraft, the needed information is processed by the check-in system, for instance, the RFID equipped luggage (see chapter 5.4.4 *Baggage Systems*).

As the passenger defines his meals and time slots for the flight by using the check-in terminals or a web-browser, this information is stored together with his other data. The check-in system will provide this information to the caterer (see chapter 5.4.7 *Caterer*) and the on-board database (see chapter 5.4.6 *On-Board Database*).

## 5.4.2 Profile Data Store

The profile data store is a database which stores and guards all passenger information. All connected systems will receive only the necessary information to perform their tasks.

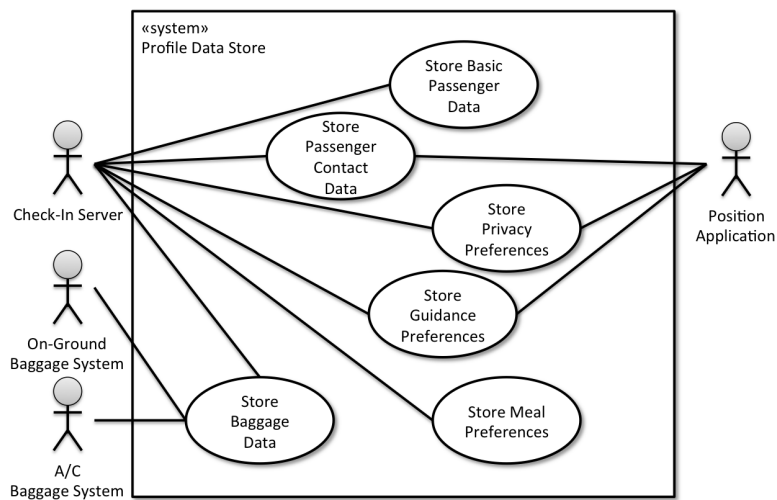


Figure 5.6: Use Case - Profile Data Store

Data stored in the database can be divided into three groups: static data, temporary data and process data.

Contact information (telephone number and address) or basic data (birthday and flyer status) does not change during a flight. The contact data is used by the position application which needs a mobile phone number for sending notifications.

Meal selections and guidance preferences are temporary data which might be changed before the flight.

Data which is only used for processing a task is subsumed as process data. This applies for IDs which are used only once internally. For instance, the assigned RFID for the baggage is used for storing the baggage in the aircraft.

All connected systems receive data from this database. The database takes care that only the necessary information is provided. For this task, disposable IDs are used so that a passenger cannot be tracked back.

The temporary and process data is deleted some time after it is used. The static data is also removed from the database after a specified amount of time (this does not reflect the data which is stored at internal airline systems).

### 5.4.3 Notification System

The notification system sends messages to a passenger if he opted-in for this service. If a passenger needs to be informed, he will receive a message via Short Message Service (SMS) or other push service to his mobile device.

The notification system will translate the content to the preferred language of the passenger (if possible). This information is stored in the profile data store and can be updated by the passenger.

If a passenger cannot catch his flight anymore and his luggage is unloaded from the aircraft, he will also receive a message where he can fetch his belongings.

The passenger can opt-in or out of this service anytime by visiting a web site and alter this privacy settings. Also, he can specify when and how he is informed, for instance only if he runs late or even if he might get late.

The notification system does not need to know where the passenger is or any other information. The system receives an ID from the position application. The contact data, like phone number and name, and preferred language is received from the profile data store. The notification is sent to the passenger. No further information is stored at the notification system — all temporary data is deleted right afterwards.



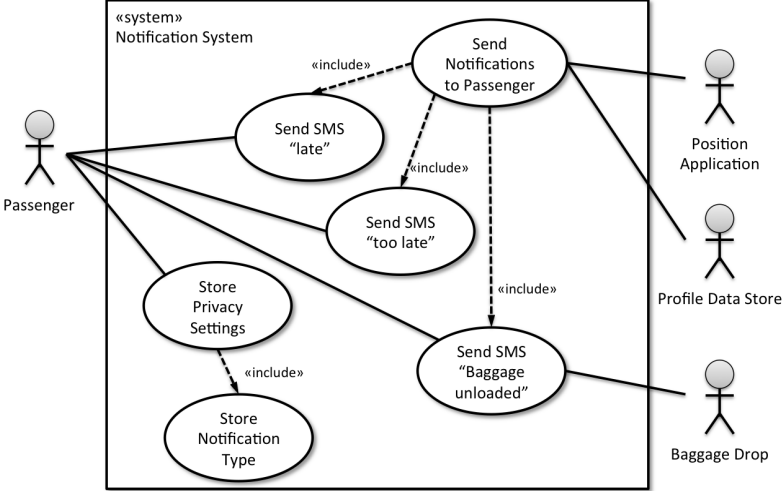


Figure 5.7: Use Case - Notification System

5.4.4 Baggage Systems

The baggage of the passengers is weighted and equipped with RFID tags. It is then stored in containers (so called ULDs). These ULDs are then tucked in the aircraft. After landing, the ULDs are unloaded from the aircraft and the luggage is transported to the baggage drop where the passengers collect their belongings.

In general, two kinds of baggage systems are responsible for this task. Firstly, there is a system responsible for the luggage on-ground at the airport. It starts with the acceptance of the baggage and ends with the storing of the ULD in the aircraft.

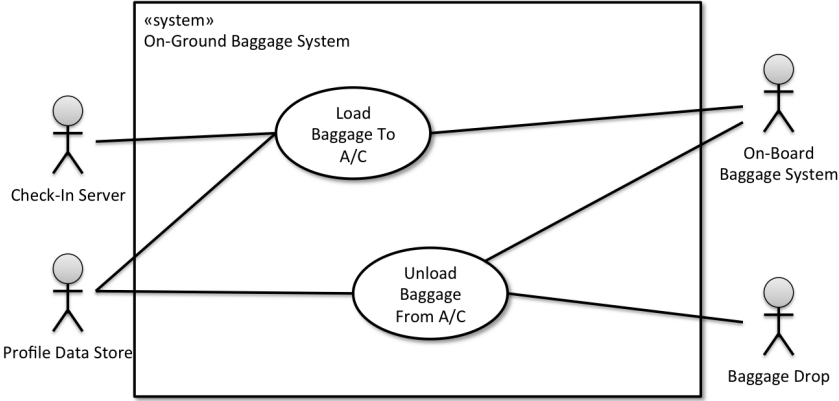


Figure 5.8: Use Case - On-Ground Baggage System

The second baggage system is installed in the aircraft. This system takes care of storing the containers in the cargo hold of the aircraft.

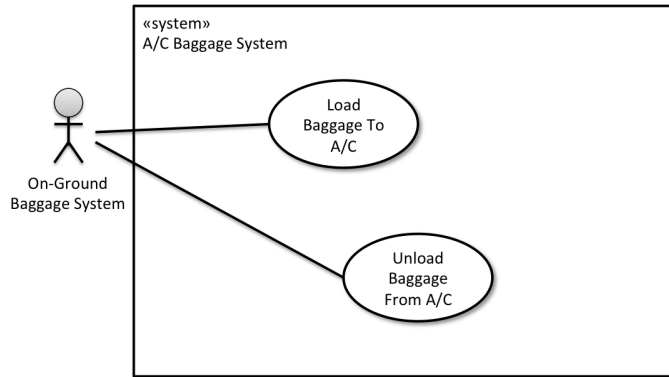


Figure 5.9: Use Case - A/C Baggage System

Each ULD is equipped with an RFID reader. As each piece of luggage is tagged with an RFID chip, the reader detects each add and removal of baggage. The ULD does not need to know which item belongs to which passenger — it only stores the tags.

At the cargo doors of the aircraft, RFID readers are installed. If a ULD is loaded or unloaded, it is registered at the aircraft as each ULD is also equipped with an RFID tag. The content of the containers are not stored in the on-board database. Instead, the position of the container is resided.

If an item needs to be removed, the containers can answer the query if a certain luggage is stored inside. The knowledge of the position of the ULD in the aircraft can then help to unload the container as quickly as possible.

Instead of looking for a specific item in the complete cargo hold, it is therefore a simpler task just to look inside one of the containers.

As only the baggage of effective passengers is allowed on-board, the luggage of each “no-show” passenger must be removed from the aircraft before take-off. If this task can be reduced to a minimum of time (in best cases even before the gate closes), the airline can save time and money. Also, the passengers on-board do not need to wait for this procedure to be completed as this can be performed during the boarding.

#### 5.4.5 Position Application

If the passenger opted-in for the notification service, the position application will track the passenger at the airport. The position application serves this information anonymously to other services. The application provides an interface where this

function can be switched on or off by the passenger. Only if the passenger decides to use this function, the current position at the current airport will be tracked.

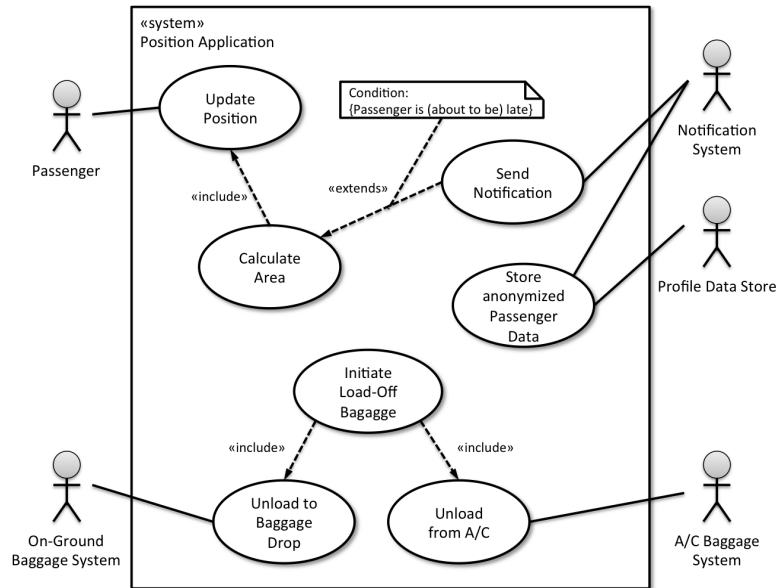


Figure 5.10: Use Case - Position Application

The position system is informed by the profile data store about a passenger if he opted-in while checking-in. Only two properties of the passenger are transferred from the profile data store: the passenger ID and the flight information.

The passenger ID is used as a temporary identifier for the passenger which cannot be traced back by any system (except the profile data store). The position application depends on the flight information for the passenger stored by the profile data store.

The position application will utilize the notification system to inform the passenger. In case he runs late, the position application will send a message type and the passenger ID to the notification system. Therefore the position application does not need any more details about the passenger (e.g. name or phone number).

The position application needs a floor plan of the airport. All possible gates, entrances and exits must be incorporated in this map. Furthermore, the map must be divided in partitions. Each partition reflects an area of the airport. As the application knows where the gate for a passenger is located and knows the actual position of this passenger, it can calculate how long the passenger needs until he reaches his destination at the airport. Furthermore, if many passengers need to get through a certain point and it might get crowded, the position application can incorporate this knowledge for the calculation.

The position application does not share the actual position of a passenger with any other systems. In fact, it reduces the resolution of the current position. This means that the exact position is mapped to an area (partition of the floor plan) of the airport.

The localization is done by RFID chips which are placed on the ticket (see figure 5.11 *Boarding Ticket* and figure 5.12 *RFID Chip implemented in the Boarding Ticket*). The calculation of the exact position is done by measuring the distances from the ticket to a couple of antennas.



Figure 5.11: Boarding Ticket

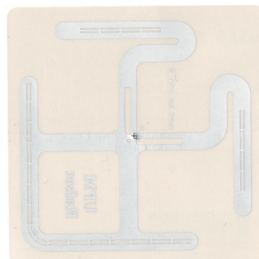


Figure 5.12: RFID Chip implemented in the Boarding Ticket

RFID readers are placed within the airport areas and also in the doors of the aircraft. If a passenger walks through such a door, he is considered to be in the aircraft and the temporary data is deleted.

If a passenger is too late to catch his flight, the position application can inform the baggage systems. The baggage of this passenger can be unloaded even before the gate closes. This procedure can save time (and therefore money) for the airlines.

### 5.4.6 On-Board Database

The on-board database stores the meals and time slots the passengers chose while booking the flight. Also, the database stores the actual transferred food which is carried into the aircraft by the caterer.

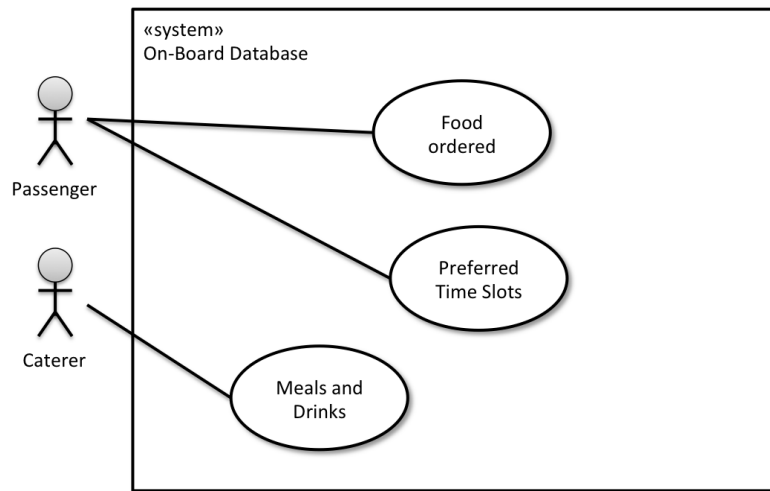


Figure 5.13: Use Case - On-Board Database

The data is transferred from two instances. Firstly, the passenger (via the check-in system) provides the selected food and time slots when he wants to consume his meal. Secondly, the caterer provides the food. It is stored in trolleys and delivered to the aircraft. Each trolley contains several meals which are organized by the time slot schedule (if the airline provides this information to the caterer). The on-board database is aware of the content of the trolleys. The actual status of meals and beverages is stored here and therefore, reordering of items during a stopover can be processed automatically.

### 5.4.7 Caterer

The caterer receives orders from the airlines. This reflects the orders from the passengers. Of course, the caterer does not need to know which passenger ordered which food. Also, the airline might order additional meals. For one thing, that a passenger switches his mind, for another thing, that some passengers might not have ordered food before they checked-in.

The caterer delivers the food in trolleys to the aircraft. As the airline has the information about the time slots, they can instruct the caterer to load the trolleys

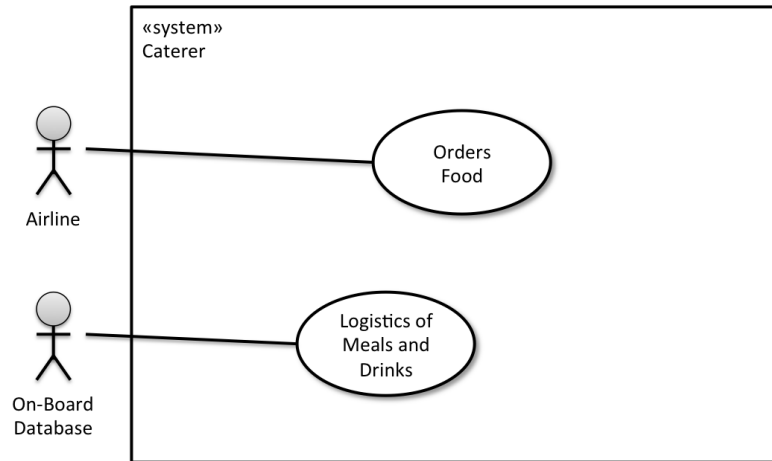


Figure 5.14: Use Case - Caterer

appropriately. For instance, all meals which are served at the same time can be stored in the same trolley. If one of these trolleys is emptied during the flight, they can be exchanged by a loaded one. This results in a minimum amount of opened trolleys and eases the handling in the aircraft galley.

---

---

## CHAPTER 6

---

# E-Cab Domain Specific Language

The E-Cab DSL with all its attributes and characteristics is explained in this chapter. The language was designed so that all involved stakeholders can identify their parts of the project. It was also designed with the identified key elements which are necessary for a visual language (see chapter 3.5.2 *Visual Notation*). For instance, effort was put on the easy readability of the symbols. By using a combination of visual variables like colors and forms, different objects are clearly distinguishable from each other. Also, by using same variables (e.g. the same or similar color) a grouping of symbols is intuitively done. Effort was also put in adding icons for most of the symbols as an additional variable.

For instance, each domain of the main graph is modeled as an island which fits to the work breakdown structure of the project (see chapter 6.2 *Main Scenario Graph*). Bright colors, icons which describe the content of such a domain and a textual description provide a cognitive visual effect: semiotic clarity, perceptual discriminability, semantic transparency, visual expressiveness, dual coding and cognitive fit.

The same applies for the airport locations (see chapter 6.7.5 *Airport Location Object*). Depending on the type of location, the color, icon and description of the graphical representation changes.

Effort was also put into the idea of re-using already known graphical symbols, bridging to the knowledge of the stakeholder. For instance, the “pin” which is used to display the location of a passenger on an airport layout uses a similar representation like it is used in navigation systems or maps (see chapter 6.7.7 *Passenger Move Object*). This symbol also uses common colors to display further information: everything is okay if the color green is displayed. In contrast, if the passenger is too late, the color switches to red.

In order to prevent illegal usage of symbols and their relationships, templates, regular expressions, and predefined lists have been used. For instance, the transitions in the behavior graphs only show the name of the incoming message once at the beginning. As the type of message does not change during this path, displaying the same information over and over again is omitted. Also, as the modeler chooses from a predefined list, no wrong input can be given which leads to a well-formed model which is understandable by all stakeholders.

Different colors, types and thickness of lines and arrows are used for different types of transition between objects. This should make it simpler to distinguish the meaning of context between symbols.

The same approach is followed for most other symbols and their relations.

## 6.1 Tool Support

The DSL modeling tool MetaEdit+ [49] supports tracking of requirements in a way that for every single item (object, relation, property or complete graphs) within the model, a reference ID can be inserted. These IDs can then be further processed and used in Excel or Doors. Testing the coverage of all requirement IDs can be done automatically with running a check after the model is designed. Also, the IDs of test cases and test steps can be recorded in the Modeler. These IDs will be used in the generated reports of the test runs.

Consistency checks of the model are done automatically within the tool. Rules within the meta-model ensure that only valid objects are allowed to be used in a graph (see chapter 6.12.4 *Check Model*). Furthermore, restrictions of relations (via the port concept) can be defined. The result is a well-formed graph.

Another important task is the support for transforming the model into other models, textual descriptions or languages. MetaEdit+ has a built-in language to access all elements (see chapter 6.12.1 *MERL* and chapter 6.12.3 *GUI Enhancements*). This is used to extract the needed information from the model as input for the tool chain.

MetaEdit+ has a built-in SOAP server. All items can be accessed through this web service. For example, this means that items can be highlighted if they are executed or triggered on the Test Bench. If the behavior of a system is modeled as a state-chart, the visited states and transitions can be highlighted, marked with a different color or annotated with comments.

## 6.2 Main Scenario Graph

The main scenario graph is the start for each test campaign. All necessary equipment is set up and their relationships are defined here.

The main graph is the starting point for all other activities. It contains the objects for the several domains which are set into context. Each object which is placed on the main scenario graph is a place holder for further sub-graphs. The specific internals of a domain are handled in these associated domain graphs. For instance,



meals are defined in a caterer domain and they need to be connected passengers in a terminal domain in order to assign menus to passengers.

By subdividing the internals into the several domain graphs, the main scenario graph is not cluttered with details but provides a test campaign like a story board.

The main scenario graph will contain the objects for the following domains

- Terminal Domain
- Baggage Domain
- Aircraft Domain
- Caterer Domain
- Airport Domain

All these objects are designed as “islands” to clarify their area of competence.

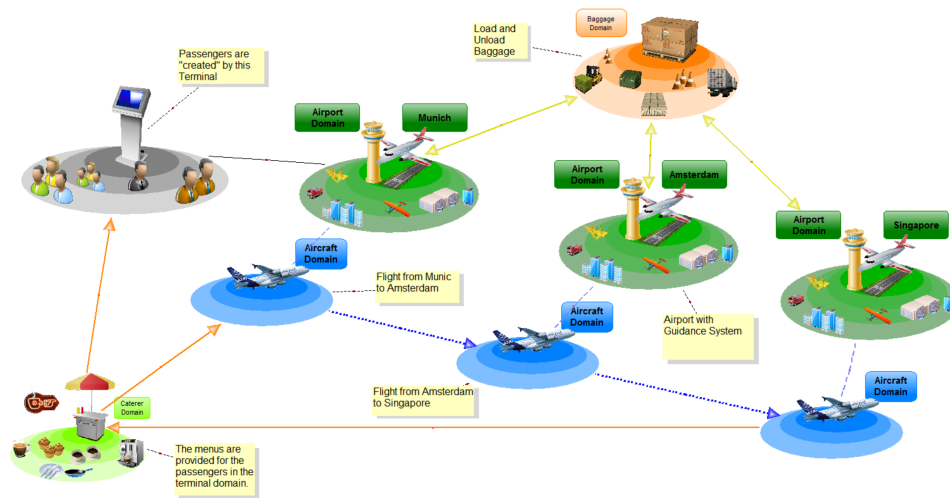


Figure 6.1: Scenario

An example of such a story board setting is shown in figure 6.1 *Scenario*.

The scenario graph has got 21 properties. Firstly, a name is assigned in order to distinguish it from other scenario graphs. Also, a date is attached which is used for documentation reasons.

Several properties are necessary for the underlying test bench components. For instance: the type of the test, an ID, whether the test shall stop in case of a failure, a verbose flag in order to generate more logging data and a switch which defines the

communication type of the underlying test programs. Also, IP addresses and ports are needed for communicating with the test bench and the MetaEdit+ proxy (see chapter 7.5 *MetaEdit Bridge*). The access data for logging into the test bench, the local and remote paths are also present here so that everything related to the test execution can be configured in one place.

The screenshot shows a dialog box titled "Ecab-Scenario: Graph" with the following fields and values:

- Name: E-Cab Scenario
- Date: 01.05.2014
- TIL: HSI
- SUT ID: System Test
- Stop On Fail: YES
- Verbose Flag:
- Verbose Flag More Details:
- Time Switch:
  - Local time on Test Bench
  - Specify time within model
  - No Time AFAP
- ScheduledDate Minute: 40
- ScheduledDate Hour: 11
- MetaEdit Proxy IP: 192.168.1.200
- MetaEdit Proxy Port: 7896
- Webservice Address: 192.168.1.43
- Webservice Port: 8080
- TB Network Address: 192.168.1.43
- TB Network Port: 22
- TB User: hartmann
- TB Password: ecabdis
- TB RTT Testcontext: /home/hartmann/ecab/
- TB Remote Path: /home/hartmann/bin/
- Local Temp Directory: c:\temp\

Buttons at the bottom: OK, Cancel, Info...

Figure 6.2: Scenario Graph Properties

Two additional properties store the IP address and port for the web server. If the notification system is simulated by the test bench, these values are used as the configuration for the landing page of the service (see chapter 5.4.3 *Notification System*).

Another three properties are essential for the model and its execution. Time and date for flights are defined within the model. In order to trigger actions depending on these points in time, the time of the test environment needs to be set. For instance, if a flight is scheduled in a week and the test is executed right now, it should not be in an idle state for a week. Therefore, three different execution models are implemented. Either the local time of the test bench is used (Local Time on Test Bench), the date and time can be defined within the model (Specify time within model) or the timing is adjusted in such a way that no idle time can occur (No Time - AFAP (As fast as possible)).

If the time is specified within the model, the two properties “Scheduled Day (Minute)” and “Scheduled Day (Hour)” are incorporated as local time. In case of a different time model, these values are ignored.

### 6.2.1 Terminal Domain Object

The terminal object contains everything which is related to passengers, their flights and their preferences for meals. Everything can be defined and assigned within the nested terminal graph (see chapter 6.3 *Terminal Domain Graph*). All details are set up in that graph.

The name of the terminal object is the only property. It is used to distinguish between different terminal objects (if present).

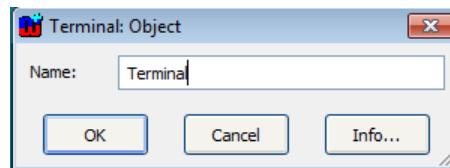


Figure 6.3: Terminal Domain Properties

There are two connections to other objects. First, at least one airport needs to be connected. Only existing flights can be assigned to passengers. The flights are evaluated by the aircraft routes which are connected to the origin airport. All possible routes from this airport can be assigned to a passenger. The second connection leads to the caterer domain. All menus are derived from this domain and can be assigned to passengers.



Figure 6.4: Terminal Domain Object

The terminal object is represented by a grayish island. A terminal computer is displayed in the middle of the object. It is surrounded by passenger icons.

### 6.2.2 Caterer Domain Object

Everything related to meals and drinks is part of the caterer domain. This object is used to assign meals, drinks and menus to existing flights. The caterer object is a

placeholder for the caterer graph which contains all the details. This information is later used in the terminal domain in order to assign meals to passengers.

The information for the caterer flight objects is derived from the connected aircraft domains (the attached objects are traversed and the flight information, e.g. connected airports are collected). All connected flights can be used for the caterer flight object within the caterer domain graph (see chapter 6.4.4 *Caterer Flight Object*).

The menus are presented to all passengers in the connected terminal domain. This is the input for the meal database objects (see chapter 6.3.2 *Meal Database Object*) in the terminal graph (see chapter 6.3 *Terminal Domain Graph*).



Figure 6.5: Caterer Domain Object

The object itself is assigned to the terminal domain and to the aircraft domain. This is done by the same type of transition.

The caterer domain object is represented by a light green colored island. Several typical icons (like coffee or a plate) related to food or drinks are placed on it.

### 6.2.3 Airport Domain Object

The airport domain object contains all systems and functions related to airports. These functions are subsumed in a system deployment and configuration graph (for instance, the position application (see chapter 5.4.5 *Position Application*) or the notification system (see chapter 5.4.3 *Notification System*)). The settings and the behavior of these systems are specified here. Furthermore, the layout of the airport is presented. This map is the basis for several services.

The airport object is a placeholder for the following specific graphs:

- Airport Layout

- Airport Area
- System Deployment and Configuration

The details for these graphs can be found in the chapters 6.6 *Airport Domain Graph*, 6.7 *Airport Layout Graph*, 6.8 *System Deployment and Configuration Graph* and 6.9 *Behavior Graph*.

The airport has got two properties. First, an ID needs to be assigned. This is the known abbreviation of an airport. Furthermore, the name of the airport is specified.

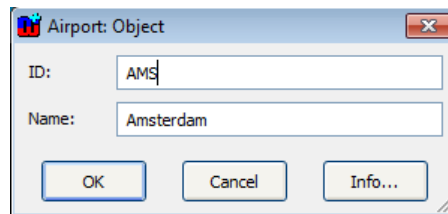


Figure 6.6: Airport Domain Properties

The airport has two different connections to other objects. In order to provide flights, at least one aircraft domain needs to be linked to an airport object. These flights can be assigned to passengers, if a connection to the terminal domain is drawn. If such a link is present, all connected flights can be chosen by passengers defined in the terminal graph.



Figure 6.7: Airport Domain Object

The airport domain object is represented by a green colored island. Several icons related to airports and their environments are shown. The name of the airport is displayed in a box in the upper right corner.

### 6.2.4 Baggage Domain Object

The baggage domain contains everything which is related to the baggage transport to and off the aircraft. The details are defined in the nested baggage domain graph (see chapter 6.5 *Baggage Domain Graph*).

The object needs links to all airports which contain baggage systems. No other connections are needed.



Figure 6.8: Baggage Domain Object

The baggage domain object is represented by an orange island. Icons related to baggage and containers are placed on the object.

### 6.2.5 Aircraft Domain Object

The aircraft domain object symbolizes the start and/or landing of an aircraft. It is used to create flight connection between airports. No further subgraphs are present for this object.

There are two types of connections for an aircraft domain object. A flight is created by placing two aircraft objects in the main scenario graph. Both aircrafts need to be linked to two different airports (one each) as an origin and destination airport is needed for each flight. In the last step, both aircraft objects need to be connected to each other.

The object is represented by a blue island. An icon depicting an aircraft is displayed in the middle.



Figure 6.9: Aircraft Domain Object

## 6.2.6 Scenario Connection Types

There are five different connection types in this graph. First, the terminal domain is connected to the airports in order to provide flights for the passengers. Second, the airport itself must be connected to at least one aircraft. Also, a link to the baggage domain is necessary. The aircrafts are connected among each other in order to create flights. The baggage domain uses one type of connection for links to the terminal domain and to the aircraft domain.

## 6.3 Terminal Domain Graph

The terminal graph is the place where all passenger objects are created. Also, the passenger objects are connected to the meal database and the flight database.

The created passenger objects (see chapter 6.3.1 *Passenger Object*) need to be assigned to a specific flight (see chapter 6.3.3 *Flight Database Object*). This is done by creating links between the objects using the defined ports.

The same approach is used to assign a meal (see chapter 6.3.2 *Meal Database Object*) to one or more passengers. By using the ports for meals, a connection is applied between these two objects.

The graph is located on the main scenario graph and is represented by the terminal object (see chapter 6.2.1 *Terminal Domain Object*).

It contains three different objects: the passenger object, the flight database and the meal database.

The graph has got two properties. Firstly, a name is assigned to distinguish it from other graphs. Furthermore, a flag can be set in order to enable or prevent live checking of the graph (this is used by the test bench object (see chapter 6.11.2 *Test Bench Configuration Status Object*)).

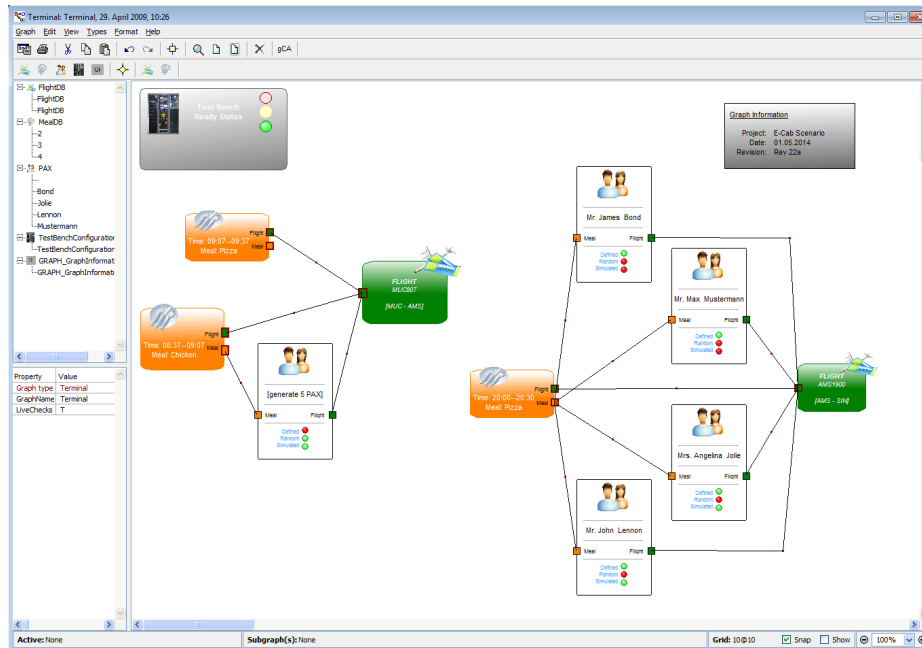


Figure 6.10: Terminal Domain Graph: Passenger Objects, Database Meal and Database Flight

### 6.3.1 Passenger Object

The purpose of the passenger object is to store all information for a passenger in one place. This data is collected for all kinds of systems and services which can be used by the passenger and the airlines. Of course, privacy is an important topic. Therefore, all personal information about a passenger is treated with data privacy in mind. Partly, these properties might be valid for longer times, e.g. name and address are stored as long as the customer permits this. Other properties are only valid for a short period of time. This includes single-use ID, e.g. tags which are used to identify the baggage of a passenger. In the real systems, this information is segregated in such a way that it is not possible to connect the information to gain detailed user profiles. All the properties are only stored in one place in the model to ease the work of the tester and to fulfill the paradigm of storing information in one place instead of scattered within the model.

Several properties are needed to specify a passenger. First of all, an ID is attached to the passenger. This is needed to identify this passenger later on. Furthermore, the passenger needs a name, a gender and a nationality.

A variety of contact information, e.g. mobile phone number, landline number, or email address, can be inserted for each passenger. The airline can choose the right



medium for publishing the requested information by the passenger. For instance, information about the current flyer program can be sent out by email while the information of a gate change needs to be pushed out quickly and can be sent by SMS or other push service.

The number for a mobile phone and a flag whether the passenger needs guidance at the airport is used in order to provide location information for the visited airports. If the flag is checked, the passenger will receive short messages on the defined mobile, e.g. information about gate changes, the quickest way through the airport perimeter to the gate.

The RFID as well as a 2D barcode is stored with the user profile. These IDs will be used for tracking the passengers as well as their baggage. For instance, the RFID chips are implemented in the boarding ticket (see figure 5.11 *Boarding Ticket* and figure 5.12 *RFID Chip implemented in the Boarding Ticket*).

A preferred language can be assigned. This can be used by the crew to address the passenger in the favorite language if possible. Also, the airline can contact their customers in a more personal way — in contrast to send out a short message either only in one language (English in most cases) or a larger message which contains several languages and the customer needs to find the appropriate section.

By two other properties, the meal and the meal time slot can be defined. The passengers can do this while booking the flight and selecting the meal. Of course, this information can be added later on by the web application from the airline.

The complete list with all properties is shown in figure 6.11 *Passenger Properties*.

In order to switch the status of the guidance system, login and password field is present. By using this access data, the passenger can log into the website and alter or confirm guidance help on the airport.

In the lower right corner of the property window, four special properties are located. These can be set in order to set up parts of the test environment. If simulated passenger objects are needed, the property “Simulated?” needs to be set. Two types of simulated passengers can be created by the test system later on: defined passengers and randomly created passengers.

Therefore, a boolean flag called “Random” is defined. This flag is set or unset by the tester in the storyboard while defining the scenario. If the flag is set to *true*, all properties are created by random choice. Furthermore, the tester can specify how many passengers will be created (Property “SimCount”). Both properties only make sense when used together. The value for SimCount must be at least one if the random flag is set.

If the random flag is not set but the simulated flag, this passenger is also created by the test environment. In contrast to the randomly created passenger objects, this passenger has a unique identity (as specified by the other properties) and can therefore be spotted easily during the test runs.

The property “Late?” is used to define if a certain passenger will run late and will miss the flight or if the plane will be reached in time.

The screenshot shows a dialog box titled "PAX: Object" with the following fields and values:

- Gender: Female
- Title: Mrs
- First Name: Angelina
- Second Name:
- Surname: Jolie
- Birthdate: 1.1.1970
- Nationality: USA
- Passport Number: 012345
- Handicap Status: Normal
- Street Name: Pittstreet
- Street Number: 10
- Postoffice Box: 55555
- City: New York
- City Code:
- Country: USA
- Country Code:
- Mobile: +4915154627398
- Phone: 555-2345
- Fax: 555-3456
- Email: jolie@pitt.com
- Website: www.hollywood.com
- Frequent Flyer Number: 12345
- Frequent Flyer Status: Gold
- Profession: Actor
- Login: jolie
- Password: jolie
- Status Notification Level: Information
- Guidance Status:  Enabled  Disabled
- Notification Language: UK
- Notification Level: High
- Notification Media: Text
- Tracking Status:  Enabled  Disabled
- Process Status: Checked-In
- Process Value:
- Sequence Number: 11
- Booking Reference: 11
- Ticket Number: 12
- TempID: tmpid\_100
- RFID (Passive): 500
- RFID (Active): 600
- 2D Barcode: 700
- Seat Number: 2C
- Random:
- SimCount: 5
- isSimulate:
- isLate:

Figure 6.11: Passenger Properties

Each passenger object has two connections. One relationship assigns the passenger to a specific flight (see chapter 6.3.3 *Flight Database Object*). The other connection is used to specify the meal (see chapter 6.3.2 *Meal Database Object*).

The object is represented by a black and white box. The name of the passenger is displayed in the center, in case of a real person. If this object shall generate and simulate one or more passengers, the count of the simulated passengers is displayed.



Figure 6.12: Passenger Object

Also, three led bulbs indicate if this passenger object contains a real, simulated or randomly defined person. Two ports are located at the left and right side. These are used to connect the meal database and the flight database.

### 6.3.2 Meal Database Object

The passenger selects a meal and time slot before the flight. The list of the menus is taken from the caterer object (see chapter 6.2.2 *Caterer Domain Object*). The information which flights exist and when they will take place is derived from the flight database (see chapter 6.3.3 *Flight Database Object*). There, the airports and their connections are specified. The meal database object is a collector for information stored in other places of the model.

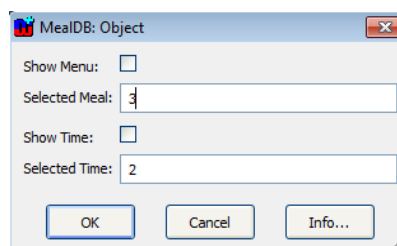


Figure 6.13: Meal Database Properties

The meal database has four properties: two of them specify the meal and the selected time slot (see figure 6.13 *Meal Database Properties*). Also, two more properties are used to present the possible list of meals and time slots for the tester. By enabling/disabling such a checkbox a menu is presented to the scenario designer (see figure 6.14 *Menu Overlay*).

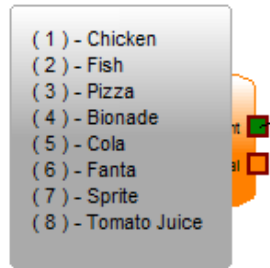


Figure 6.14: Menu Overlay

There are two relationships to other objects. First, the meal database depends on the flight as there might be various menus for different flights. Second, the meal database must be connected to a passenger.

Ports are used to assign a passenger to a meal. This is done by drawing a transition between two instances of passenger and meal. By using this concept, one type of meal can be assigned to more than one passenger which cleans up the graph because less objects need to be inserted.



Figure 6.15: Meal Database Object

The object is represented by an orange box. An icon of a plate and silverware is displayed in the upper left corner. The chosen meal and time slot are displayed in the middle of the object.

### 6.3.3 Flight Database Object

The flight database is used to assign a passenger to an existing flight. The origin and destination airports are specified in the main scenario graph (see chapter 6.2 *Main Scenario Graph*). This means that the names of the airports and the connected flights do not need to be copied manually as this data is derived. There need to be as many flight database objects in the terminal graph as flights exist in the main graph.

The flight database object has five properties. Two of them specify the date of the flight (day and month), two of them describe the time (hour and minute) when the

flight takes place. The fifth property specifies the direction of the flight whether it is an arrival or departure.

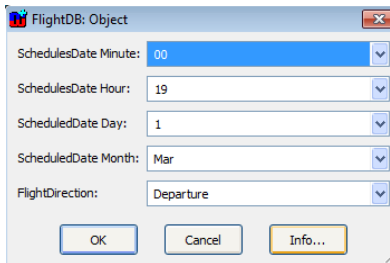


Figure 6.16: Flight Database Properties



Figure 6.17: Flight Database Object

The flight database has got two relationships with other objects. First, a connection is needed to a meal database (as the meal depends on the flight). Furthermore, the flight needs to be connected to a passenger object.

The object is represented by a green box. An icon of an aircraft with tickets is displayed in the upper right corner. In the middle of the object, the flight ID is shown (consisting of the origin airport concatenated with the departure time). Also, the abbreviations of the origin and destination airport are shown.

#### 6.3.4 Terminal Domain Connection Types

Two different connection types are used in this graph. First, a connection is needed to assign a meal database to a specific flight. This is needed as the menu depends on the chosen flight. The same connection type is used to assign a passenger to a flight.

Second, the passenger object needs to be connected to the meal database. A different connection type is used for this assignment.

### 6.3.5 Terminal Domain Checks

Ten checks are performed for this graph. Two checks are related with the flight database and two checks are performed for the meal database. Further six checks are carried out in order to ensure a well-formed passenger object.

Regarding the flight database, there must be at least one flight database object. Furthermore, it is not allowed to place more flight database objects onto the graph than existing flights.

Also, at least one meal database must exist. If this is the case, a check is performed if each meal database is connected to a flight database.

The first test for the passenger object checks if at least one is defined. Also, the passenger must be connected to a flight database. Furthermore, the passenger must be connected to a meal database — which is connected to the same flight as the passenger. A specific set of properties must be set for a passenger, so a test is performed if the mandatory properties are present. At last, two checks are executed to test if the passenger chose a meal and a time slot.

## 6.4 Caterer Domain Graph

In the caterer domain, the meals and drinks are defined. Also, menus are assembled and assigned to flight connections. This information is used in the Terminal Graph.

The caterer domain graph is part of the main scenario graph. It is represented as the caterer domain object (see chapter 6.2.2 *Caterer Domain Object*) on that graph.

The caterer domain contains four different objects: meal and drink items, an item box and a flight object.

It has got two properties. The name is used to give a meaningful information on what to expect. Furthermore, a live check flag can be switched in order to enable or disable checks while creating the graph (see chapter 6.11.2 *Test Bench Configuration Status Object*).

### 6.4.1 Drink Item Object

A drink item object is a part of a menu. This menu is served during a flight. Drinks can be chosen by passengers in the terminal domain (see chapter 6.3 *Terminal Domain Graph*).

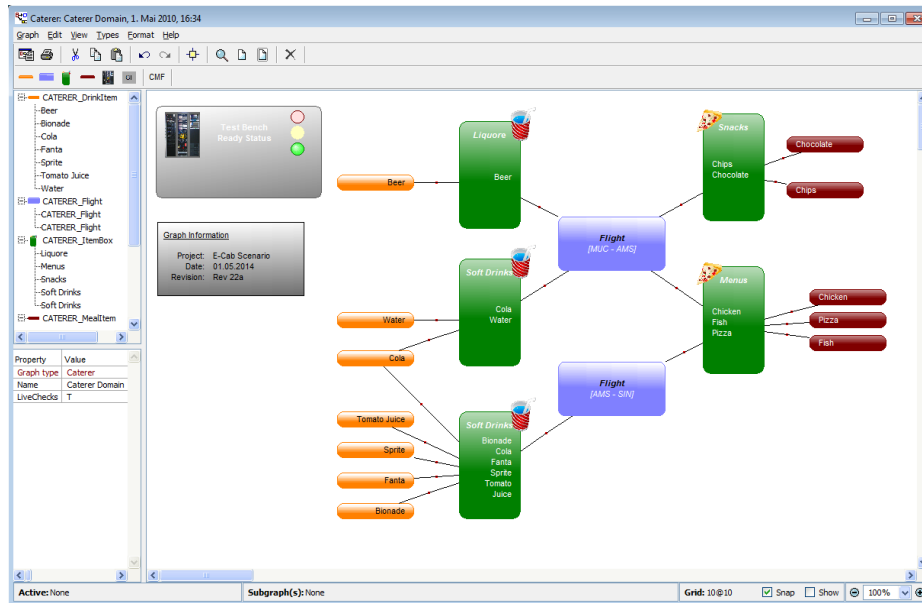


Figure 6.18: Graph Domain Caterer: Meals are defined and dedicated to flights

The drink item is a simple object which only consists of the name of a drink. The drink is part of an item box which stores menus for a flight.

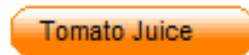


Figure 6.19: Drink Item Object

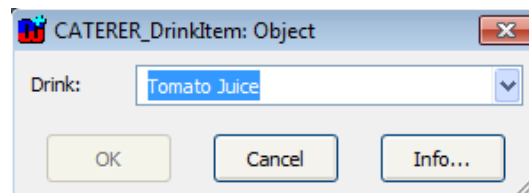


Figure 6.20: Drink Item Properties

The object is represented by an orange box. The name of the object is displayed in the middle. The alignment (left or right) of the name is towards the connected item box object. This makes it easier to comprehend which drink is assigned to the associated item box.

### 6.4.2 Meal Item Object

A meal item object is a part of a menu. This menu is served during a flight. Meals can be chosen by passengers in the terminal domain (see chapter 6.3 *Terminal Domain Graph*).

The meal item is a simple object which only consists of the name of a meal. The meal is part of an item box which stores menus for a flight.

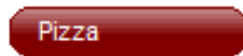


Figure 6.21: Meal Item Object

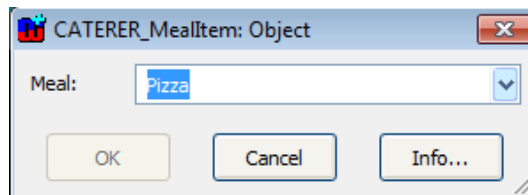


Figure 6.22: Meal Item Properties

The object is represented by a brown box. The name of the object is displayed in the middle. The alignment (left or right) of the name is towards the connected item box object. This makes it easier to comprehend which meal is assigned to the associated item box.

### 6.4.3 Item Box Object

The item box object stores meal items and drink items in order to present a menu. This menu is attached to a specific flight and can be assigned to a passenger in the terminal domain graph (see chapter 6.3 *Terminal Domain Graph*).

The item box is connected to meal and drink items in order to create a menu. Furthermore, this menu needs to be connected to the caterer flight object.

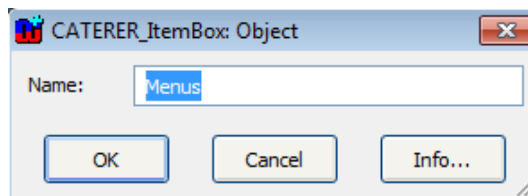


Figure 6.23: Item Box Properties



The item box has only one property which stores the name of the box.



Figure 6.24: Item Box Object (Meals Only)



Figure 6.25: Item Box Object (Drinks Only)

The object is represented by a green box. The names of the attached meals and drinks are displayed in the middle of the object. The alignment (left or right) of the names is towards the connected flight object. This makes it easier to comprehend which menu is assigned to the associated flight.

In order to distinguish between different menus, a small icon in the top left or top right corner provides a fast access to the content of such a menu. This is either a cup in case of only attached drinks or a slice of pizza in case of only attached meals. If meals and drinks are attached, both icons are displayed.

#### 6.4.4 Caterer Flight Object

The caterer flight object is used to attach meals and drinks to a specific existing flight. The food which will be served during this flight can be chosen by passengers in the terminal domain graph (see chapter 6.3 *Terminal Domain Graph*).

The information for the caterer flight object is derived from the already existing flights in the main scenario graph (see chapter 6.2 *Main Scenario Graph*). If such an object is dropped on the caterer domain graph, the shown flight information is extracted automatically. This ensures that menus are attached to existing flights.

As all information is derived, no properties are needed for this object.



Figure 6.26: Caterer Flight Object

The object needs to be connected to item box objects in order to set up meals for a flight.

The caterer flight object is represented by a blue rounded box. The abbreviations of the origin and destination airport are displayed in light colors in the middle of the object. This eases the recognition of a specific flight. The information itself is generated of the attached main scenario graph. It is traversed and all flight connections are derived from the model. Therefore, only valid flight connections are displayed and no misconfiguration is possible.

#### 6.4.5 Caterer Domain Connection Types

There is one connection type used in this graph. It connects meal and drink objects to item box objects. Furthermore, the same connection is used to link such an item box to a flight object.

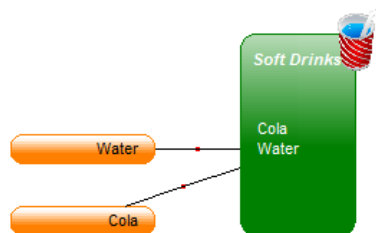


Figure 6.27: Caterer Domain Connection

### 6.4.6 Caterer Domain Checks

Four different checks are implemented for the objects in this graph. Two are related to the caterer flight object, one is related to the item box object and one detects loops between all objects.

As it is possible to create loops by connecting various objects with each other — which results in endless tries to display the information of the connected objects — it is necessary to detect these loops in order to interrupt the generators running within the objects. This is the first check which is executed by the test bench object.

Second, it needs to be ensured that there is exactly the same count of caterer flight objects as there are existing flights in the main scenario graph. If this cannot be ensured, passengers could not order meals in the terminal domain (see chapter 6.3 *Terminal Domain Graph*) as these flights would not exist for the meal database object (see chapter 6.3.2 *Meal Database Object*).

Furthermore, each caterer flight object must be connected to at least one meal or drink. This connection is the base for the meal database in the terminal graph. The meal database object would not contain any data if no connection was present here.

Consequently, each item box must contain at least one meal or drink. The connected food items are collected and provided for the caterer flight object.

## 6.5 Baggage Domain Graph

The baggage domain graph is used for displaying the different baggage domain applications which are located at the different airports. The purpose of this graph is to collect and display the configurations of all baggage applications.

The baggage domain graph is part of the main scenario graph. It is represented by the baggage domain object (see chapter 6.2.4 *Baggage Domain Object*).

The graph contains the baggage application object and the test bench object only.

Three properties exist for this graph. Firstly, it has a name to distinguish it in case of multiple occurrences. Furthermore, it contains a flag if live checks shall be performed. This flag is evaluated by the test bench object (see chapter 6.11.2 *Test Bench Configuration Status Object*). Also, a flag can be set which enables or prevents the display of additional information.

### 6.5.1 Baggage Application Object

The baggage application object displays the configuration of one baggage setting at a specific airport. The object itself can be regarded as “syntactic sugar” [coined by Peter Landin] as the settings are not defined here. The object is a view on information which is stored at the airports (see chapter 6.8.1 *System Deployment and Configuration Object*). This information cannot be changed within this object.

However, the displayed content is derived from the application configuration object. This is done automatically. The baggage application object is dropped onto the baggage domain graph. The object then traverses through the main scenario graph and collects the settings automatically. This ensures that the settings are not stored in the model twice and the configuration is well-formed. The potential of introducing mismatching settings is reduced.

As the baggage application object displays only information, it has no properties of its own. Also, no connections are necessary to other objects in this graph.



Figure 6.28: Baggage Application Object

The baggage application object is represented by a blueish rounded box (see figure 6.28 *Baggage Application Object*). The name of the airport is displayed in grey letters at the top of the object. A red and a green led bulb indicate if this application is simulated by the test bench or if the real existing equipment is used. In both cases, the communication address is displayed at the bottom of the object.

If more objects are placed onto the graph than existing application configurations, the led bulbs are turned to red and the error message “not defined” is displayed (see figure 6.29 *Baggage Application Object is Not Defined*).

### 6.5.2 Baggage Domain Connection Types

No connection types are needed for this graph.



Figure 6.29: Baggage Application Object is Not Defined

### 6.5.3 Baggage Domain Checks

One check is implemented for the objects in this graph. This check ensures that there are no more baggage application objects than existing application configurations (see figure 6.29 *Baggage Application Object is Not Defined*).

Even though this check has no influence on the generated target code or the execution of tests, it helps to create a consisting well-formed model.

## 6.6 Airport Domain Graph

The airport domain graph is used to cluster two different subdomains. These two independent topics are related to the construction of the airport and the systems which are installed there.

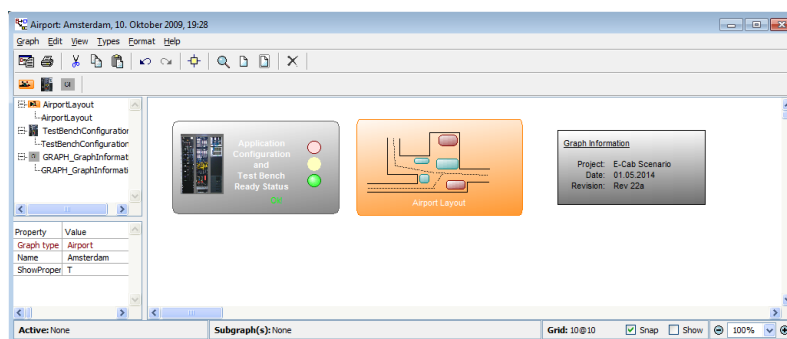


Figure 6.30: Airport Domain Graph

This means that the layout of the gates, shops and other localities need to be described. This is done by the airport layout object (see chapter 6.6.1 *Airport Layout Object*).

The configuration of the systems and their behavior is described by the test bench configuration object (see chapter 6.6.2 *Test Bench Configuration Object*).

The airport domain graph can be accessed by the airport domain object (see chapter 6.2.3 *Airport Domain Object*).

This graph is kept rather small as the single intention is to keep both subdomains on the same level — and hiding the details in the subgraphs within the two objects airport layout and test bench configuration.

### 6.6.1 Airport Layout Object

The airport layout object is a placeholder for the actual layout of the airport. The object itself has no properties and no connections to other objects.

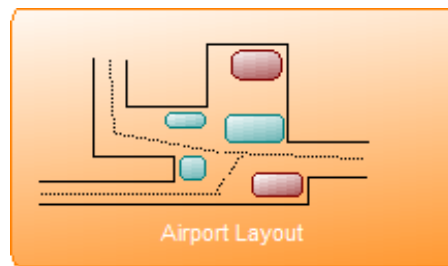


Figure 6.31: Airport Layout Object

The object is represented by an orange rounded box with a stylized floor plan.

### 6.6.2 Test Bench Configuration Object

The test bench configuration object is a placeholder for the actual settings and configuration of the airport systems. The object itself has no properties and no connections to other objects.

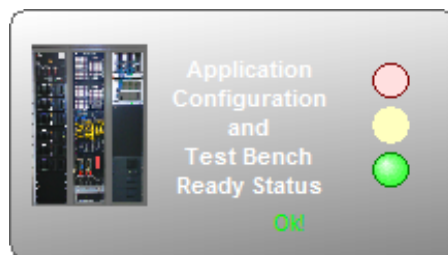


Figure 6.32: Test Bench Configuration Object

The object is represented by a gray rounded box. An icon showing a test bench is located on the left side. On the right side and on the bottom line, the status of the attached graph is presented. The object has a checker implemented which presents the status of the underlying graph. If the attached graph is found to be semantically and syntactically correct, the green led of the “traffic lights” is active. If the red led is switched to on, the configuration of the airport systems is not correct. If no checks are performed, the yellow light is active.

## 6.7 Airport Layout Graph

The airport layout graph is the exact reflection of a part of an original airport or area. It is used for the position application guidance function (see chapter 5.4.5 *Position Application*) and the notification system (see chapter 5.4.3 *Notification System*).

The airport layout consists of border objects which can be assembled to form a site plan. In fact, this graph contains two different views on the airport. First, this layout can be completed by adding location objects, like shops, cafés and bars. Then, the layout looks like a common site plan which can be found at each airport.

Second, by dividing the layout into sections (represented by area objects) and by hiding the location objects so that only the border objects and area objects are displayed, the guidance function can be configured.

By using two different views on the same graph, multiple layers of information are created (see chapter 3.6.2 *Views*). This enables different projections on the underlying model. This technique is used to create an airport layout and the configuration of a system at the same time without interfering with each other. However, two graphs could have been used instead but then the common details of both graphs would have been needed to be copied between them. Also, by grouping all necessary information in one place and not scattering it over various places, the management of details becomes simpler and the consistency of data can be ensured without greater effort.

Furthermore, regardless of the view, passenger objects are displayed on the graph. These objects represent real and simulated passengers. They opted-in for the guidance function and are therefore tracked at the airport. Their location is mapped to the graph in real-time.

This graph (see figure 6.33 *Graph Airport Layout*) contains such an airport layout, including all locations (see chapter 6.7.5 *Airport Location Object*) and paths between them. The paths are annotated with possibilities so that more complex simulations can be computed (all simulated passengers shall not take the same route through the airport).

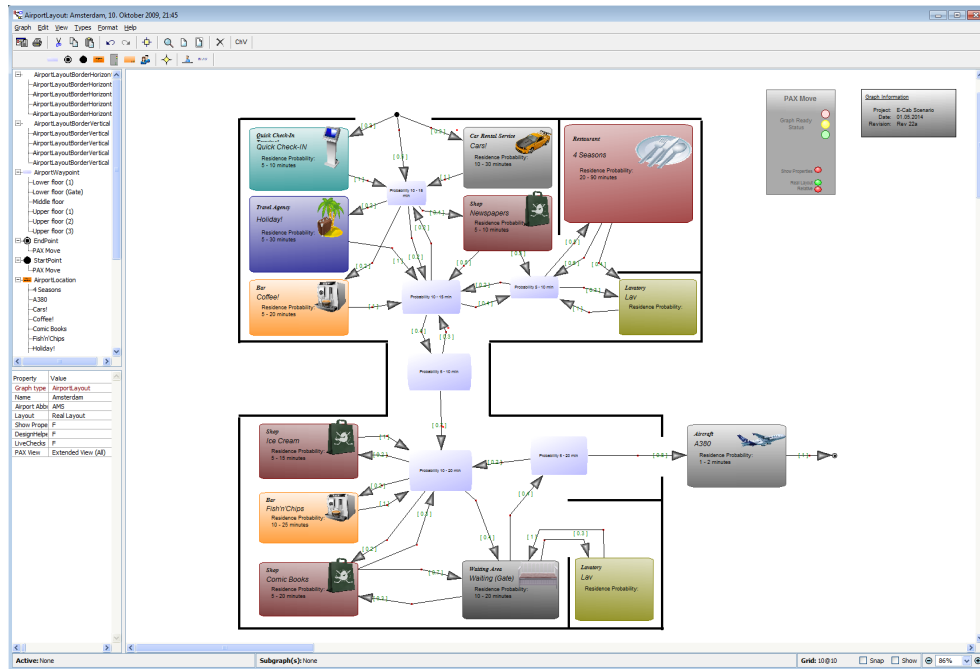


Figure 6.33: Graph Airport Layout

Each passenger is assigned to a specific flight. Each flight has a departure time. If the passenger has enough time to visit one of the locations (based on the properties which define the time range the passenger will stay at the location) the route might be chosen by the test data generator framework (see chapter 8 *Test Data Generation*).

With Dijkstra's shortest path algorithm [17], it is calculated whether a passenger has enough time to visit one or more of the locations or if the passenger needs to go to the next location in order to proceed to the gate.

Also, paths can be calculated by regarding routes of other passengers. If the guidance system tracks a lot of passengers in one location or area, it tries to route other passengers around the crowded places. Else, the speed of the passengers will slow down and the plane might not be reached anymore. Therefore, another route is calculated and the passenger is informed via push service (e.g. SMS).

The graph itself has six properties. First of all, a name needs to be assigned. Furthermore, the layout needs to be specified. Either a real layout is used, then the dimensions of the location objects are used (and the graph can be regarded as an exactly scaled model of the original layout) or relative coordinates are used. In this case, the dimension properties of each location object are used (see chapter 6.7.5 *Airport Location Object*).



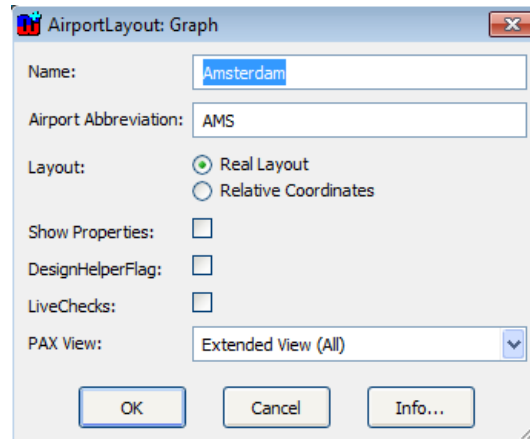


Figure 6.34: Airport Layout Graph Properties

Three properties are used for assistance while creating the graph. If “show properties” is enabled, more information is shown for the displayed objects. The design helper flag displays information about the current position of an object, which makes it easier for the designer to place it to a certain position. If “live checks” is enabled, the graph is constantly monitored and possible errors in the graph are displayed immediately.

If too many passengers are present at the airport, it could be difficult to spot some of them. Therefore, filters can be applied. For instance, if not all information should be displayed, each passenger could be represented by only an icon. If more information is needed, the extended view can be switched on. In this case, some basic data is displayed additionally. Furthermore, if only passengers need to be identified who will not catch their flight, the switch could be flipped to the appropriate position. Also, all passengers could be displayed who will reach the gate in time.

### 6.7.1 Horizontal Border Object

The horizontal border object is a simple one. It has no properties or connections to other objects. It is used to create a floor plan for the airport.

---

 Figure 6.35: Horizontal Border Object

The object is indented as “syntactic sugar”. It is used to draw a visual boundary but is not evaluated or used in any process.

The object is displayed in both graph modes (PAX Move and position application).

### 6.7.2 Vertical Border Object

The vertical border object is a simple one. It has no properties or connections to other objects. It is used to create a floor plan for the airport.



Figure 6.36: Vertical Border Object

The object is indented as “syntactic sugar”. It is used to draw a visual boundary but is not evaluated or used in any process.

The object is displayed in both graph modes (PAX Move and position application).

### 6.7.3 Startpoint Object

The start point sets the beginning of the passenger movement. From this location, the passenger will move into the airport area. Afterwards, this location will never be passed again.

The object has no properties. It has only outgoing connections to either the end point (see chapter 6.7.4 *Endpoint Object*), waypoints (see chapter 6.7.6 *Airport Waypoint Object*) or locations (see chapter 6.7.5 *Airport Location Object*).



Figure 6.37: Startpoint Object

The endpoint object is designed as a black circle.

The object is displayed in both graph modes (PAX Move and position application).

### 6.7.4 Endpoint Object

The endpoint object is used to mark the end of a path through the airport. There are two possibilities to terminate such a trail. Either this endpoint object is used or

the airport location object is used if its type is set to “aircraft”. This is also a valid endpoint for a defined path.

The object has no properties. It has only incoming connections from either the start point (see chapter 6.7.3 *Startpoint Object*), waypoints (see chapter 6.7.6 *Airport Waypoint Object*) or locations (see chapter 6.7.5 *Airport Location Object*).



Figure 6.38: Endpoint Object

The endpoint object is designed as a black circle within a gray one.

The object is displayed in both graph modes (PAX Move and position application).

## 6.7.5 Airport Location Object

The airport location object is a place at the airport like a bar, restaurant or shop. The locations are placed within the bounding boxes of the floor plan (see chapter 6.7.1 *Horizontal Border Object* and 6.7.2 *Vertical Border Object*).

Figure 6.39: Airport Location Properties

Each location has got a type. This is a predefined list and the creator of the floor plan can choose between ten different types: travel agency, check-in kiosk, shop, bar, restaurant, lavatory, waiting area, car rental service, money exchange and aircraft. If the type is set to “aircraft”, this location also acts as an endpoint (see chapter 6.7.4 *Endpoint Object*). Also, each location has a property to define its name.

The locations are placed onto the airport layout graph. In order to create a fitting plan, each location needs a position and size. This can be achieved in two different ways. If the graph property “layout” is set to “real layout”, the actual sizes and

positions of the objects are used. This means that the layout you see is the layout you get. The properties of the location object are not regarded then.

However, if the graph property is set to “relative coordinates”, the values for the position are extracted from the properties of the location object. These properties are defined by setting the values for the top, bottom, left and right coordinates.

A passenger will reside in such an area for some time. If the location is entered, the passenger stays for the specified period. The lower and higher bound are defined by two properties of this object. Furthermore, a probability is assigned to each location. If the locality is on the path of a passenger, he does not need to enter it under all conditions. The test data generator calculates individual routes for every passenger. The probability is used to define a likelihood of entering the location — if this would not be the case, all passengers would have the same route within the airport.

Via the RFID properties, the tracking mechanism for this location is specified.

Requirements can be entered for each location as well. This includes ID for high-level and low-level requirements. If the particular flag is switched-on, the requirements will be tracked during the execution of the test. In order to refine this tracking, three more values can be entered. First, a minimum and a maximum of visits can be defined. This means that this requirement will be fulfilled if the amount of passengers which will reach this location are between these values. Furthermore, if a pre-defined “count” can be added as well. If this value is set, it will be used as initial value (e.g. if set to five then the next visit will increase the “visit counter” to six). Values set to zero are not regarded in case of minimum and maximum.



Figure 6.40: Airport Location Object

The location objects have connections from the start point and to the end point. Also, connections between location objects and waypoints are allowed in both directions.

The object is represented by a rounded box. The color depends on the type of the location. For instance, a bar is colored orange while a restaurant is tinted in red (see figure 6.40 *Airport Location Object* and figure 6.41 *Airport Location Object with Show Properties*). Also, an icon is displayed in the upper right corner which reflects the type. Under the type, which can be found in the upper left corner, the name is



Figure 6.41: Airport Location Object with Show Properties

displayed. In the middle, the residence probability is shown.

If the airport layout graph has switched on its property “show properties” flag, the coordinates of the object are displayed in the lower part. This can be used as means for the designer while creating the floor plan.

The object is only displayed in the PAX Move graph mode.

## 6.7.6 Airport Waypoint Object

An airport waypoint object is a kind of airport location. However, this location represents the floors and paths between the stores and other areas of the airport. The waypoint object is designed to simulate the time consumed by passengers walking through the airport.

A waypoint has 17 properties which are divided into three sections. The first section contains a name (even though it is only used for the modeler as an internal reference) and the coordinates for this object. As described before, these values are only used if the graph property “layout” is set to “relative coordinates”. The retention time is defined by two values: one for the lower bound and one for the upper bound (see figure 6.42 *Airport Waypoint Properties*).

The other two sections contain the information which is needed for the tracking of requirements. This includes the ID for high-level and low-level requirements, a flag indicating whether the test bench shall regard and therefore record the tracking. In this case, the location “Middle floor” shall be visited at least once. There is no specific count for a maximum of visits (as it is set to zero) and the initial count is also not specified, respectively set to zero.

A waypoint is connected to other waypoints and/or airport locations (see chapter 6.7.5 *Airport Location Object*).

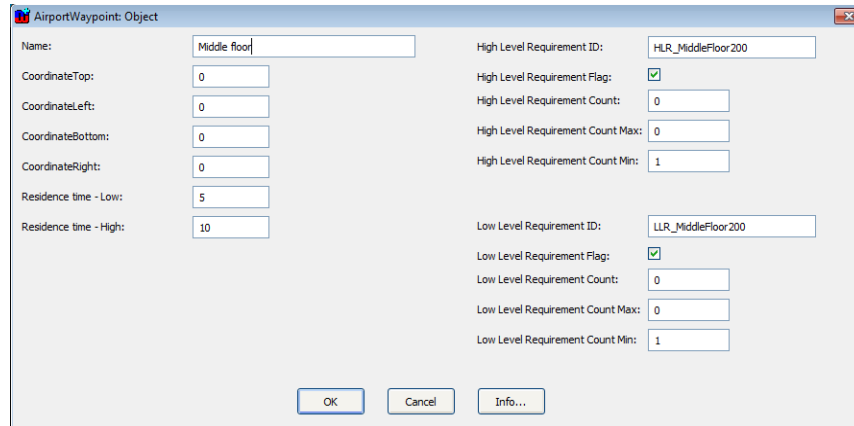


Figure 6.42: Airport Waypoint Properties

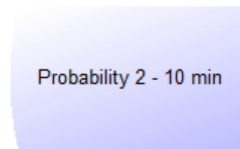


Figure 6.43: Airport Waypoint Object

The waypoint object is designed as a purple rounded box. The retention time is displayed in the middle of the object.

The object is only displayed in the PAX Move graph mode.

### 6.7.7 Passenger Move Object

When the test is running and the passengers (real and simulated) are walking through the airport, the passenger move object is used to reveal the actual position of these people.

As depicted in figure 6.44 *Passenger Move Properties*, the passenger has got seven properties. The first five properties are derived from the passenger definitions in the terminal domain (see chapter 6.3.1 *Passenger Object*). This information is set during runtime by the test. It is used to identify and distinguish between the passengers (in most cases, more than one passenger will be present on the airport).

The “is late” flag is also defined in the terminal domain for this passenger. This indicates if the passenger will reach his plane in time.

The “get late status” is calculated during runtime. If the status is calculated to be “is not late” then the persons position is in a suitable airport area (see chapter 6.7.8 *Airport Area Object*). An “is late” state is calculated if the passenger was present in an area for too long and cannot make it in time for his flight. However, there is a gap between these two states: too late means that there is no chance to catch up the delay, early enough means that there is plenty of time and there is no hurry. The gap in between marks the period of time where the passenger will hit the gate in time if he proceeds directly to the next area.

Figure 6.44: Passenger Move Properties

The passenger move object has no connection to other objects.

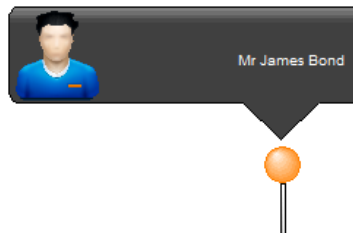


Figure 6.45: Passenger Move Object

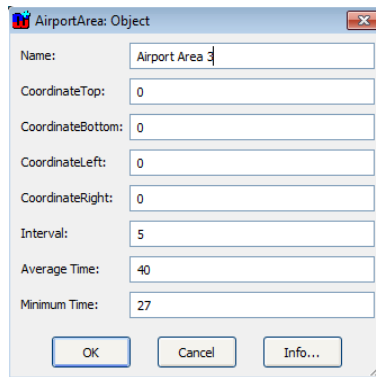
The object is designed as a pin. The pinhead indicates if the passenger will reach his flight in time. If the passenger might get late, the pinhead is colored orange (like depicted in figure 6.45 *Passenger Move Object*). If the passenger is too late and will not make it, the pinhead turns red. If there is enough time, the pinhead is colored green.

The name of the passenger is displayed above the pin in the legend. Also, a male or female icon is displayed according to the sex of the passenger.

The object is displayed in both graph modes (PAX Move and position application).

### 6.7.8 Airport Area Object

This graph divides the same airport layout (from the layout graph, see figure 6.33 *Graph Airport Layout*) into different areas (like an overlay, see figure 6.47 *Graph Airport Area*). The exact location of a passenger is tracked by the guidance system and then converted into areas. This shall give a good trade-off between privacy and the chosen service of guiding as only the information about the area is reported and processed.



Property	Value
Name	Airport Area 3
CoordinateTop	0
CoordinateBottom	0
CoordinateLeft	0
CoordinateRight	0
Interval	5
Average Time	40
Minimum Time	27

Figure 6.46: Airport Area Properties

The airport area object has got eight properties. First, the object needs a name. Furthermore, the positions of this area need to be specified — if the Airport Layout Graph property “Layout” is set to “Relative Coordinates”. In this case, the properties top, bottom, left, and right need to be filled with values unequal zero. If this is not the case, then the current position of the graph is used.

Three properties are used by the guidance system during execution of the test. The interval value specifies the distances between notifications (text messages are sent to the user’s mobile device). The average time describes the time a passenger is likely to stay at this zone at the airport. The minimum time depends on the airport locations and their paths within this area. This is the minimum time a passenger needs to walk from the entrance to the exit of such a sector.

Each airport area object has got a connection to the next airport area. As the contained information of average and minimum residence time is needed for the guidance system, only the way to the aircraft is important and therefore modeled as such a connection. The specified values are displayed on the object. In addition, the values for all areas (in that path) are calculated and displayed on the object as well.

The object is designed as an orange colored rounded box. The name of the area is displayed in the middle of the object.

The object is only displayed in the position application graph mode.



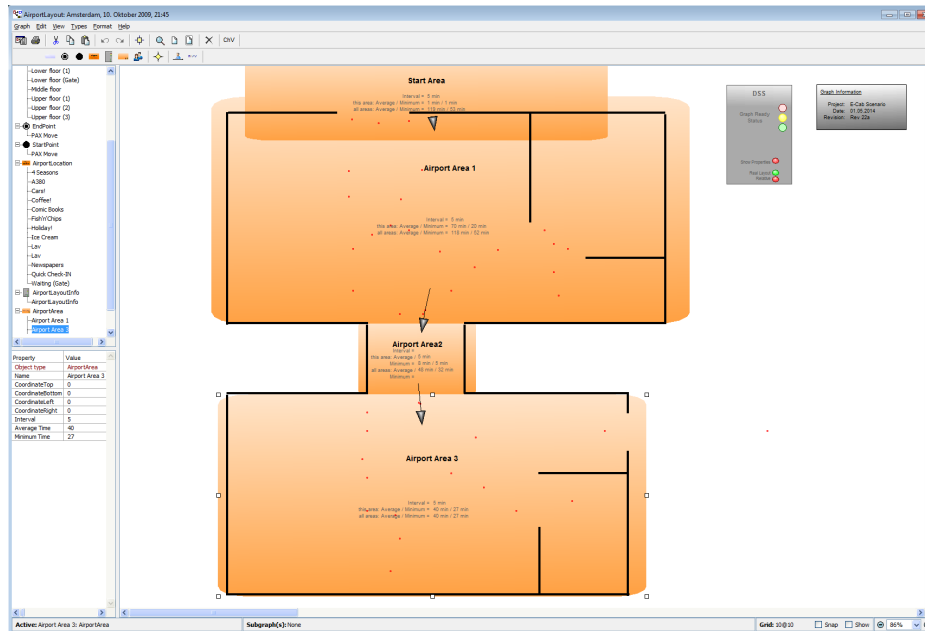


Figure 6.47: Graph Airport Area

### 6.7.9 Airport Layout Info Object

The airport layout info object has got two functions. It is used while creating the floor plan and can be removed later on.

It displays the current design mode (PAX Move or position application). Also, the current settings of properties of the graph are shown. This includes the “show properties” value also as the current setting of the layout.

Furthermore, it contains a graph ready status which signals the modeler if the graph is well-formed. If this is not the case, hints are given which object does not conform to the specification or intended use. The graph ready status displays the outcome of the checks for the current type of graph (PAX Move or position application). This means that, if errors are present in the other type, they would not be displayed here. This is an intended behavior as only the errors of the current graph can be corrected. To reveal faults in the other type of graph, the mode must be switched.

The object itself has no properties and there are no connections to other objects.

The object is represented by a gray box. In the upper part, the current mode is displayed. Right below the mode, the calculated graph ready status is shown. Any errors would be displayed in the middle of the object. In the lower part, the current graph settings are shown.

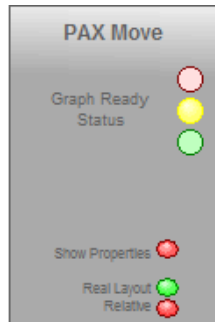


Figure 6.48: Airport Layout Info Object

The object is displayed in both graph modes (PAX Move and position application).

### 6.7.10 Airport Layout Connection Types

There are two types of connections for the objects in this graph. Depending on the mode, either one or the other is displayed. This allows two independent views on the graph (see chapter 6.7 *Airport Layout Graph*).

The first type of connection is displayed in the PAX Move mode only. It is used to connect the start point, end point, airport location objects, and airport waypoint objects with each other. This symbolizes the path a passenger might walk through the airport.

Each path is annotated with a probability. This is used for the test data generator (see chapter 8 *Test Data Generation*). By using likelihoods, different paths for the passengers can be expected.

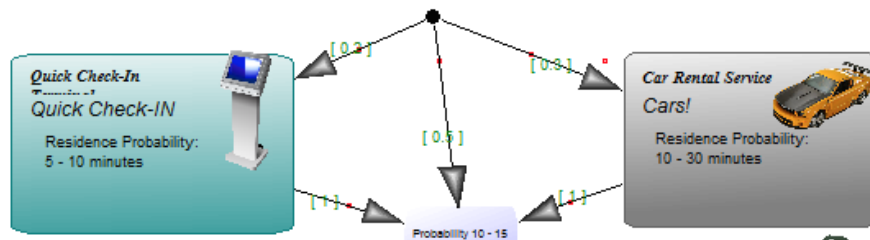


Figure 6.49: Airport Layout Connection Types

As shown in figure 6.49 *Airport Layout Connection Types*, the sum for the probabilities of all outgoing paths must equal 1. If this is the case, the probabilities are displayed in green letters. In order to indicate an error, red colored letters are used if the sum of the probabilities does not equal 1.

The second type of connection is displayed in guidance mode view only (see chapter 6.7.8 *Airport Area Object*). For each area, a time range is defined. The lower bound of the range defines the minimum of time ( $t_{High}$ ) a passenger needs to proceed from one area to the next one. Also, a second bound is specified ( $t_{Low}$ ). It represents the maximum bound. This shall ensure that not all passengers will move directly to the gates and the last location will be too crowded.

The third parameter ( $t_{Interval}$ ) describes the interval (see figure 6.50 *Timeline*) in which the next push service notification is sent (e.g. a value of 600 means an SMS is sent every 600 seconds).

If a passenger is about to be late, he receives a notification to remind him to proceed to the next area which could be the gate or the next zone of the airport (e.g. the passenger needs to pass the security checks or leave the duty free section). The behavior is shown in figure 6.50 *Timeline*.

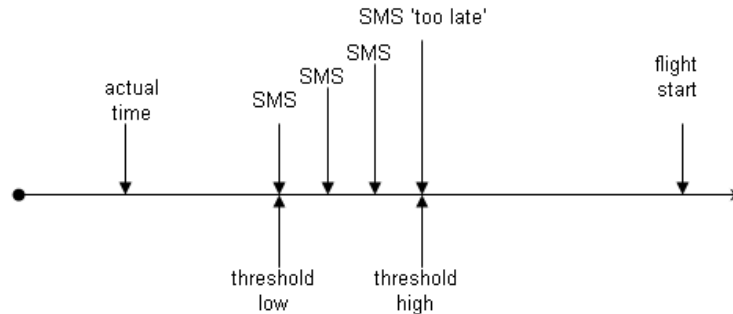


Figure 6.50: Timeline

If the actual time  $t$  is less than  $flightStart - t_{Low}$ , no SMS is sent to the passenger. If this  $t_{Low}$  value is exceeded, a first notification is sent. When the timeout  $t_{Interval}$  elapsed, another message is sent. This is done until the  $t_{High}$  value is exceeded. In this case, the passenger is informed by a “too late” message that he cannot reach the plane anymore.

### 6.7.11 Airport Layout Checks

Several checks are needed to ensure a well-formed graph. Eleven checks are implemented for the PAX Move view and eight checks are carried out in the position application view.

In the PAX Move view, exactly one start point must exist. Furthermore, at least one outgoing transition from the start point must be present. If an end point exists, there must be at least one incoming transition.

Furthermore, locations and waypoints cannot be used as start or end points. This means that each location needs at least one outgoing and one incoming transition (except if the type is set to A/C as this can act as an end point — see chapter 6.7.5 *Airport Location Object*). The same rule applies to the waypoint object: at least one incoming and one outgoing transition must be present.

Both, locations and waypoints have a retention time property. This needs to be checked as well. For instance, the lower bound of the dwelling time must be greater than zero. Also, the higher bound must always be greater than the lower bound. These two rules are carried out for the location and the waypoint object.

If the graph property (see chapter 6.7 *Airport Layout Graph*) “Layout” is set to “Relative Coordinates”, the values for the object’s position must be stored within each object. Therefore, a check is performed if all values for the properties top, bottom, left, and right are unequal zero. No more checks are necessary as the property values are sorted out in the test. Therefore, if the value for bottom is greater than the top, the area is calculated.

A name must be present for each location. Also, each waypoint needs a name as well.

The values of the probabilities on each transition are checked as well. The sum of all probabilities of all outgoing transitions from one location or waypoint must equal one. This must be the case in order to calculate the different paths for the simulated passengers.

Some of these tests are also performed when the decision support system view is selected. As well as in the PAX Move view, only one start point is allowed and there must be at least one outgoing transition. If an end point exists, it must have at least one incoming transition. Each area needs at least one incoming and one outgoing transition. Also, the name property must be set for all areas. If the graph layout is set to “Relative Coordinates”, the values for the properties top, bottom, left, and right are unequal zero.

Regarding the transitions, the values  $tLow$  and  $tHigh$  are evaluated. First,  $tLow$  must be greater than  $tHigh$  for each transition. Furthermore, the values  $tLow$  and  $tHigh$  depend on the position within the graph. For instance, the value for  $tLow$  of the transition from the start point to the next area must be greater than the value of the transition from the last area to the aircraft. The same applies for the  $tHigh$  property.

The variable *Interval* is also controlled as the value must be greater than 60. This means that the interval between two notifications will be at least one minute.

## 6.8 System Deployment and Configuration Graph

The system deployment and configuration graph defines the properties of all related E-Cab systems on a specific airport.

The graph can be accessed by the test bench configuration object (see chapter 6.6.2 *Test Bench Configuration Object*).

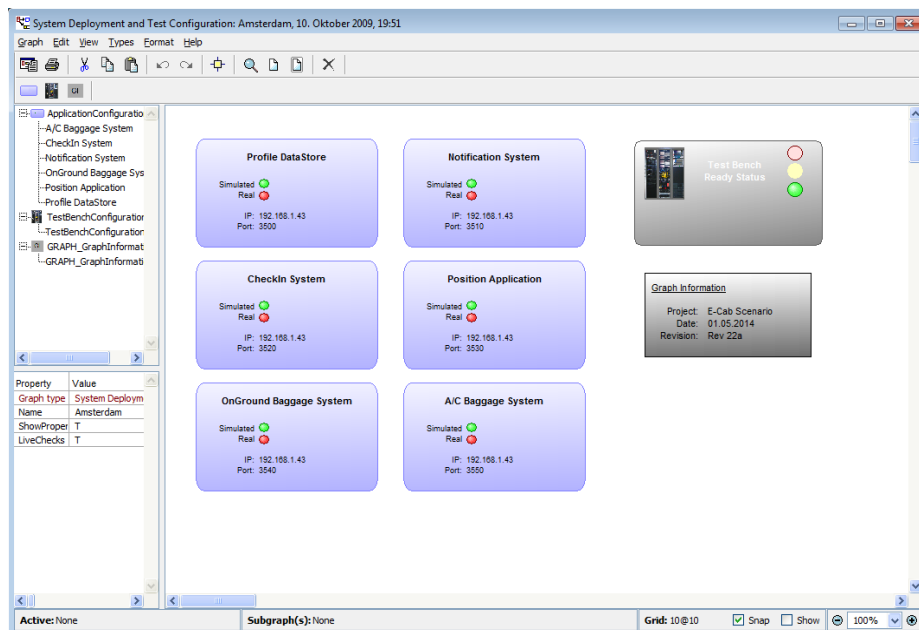


Figure 6.51: System Deployment and Configuration Graph

The graph contains objects for all related systems at this airport. These systems need to be named and their properties need to be defined. Each system could exist as an original system or a simulated one and in both cases it must be accessible from the network.

Three properties exist for this graph. Firstly, it has a name to distinguish it in case of multiple occurrences. Also, a flag is used if additional information should be displayed. This flag is used by the system deployment and configuration objects. Furthermore, it contains a flag if live checks shall be performed. This flag is evaluated by the test bench object.

### 6.8.1 System Deployment and Configuration Object

The system deployment and configuration object is used to define if a system needs to be simulated (and therefore, if code needs to be generated from the model) or if an original system is used. In both cases, the connection details must be present.

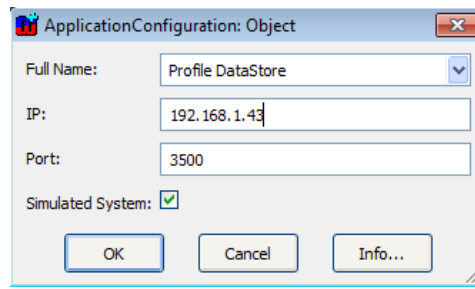


Figure 6.52: System Deployment and Configuration Object Properties

The object has three properties. Firstly, the name of the system needs to be specified. This can be chosen from a drop-down list where all the systems of the airport are listed. This eases the selection and prevents a misspelled and therefore not existing system.

Furthermore, the contact details must be filled in. In this case, this is the IP address and the port number under which the service can be reached.

This object defines solely the static information of the systems. The interaction between systems is specified in the behavior graphs (see chapter 6.9 *Behavior Graph*). Therefore, no connections between the objects are necessary in this graph.

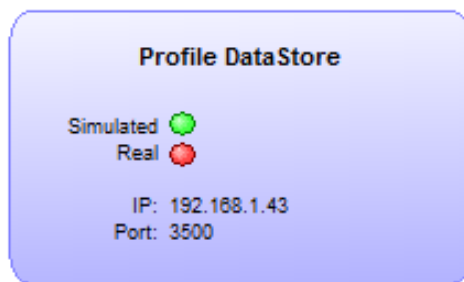


Figure 6.53: System Deployment and Configuration Object

The object is represented by a blueish rounded box. The name of the system is displayed in the upper part. Two led lights indicate if the system is simulated or if an original one is used. In the lower part, the connection details are shown — in case the flag for the additional information is switched on.

## 6.8.2 System Deployment and Configuration Checks

Four checks are implemented. Firstly, each system needs to be represented by an object. If there are too many or too less objects compared to systems, the graph is not well-formed.

Furthermore, each system is only allowed to occur once. No multiple definitions are permitted.

If two systems are running on the same host (identified by the same IP address), they need to listen on two different port numbers.

Finally, for each system, a behavior needs to be defined. This is necessary in case of a simulated system as the behavior is translated into code and is executed during the test. Also, the behavior can be used as an oracle and must therefore be present.

## 6.9 Behavior Graph

The behavior graphs describe the internal processes of the E-Cab systems (see chapter 5.4 *E-Cab Systems*). Basically, the systems can be described as reactive systems which means that they receive information, process this incoming data and send out messages accordingly. Therefore, the graph contains objects which represent these actions. By combination of these objects, complex behavior can be specified.

The system logic of most E-Cab systems can be categorized into six main functions:

- Wait for incoming messages
- Handle incoming messages
- Store message (into local storage)
- Erase message (from local storage)
- Send message (with content from local storage)
- Wait for a specified amount of time

The breakdown into rather small and simple to use but complex objects makes the definition of the system behavior rather clear because it can be described by only few behavior functions. As much information as possible is derived from the template graph in order to reduce errors and mismatches in the further processing.

The objects are designed to show only the necessary data and exclude all programming language details. For instance, except the send message object, all objects do not

have any properties. All needed information is derived from the template graph (see chapter 6.10 *Template Graph*). The logic to retrieve the displayed information is embedded in the object so that no manual interaction is needed. Therefore, each graph consists only of the condensed information, necessary for this task. The graphs are clear and simple.

These five logic blocks can be regarded as the main functions. Further logic needs to be defined in special purpose objects. These objects cover a system specific functionality.

Additionally to the six main objects, three special behavior objects have been implemented which are necessary for performing tasks as they cannot be described by the main functions. One object is created for the specified task of processing updates of a passenger's location in order to keep the exact position private but also provide enough information for helping to catch his flight. The second object can be used as an empty stub. The modeler can insert code which is incorporated in the creation of the target. The third object calculates possible area changes for one passenger and updates an internal database which is used for sending out push notifications to passengers.

Each object has a different color in order to ease the comprehension. The logic blocks can be distinguished from each other by using this color code.

The behavior graphs are part of the system deployment objects (see chapter 6.8.1 *System Deployment and Configuration Object*) and can be accessed by selecting an E-Cab system.

Two properties exist for this graph. Firstly, it has a name to distinguish it in case of multiple occurrences. Furthermore, it contains a flag if live checks shall be performed. This flag is evaluated by the test bench object.

### 6.9.1 Erase Object

The content of arrived messages might need to be erased (either by a trigger event or time constraint). In most cases a message is received which transfers information about the deletion of certain stored data. For instance, if a passenger is late and cannot reach the plane, the baggage needs to be unloaded. When the baggage is unloaded, its ID — and also associated data — must be deleted.

As the deletion depends on the type of the incoming message, this object does not need any properties. The database and its key is derived from the message type. This information is derived from the template graph (see chapter 6.10 *Template Graph*).



System behavior connections can be attached from and to all other behavior graph objects.



Figure 6.54: Behavior Erase Object

The behavior erase object is depicted by a rounded brownish box. The name of the object is displayed in the upper part. In the lower part, the name of the derived database and its key is displayed.

## 6.9.2 Handle Incoming Message Object

This object is responsible for parsing the incoming message. It handles the content over to the appropriate sub-functions. This function acts as a gateway for all further processing.

The object needs at least one incoming and one outgoing transition from and to the wait for messages object (see chapter 6.9.6 *Wait for Messages Object*). All incoming messages are forwarded and then processed by this object. All recognized messages are relayed to the attached objects. If a message is not recognized, it will be discarded and returned to the wait for messages object. The connection from this object to the wait for messages object is “syntactic sugar” and not explicitly needed. It helps the understanding of the control flow if an unknown message is received. Without this connection, it could be assumed that the object is still active which would mean that no more incoming messages would be processed.



Figure 6.55: Behavior Handle Incoming Message

The behavior handle incoming message object is depicted by a rounded greenish box. The name of the object is displayed in the upper part.

### 6.9.3 Opaque Object

This object is used as an empty stub. If no predefined block fits the needed behavior, c/c++ code can be inserted here. The needed frame is constructed and inserted in the simulation code. This object is used if a functionality is needed but is used only once and the functionality itself is rather limited. If this is the case, a simple line of code can be inserted. In all other cases, a special object needs to be created additionally (for instance, see chapter 6.9.7 *Process Update Object*).



Figure 6.56: Behavior Opaque Object

The object is represented by an orange rounded box. The name of the object is displayed in the upper part.

### 6.9.4 Send Message Object

This object constructs a new message with content from the local storage. The information for this message is retrieved from the local database.

Two properties must be defined for this object. First, the destination needs to be specified (for instance, the “notification system”) and second, the message type. The function will then derive all further information from the local data storage which is defined in the template graph (for instance, the used database and the key).

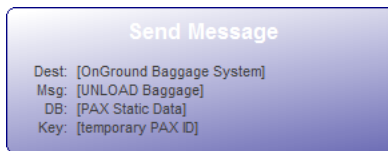


Figure 6.57: Behavior Send Message Object

The object is represented by a blueish rounded box. The name of the object is displayed in the upper part. In the lower part, the destination system is displayed. The name of the message type is shown right below. Furthermore, the name of the database and its key is displayed.

### 6.9.5 Store Message Object

In most cases, the content of the incoming messages needs to be stored. It must be disassembled and the atomic information must be extracted. The content of the message is derived from the message structure defined in the template graph. Also, the database name and its key is located there and retrieved by the object.

No properties are necessary for this object as the database name and its key are derived from the incoming message type.

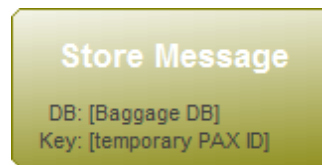


Figure 6.58: Behavior Store Message Object

The object is represented by a dark yellow box. The name of the object is displayed in the upper part. In the lower part, the database and its key are displayed.

### 6.9.6 Wait for Messages Object

This object waits for incoming messages. In this case it will listen on a defined port for SOAP messages. If such a message is received, it will trigger the incoming message handler which will process the message and relays it to the appropriate objects. Afterwards it returns immediately and listens for more incoming messages. This object has no properties.

At least two connections are necessary. One outgoing connection to the incoming message handler is necessary as the sorting and processing of the message will take place there (see chapter 6.9.2 *Handle Incoming Message Object*). Furthermore, in order to emphasize the direct return in case of a not recognized message, an incoming connection from the incoming message handler is needed (“syntactic sugar”).

Also, if a control flow is executed and all objects are processed, a transition is necessary from the last object to the wait for messages object. Again, this is not needed from the code point of view but helps to describe the control flow.

The object is represented by a blueish rounded box. The name of the object is displayed in the middle.

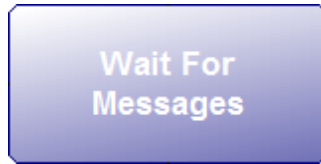


Figure 6.59: Wait For Incoming Messages Object

### 6.9.7 Process Update Object

The process update object receives a message with a location update from a passenger and performs the calculation and storage of the areaID of the current location at the airport. This information is used later on to calculate if a passenger might run late (see chapter 6.7.8 *Airport Area Object* and figure 6.50 *Timeline*).

In order to retain a passengers privacy, the position tracking and position evaluating functions are separated into two independent working systems. The process update function realizes the first part (see chapter 6.9.8 *Calculate Notification Object*).

This is one of the three special objects (see chapter 6.9.3 *Opaque Object*, 6.9.8 *Calculate Notification Object*). The behavior cannot be expressed by the six generic objects and it is too complex to use the opaque object (as too much code with access to internally generated structures would be needed).

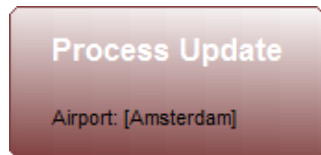


Figure 6.60: Process Update Object

The object is represented by a brownish rounded box. The name of the object is displayed in the upper part. In the lower part, the current airport is displayed.

### 6.9.8 Calculate Notification Object

The calculate notification object receives a message in order to inform a passenger. This message contains an areaID and a passenger ID. This object realizes the sending of push notifications to the passenger.

In order to retain a passenger's privacy, the position tracking and position evaluating functions are separated into two independent working systems. The process update function realizes the second part (see chapter 6.9.7 *Process Update Object*)

This is one of the three special objects (see chapters 6.9.7 *Process Update Object* and 6.9.3 *Opaque Object*).

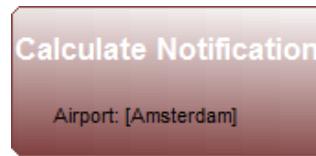


Figure 6.61: Calculate Notification Object

The object is displayed as a brownish rounded box. The name of the object is displayed in the upper part. In the lower part, the current airport is displayed.

### 6.9.9 Wait Object

The wait object does not depend on an incoming message. Its sole use is to decrease the execution time by introducing a wait cycle. The only property which can be defined is the amount of time in minutes.

This object is used to simulate real-world processing times. For instance, it takes some time to transfer luggage from the airport to the aircraft. The luggage is received at the check-in counter and routed through transport belts and trolleys into the aircraft. By inserting a waiting time between receiving and sending a message, this real-world processing time is incorporated into the simulation.



Figure 6.62: Wait Object

The object is represented as an orange colored rounded box. The waiting time is displayed in the middle area.

### 6.9.10 Behavior Connection Types

There is only one transition type to connect the behavior objects. The transition has one property which is a message type (like “REGISTER (Position Application)”, see chapter 6.10.2 *Message Object*). This information might be used in the target object.

The displayed information of this transition depends on the source and target objects. Only if the outgoing object is of type handle incoming message object (see chapter 6.9.2 *Handle Incoming Message Object*) the information of the message type is displayed (see figure 6.63 *Behavior Example*). Of course, the message type will not change for all further processing, so this information could be hidden to avoid confusion. In fact, even if another message type is declared later on, it will not be used.

There is one transition in each behavior graph which has a special meaning. It is the return from the handle incoming message object to the wait for messages object. In fact, this is “syntactic sugar” as this transition is self-subsistent for the code generation. Within the code generator, this transition is made implicitly so this transition carries no extra information. Despite everything, it leads to more understanding of the behavior of a system. If this transition would not be allowed, there would be no explicit transition back to the source state wait for messages object, if the message type is not handled in the target state.

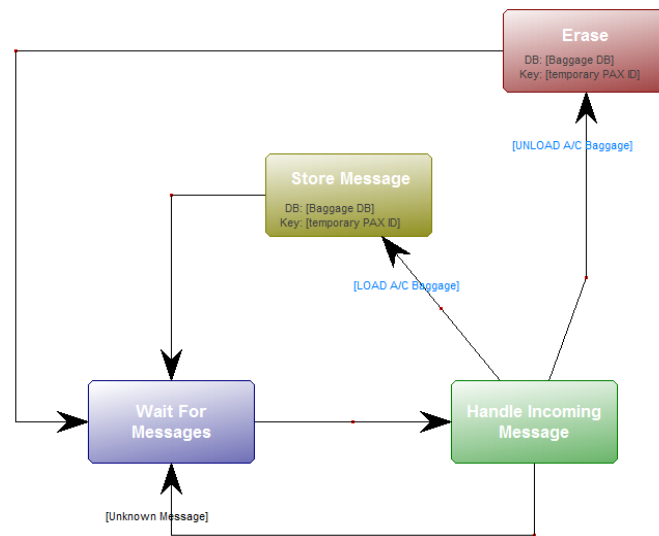


Figure 6.63: Behavior Example

### 6.9.11 Behavior Checks

Eleven checks are implemented for the objects and transitions in this graph. Firstly, each object needs an incoming and outgoing transition and both are not allowed to point to the same object. Furthermore, there must be exactly one wait for messages object and exactly one handle incoming message object.

Exactly one transition from wait for messages object to handle incoming message

object and exactly one transition from handle incoming message object to wait for messages object must be present.

If the erase object is used, then the database must be defined in the template graph. Also, if the store message object is used, the appropriate database for this message must be present in the template graph.

If the process update object is used, a graph with a layout for this airport must exist.

If the send message object is used, the database and the message must be defined in the template graph.

Also, all incoming messages (all outgoing transitions from the handle incoming message object) must be defined in the template graph.

## 6.10 Template Graph

The template graph is different from the other graphs. In all other graphs, objects necessary for the domain are defined and their relationships are specified. These objects are the “visible part” of the domain. They carry the basic information.

However, the template graph is a kind of “helper-class”. The template graph is used to create the structure and content of the messages, which are transferred between the systems (see chapter 5.4 *E-Cab Systems*) and are used within the whole model. Also, the databases for storing such messages are defined here.

Therefore, another level of abstraction is raised to eliminate the multiple definitions of messages or signals in each single graph.

Sender and receiver will use exactly the same data types and content. By defining all messages and their data stores in one place, all information about exchanging data is defined in one place and is not scattered within the model. Furthermore, no duplicate information must be defined in the model as the sender and the receiver will have access to the same message definition.

If the sender would define its message type in the sender’s logic part and the receiver defines its message type in the receiver’s part, there might be a chance that the messages do not fit. In the case that the messages would be defined in different graphs, the modeler must take care that the messages are equivalent. If one signal within a message must be changed, all other graphs and objects must be updated too. This manual task is error prone and should be avoided.

By using the template graph, all used messages are defined precisely in one place and no manual adaptations are necessary. The template graph holds all message structures

which are used to transfer information from one entity to another. The content of the messages should be easily to change current entries and able to adopt new pieces of data.

If a message type or content is changed, the behavior of a simulated application does not need to be adjusted as well. In most cases, the adding, deleting or renaming of an item can be fully transparent — only the name of the key must be present and fit to the associated database (see chapter 6.10.1 *Database Object*). For instance, if a passenger needs an additional property, the passenger object is updated. This new property is then added to the appropriate messages. As the simulated equipment is generated from the behavior graphs, each adjustment of the associated objects within the model and the template graph can be done without great effort.

Also, the evolution of the DSL is supported (see chapter 3.10 *Evolution of DSL*). Parts of the messages can be modified or new messages can be introduced. The behavioral variability is supported and new features can be implemented without great effort.

This technique ensures that the messages are well-formed. It minimizes the common mistakes of wrong defined messages.

The important part of this graph is also to define the relationships between messages. This means that a system receives messages from another system and reacts by sending a different kind of message. The content of this second message is based on the received message. Therefore, the identifier (in most cases) of the received message is used for the reply.

Automated checks are applied to this graph, so all messages defined here are checked against a given set of rules. For instance, by using inheritance and relationships only syntactically correct messages are defined. These checks are executed in one place and therefore do not need to be executed for each object which utilizes this message.

The template graph is not part of the scenario but it is located on the same level. It can be accessed from the graph browser in the MetaEdit+ main window.

The template graph contains the three objects database, message and data structure.

Two properties exist for this graph. Firstly, it has a name which helps to identify the graph. Furthermore, it contains a flag if live checks shall be performed. This flag is evaluated by the test bench object.



### 6.10.1 Database Object

The E-Cab Systems send and receive messages in order to exchange data. A database is a storage container for messages<sup>1</sup> as they need to be stored within the local applications. The database must fit to the incoming or outgoing messages in such a way that all needed information can be stored and accessed.

Each database has two properties (see figure 6.64 *Database Properties*). The first property is the name of the database in order to distinguish it from other databases. For accessing a specific set of data, a key is needed. This key is assigned to each database object via the key property.

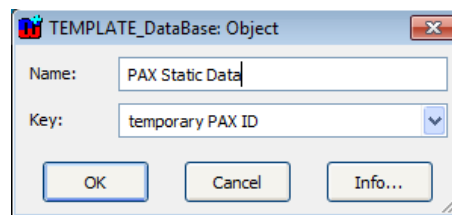


Figure 6.64: Database Properties

The database object needs to be linked to the messages which need to be stored. Also, the structure and the data types need to be stored (see chapter 6.10.3 *Data Structure Object*). Therefore, links to the data structures must be present.

The key is used to retrieve stored information. Therefore the key must be present in the attached messages.

A database can be connected to more than one message. If this is the case, the storage container is extended to hold all datatypes from all connected messages. This is used if information retrieved from different systems needs to be combined. The key must then be present in all connected messages.

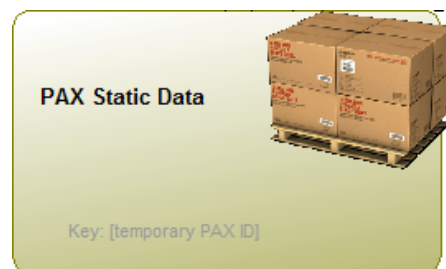


Figure 6.65: Database Object

<sup>1</sup>When the model is executed later on, messages will be transferred between different entities. Each entity needs to store the incoming and outgoing messages for further processing.

The object database itself is presented by a greenish rounded box together with an icon of a palette of boxes (see figure 6.65 *Database Object*). The name is displayed in bold dark letters in contrast to the database key property which is displayed in light grey letters.

### 6.10.2 Message Object

Interacting systems depend on the exchanged information. This information is coupled together into messages which are sent and received by these systems.

In order to establish a well-formed communication, this information needs to be agreed by all participants and therefore structured in the same manner. A message object couples a set of information into a compound which is understood by all peers. As different messages transport different kinds of information, a message type is assigned to each message. The content of a message is defined by associated data structures.

In this DSL, SOAP is used for exchanging information between the entities. Therefore, the message content must be defined as a SOAP-XML message. Each message content consists of an XML envelope and the XML body where the information of the message is defined.

The body consists of attached data structures. These data structures are mapped onto an empty message by creating a relationship between these objects. The indentation of the content of the message is created automatically. It is derived from the attached data structures (nesting is done here).

Each message has only one property which defines the type of the message (see figure 6.66 *Message Properties*).

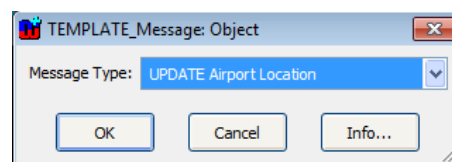


Figure 6.66: Message Properties

A message has two types of connections. One connection must be made to the data store (see chapter 6.10.1 *Database Object*) and another one to the associated data structures (see chapter 6.10.3 *Data Structure Object*).

In figure 6.67 *Message Object* an example for a message is given. “UPDATE Airport Location” is the name of the message. Within the message body several data

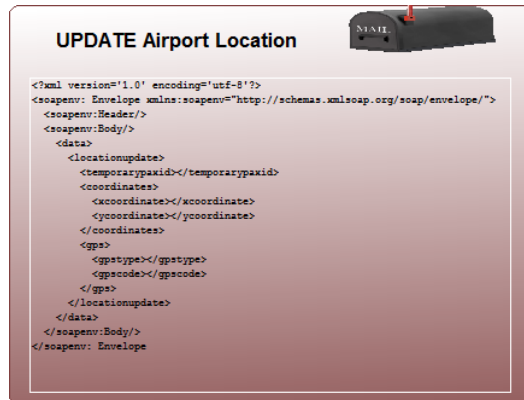


Figure 6.67: Message Object

structures can be found. In this case the “temporary PAX ID”, coordinates and location information.

A message object is presented by a brownish rounded box (see figure 6.67 *Message Object*). The type of the message is displayed in black letters in the top left corner. In the upper right corner, an icon of a letter box is displayed. If data structures are attached or removed, the payload is updated automatically to reflect the content of the message. The modeler can see the payload immediately.

### 6.10.3 Data Structure Object

Data structures are used to straighten out and pool information which is sent via messages. Parts of the information belong to each other and should be joined together. By using data structures, hierarchies of data can be expressed.

A data structure is a collection of atomic datatypes. This collection can be as large or small as needed. Data structures are introduced to create partitions within messages. In this case it can contain the following datatypes<sup>2</sup>:

- Data types (data types can be inherited by other data types)
- String
- Boolean
- Integer Types

<sup>2</sup>As these types are mapped to code by a generator in further steps, all kind of datatypes could be used here. In case of this DSL, no other types are used. To keep the DSL as small as possible, only used datatypes can be defined here.

Data structures can be nested to archive a logic coherence. This is simply done by creating a relationship between data structures.

For instance, a passenger object (see chapter 6.3.1 *Passenger Object*) has got an address and contact information. Both parts are subdivided into single sub-parts like a street name or city code in case of the address compound or in case of the contact part: an email address or phone number. This hierarchy can be expressed by data structures.

A data structure has got a name property which identifies the structure. Also, up to ten additional properties can be added. Each represents a type definition within the structure. For instance, a passenger “basic data” object consists of a title, gender, a name and so on. Also, each of these passenger properties need to have a type. This is needed for the simulated systems in order to create outgoing and disassemble incoming messages.

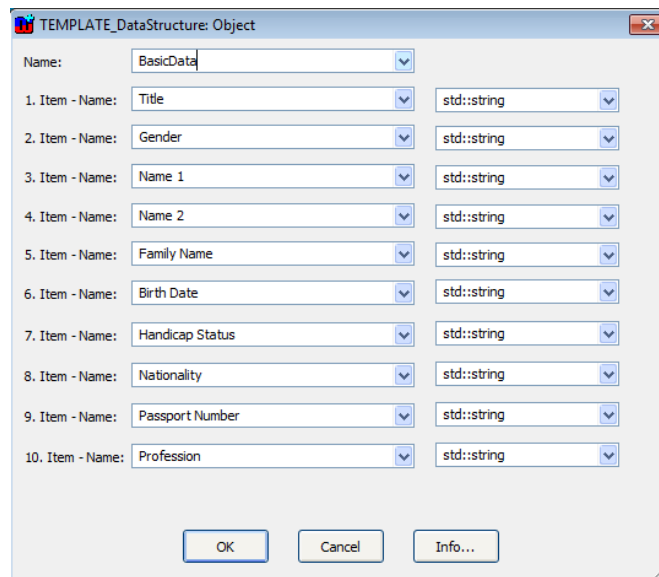


Figure 6.68: Data Structure Properties

A data structure object can have connections to other data structures in order to create hierarchies. Also, the data structures need to be attached to message objects (as the structure defines the content of the message). Furthermore, a connection to the database is needed (as the data structure defines the types for each structure entry which is needed for the definition of the database).

The data structure object is represented by a grey box. The keyword “data structure” and the name of it is displayed in the upper left corner of the object. Beneath the name, all defined properties and their types are displayed.

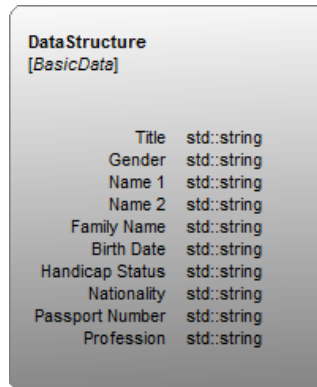


Figure 6.69: Data Structure Object

#### 6.10.4 Template Connection Types

Two different connection types are used in this graph:

- Inheritance Target
- DataBase Message Connection

InheritanceTarget is used to connect data structures to other data structures (although, recursion is not allowed) or to link data structures to messages. Nesting of data structures is achieved by connecting a data type to another data type. The same connection is also used to bind the structures to the database.

It is represented by a dotted line with an arrow.

The second connection type is used to connect the message to the database. It is represented by a solid line. This type also examines if all necessary information is specified.

If the database does not match the connected message, a red error warning is shown. If the key of the database can be found within the connected message, a green OK is displayed.

#### 6.10.5 Template Checks

Six different checks are implemented for the objects in this graph. Five of them are related to messages and one is responsible for well-formed data structures.

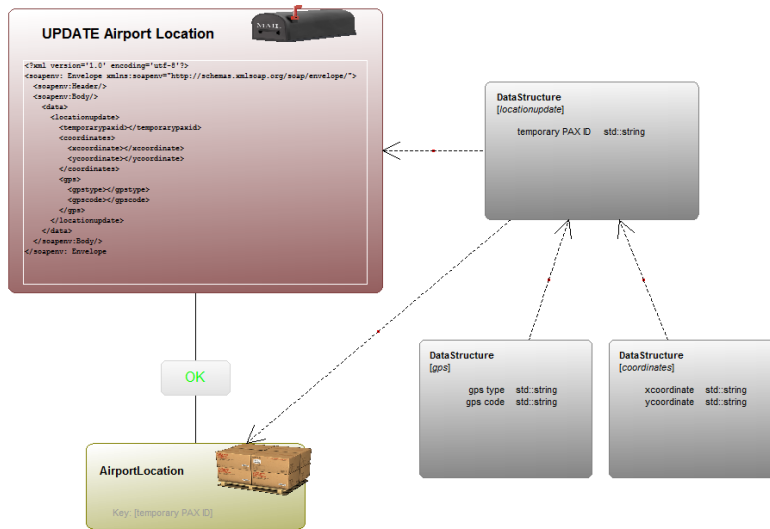


Figure 6.70: Template Connection Types



Figure 6.71: Template Connection Error

Every data structure object must have at least one entry or an attached data structure object with at least one entry. This ensures that no empty fields exist but allows creation of hierarchies of data structures.

Every message used in a graph must be defined here. It is not allowed to use a message type in a behavior graph which is not defined in the template graph. This ensures that sender and receiver have the same interface. It is also needed for the creation of simulated equipment (code generation).

Every message must be unique. No message can be defined more than once. This ensures that there is no mismatch.

Every message must have at least one data structure attached. No empty messages are allowed.

Every attached data structure must have a unique name. There can be several data structures with the same name — but only one of these structures can be attached to one message (although other structures with the same name can be attached to other messages).

Every message with an attached data base must have an attached structure containing the data base key. As the key is needed to identify a specific message, the key must

be present in the data base and the message. One of the attached data structures must therefore contain this key.

## 6.11 All Domain Objects

Two objects are not specific to a domain but can be used universally. The note object is a small text box which has no influence on the model but helps the understanding of the model.

The second object is the test bench configuration object. This object runs generator scripts and checks if the current graph is well-formed. This is an important prerequisite for generating the target.

### 6.11.1 Note Object

The note object is used to attach small text portions to an object or transition. It is not evaluated by generators and has no influence on the target or test. It is used as an additional description.

The object has one property, which is the displayed text.

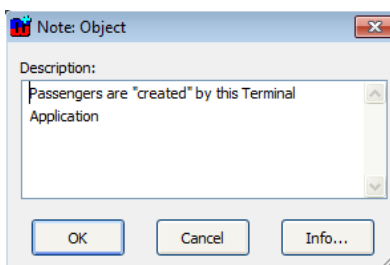


Figure 6.72: Note Object Properties

The note is attached to another object by drawing a connection between both instances. This is depicted as a dotted line.

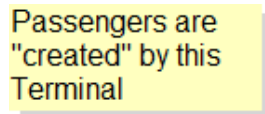


Figure 6.73: Note Object

The object is represented by a yellow box.

### 6.11.2 Test Bench Configuration Status Object

In order to ensure a well-formed graph, the syntax and semantics of each graph need to be correct (see chapter 3.11 *Syntax and Semantics*). The purpose of this object is to help the modeler by evaluating the current graph and display hints that issues are present.

The object evaluates the flag of the graph if live checks should be performed. If this is the case, the graph is constantly audited (each time a change is made) and the syntax and semantic rules are processed all the time.

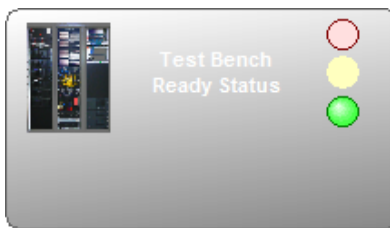


Figure 6.74: Test Bench Configuration Status

The current state is indicated by a traffic light analogy. If the yellow light is switched on, no live checks are performed. The red light indicates an error on the graph while a green light is a sign of a well-formed graph. Additionally, if an error is detected, a message is displayed which helps the modeler to correct the issue.

The test bench object has no properties. Also, no connections to other objects are necessary. It is placed onto the graph and can be removed — if wanted — when the model is completed.

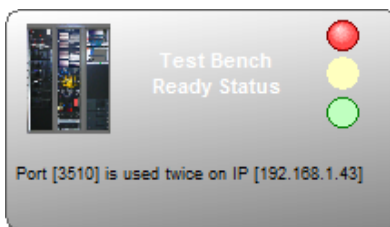


Figure 6.75: Test Bench Configuration Status - With Error Message

The object executes the local generator script `_CHECK_TBConfStatus` which might run one or several more subscripts (see chapter 6.12.4 *Check Model*). The outcome of these generator executions result in the overall status for this graph.

As the only hook is the local generator script, the same object can be placed on various graphs. The object itself does not need to be adjusted for another



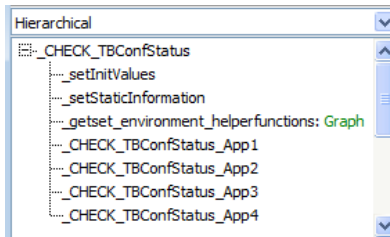


Figure 6.76: Test Bench Generator Hierarchy

graph. By specializing the generator, all kinds of checks can be performed (see chapter 6.12.4.1 *Test Bench Configuration Status Object Check Generators*).

The object is represented by a greyish rounded box. A test bench icon is displayed in the upper left corner. Next to it, the name of the object is shown. The traffic lights are located in the upper right corner. The area for error messages is located in the lower part of the object.

### 6.11.3 Graph Information Object

The graph information object is used to display the name, date and current version of the model and meta-model (see chapter 3.12.5 *Achieving Versioning*). The appropriate files from the repository system are parsed and the content is displayed.

This object might not be used all the time and can be removed from the graph whenever wanted. It is not necessary for the creation of files in the sense that the produced code differs if this object is present or not. However, the information is used in the created files as the repository information is included at least in reports.

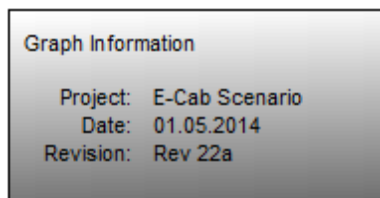


Figure 6.77: Graph Information Object

No properties are necessary for this object as internal data and files from the repository system are parsed and displayed. The object is represented by a grey box containing the project's name, date and the revision of the model and meta-model.

## 6.12 Generators

Generators are written using the built-in scripting language of MetaEdit+ which is called “MetaEdit+ Reporting Language” (MERL, see chapter 13.1.4 *MERL Reference*).

These generator scripts can be used for different kinds of tasks. In case of this work, four different tasks have been identified which are covered in the next paragraphs.

Basically, a generator is a script that was written using the built language from MetaEdit+. The generators are executed within the tool. This could be done automatically (objects can trigger these executions) or the modeler manually executes such a script by clicking a button. The model information is extracted and the input formats for the attached tools of the tool chain are created.

These generators are built into the tool and belong to the “generator” layer (see figure 4.1 *Generic Workflow*). Although the term “generator” is used in the chapter 8 *Test Data Generation*, all mentioned generators there belong to the layer “framework” (see 4.1 *Generic Workflow*). Both terms are used and should not be mixed up: all generators which are built into the modeling tool are part of the “generator” layer, while all other generators are part of the “framework” layer.

### 6.12.1 MERL

The MetaEdit+ Reporting Language, MERL for short, is built into the MetaEdit+ DSL Modeling Tool. The language is designed to traverse the model and extract the stored information from it. However, only reading of values of properties is permitted. It is not possible to set information with MERL like writing new values into a property of an object. If values need to be stored, local variables can be used.

MERL provides methods to loop through graphs, objects or properties of an object like “do” or “foreach”. Conditions can be expressed by using if-clauses and boolean operators, like “not”, “and”, “or”. Several properties can be derived by defined calls. For instance, “oid” returns the internal ID for this graph or object. Also, the coordinates can be read by using the appropriate method functions. The whole function set can be found in chapter 13.1.4 *MERL Reference*.

### 6.12.2 Types of Generators

The generator scripts can be divided into four main tasks which are discussed in the next chapters in detail.

**GUI Enhancements** The concrete syntax is enhanced in a way that status information of the model is displayed in a graphical representation.

**Check Model** Generator scripts can check the model if it is syntactically and semantically correct.

**Model2Text** The model is traversed and code or test data is generated.

**Trigger external actions** Tests can be executed by a trigger signal.

### 6.12.3 GUI Enhancements

One class of generators is used for the enhancement of the graphical DSL. For instance, the graphical representation of an object can be extended so that different kind of information can be presented. This is done by displaying additional information or by switching symbols within the representation. Also, a reduction of visible elements of an object is possible. For instance, redundant information can be hidden.

These views are useful for several cases (see chapter 3.6.2 *Views*). Three viewing concepts can be established: overview, zooming and filtering, and details on demand.

The visualization can be changed if a property of an object is changed. The change of one property could change the semantic meaning and therefore the object should be distinguishable between similar objects. For instance, if the property “AirportLocationType” of an airport location object (see chapter 6.7.5 *Airport Location Object*) is changed, the color and icon for this object is changed accordingly. In case of the message objects in the template graph, the message body dynamically changes if links to data structures are added or removed (see chapter 6.10.2 *Message Object*). A live preview of the message is generated instead of a generic and static view.

More information can be displayed for an object. For instance, within the design phase, the coordinates of an airport area (see chapter 6.7.8 *Airport Area Object*) can be displayed to help the designer place the objects on the graph. The additional information can be switched on and off by setting a global flag for the graph (see chapter 6.7 *Airport Layout Graph*).

Another part is to display error information on the test bench configuration status object (see chapter 6.11.2 *Test Bench Configuration Status Object*). Here, the traffic lights show a quick status of the state of the actual graph. Also, more detailed error information is displayed, for instance, the usage of the same port and IP address within an airport network (see chapter 6.8.1 *System Deployment and Configuration Object*) or if behavior objects within the behavior graph (see chapter 6.9 *Behavior Graph*) are not properly connected.

This technique is also used to inform the modeler if attached values are not defined as specified by the DSL. For instance, within the graph airport layout (see chapter 6.7.11 *Airport Layout Checks* and figure 6.33 *Graph Airport Layout*) the sum of the probabilities must equal “1” for all outgoing transitions from one waypoint. If this is the case, all attached values are displayed in green. If this is not the case, the color red is used to display the values.

Also, redundant information can be hidden. Within the behavior graphs, the name of the message is only displayed in the first transition from the handle incoming message object to the next behavior object (see figure 6.63 *Behavior Example*). All other transitions do not display the value for the property “message type”. This keeps the diagram as small as possible which makes it more readable and understandable.

Each graph has an attribute called “show properties”. This is used to create two different views on the graphs. If no properties are shown, the designer view is active. Objects are more “light weight” as only the essential attributes are displayed. The detailed view is set if all properties are shown. The modeler will then see much more information about the objects within the graph.

### 6.12.3.1 Airport Layout Enhancement

One of the GUI enhancements is embedded in the airport layout (see chapter 6.7 *Airport Layout Graph*). The goal was to implement two different views in the same layout but with different aspects in mind (see chapter 3.6.2 *Views*).

The airport consists of a floor plan which is described by the borders which define that area. This floor plan is the basis for two different graphs. As it would be error prone to design the same plan twice or copy such a layout to another graph, the solution is to display only either all objects and transitions for one or the other aspect.

One view shows the airport locations like shops and restaurants and the paths between them (see chapter 6.7.5 *Airport Location Object*). The other view displays the area layout which will be used in order to partition the airport into different areas (see chapter 6.7.8 *Airport Area Object*).

The transition between both views is handled by a generator which is triggered by a button embedded in the associated airport layout graph. The generator consists of four steps: it will read the current type of view from disk (line 10 in listing 6.1 *Local Script Change View*). This information stored in an associated file for this graph (of course, other graphs have their own preference file). According to the current state (in this case either “PAX Move” for the first view or “Guidance System” for the second view), a patch file is loaded and applied to the current graph (lines 21,

38). The next step is to write the new state to disk (lines 25–28, 42–45). At the end, a helper script is executed (lines 31–32, 48–49) which simulates a F5 key press event in order to refresh the graph so that no manual intervention is needed (if this would not be executed, the graph would not display the new content and the user would need to execute the menu entry *refresh* manually).

The objects, both airport location objects and airport area objects, rely on the current state of the preference file (a generator is used as *condition* for each object). However, transitions cannot be changed in that way and they would be displayed regardless of the current view. This would lead to confusion.

In order to solve this issue, two patches have been created for this type of graph. Each patch includes a set of transitions with black and transparent lines (vice versa for each state). By applying such a patch, the wanted behavior (only one type of transition is displayed) can be achieved.

```
1 Report '!Change View'
2
3 /* create path to patch files */
4 variable 'path' write 'C:\Users\Tobias Hartmann\Workspace\MetaEdit+ 4.5\
   patches\' close
5
6 /* create filename for this graph */
7 variable 'fn' write 'AirportLayoutLastState_' :Name; '_' oid close
8
9 /* read last state from file */
10 variable 'airportLayoutLastState' write filename $fn read close
11
12 /* translator for removing the LF */
13 to '%removeLF' newline '\
14 $' endto
15
16 /* last state was PAX Move? */
17 if $airportLayoutLastState%removeLF = 'PAX Move' then
18
19 /* change state to DSS, apply patch */
20 variable 'path' append 'PatchGS\PatchGS.mxt"' close
21 internal 'fileInPatch: ' $path execute
22
23 /* write last state to file, but do not use the filename write command
24 as we do not want to see the ugly information window */
25 variable 'cmd' write 'cmd /c "echo ' close
26 variable 'cmd' append 'Guidance System' close
27 variable 'cmd' append '> '$fn'"' close
28 external $cmd execute
29
30 /* excute small script to emulate F5 key press event to refresh the
   current window */
31 variable 'cmd' write 'sendf5key.vbs' close
32 external $cmd execute
33
34 else
35
36 /* change state to PAX Move, apply patch */
37 variable 'path' append 'PatchPAX\PatchPAX.mxt"' close
38 internal 'fileInPatch: ' $path execute
```

```
39
40 /* write last state to file, but do not use the filename write command
41 as we do not want to see the ugly information window */
42 variable 'cmd' write 'cmd /c "echo ' close
43 variable 'cmd' append 'PAX Move' close
44 variable 'cmd' append '>'$fn'"' close
45 external $cmd execute
46
47 /* excute small script to emulate F5 key press event to refresh the
   current window */
48 variable 'cmd' write 'sendf5key.vbs' close
49 external $cmd execute
50 endif
51 endreport
```

Listing 6.1: Local Script Change View

```
1 Set objShell = CreateObject("WScript.Shell")
2 objShell.SendKeys "{F5}"
```

Listing 6.2: Local Script sendf5key.vbs

#### 6.12.4 Check Model

One class of generators is used for checking the model for errors or misconfigurations. In this class, two types of checkers exist. One type provides so-called “quick-checks” which display faults within a specific context. This is used by the role type `DataBaseMessageConnection` (see chapter 6.10.4 *Template Connection Types*). The other type is a specialized object which only purpose is to check the actual graph if this is well-formed. This object is called test bench configuration status object (see chapter 6.11.2 *Test Bench Configuration Status Object*).

The test bench status object is designed for the “check model” task. It can be placed onto every graph. This object instantiates and executes the local check scripts for the actual graph. The modeler is encouraged to use this object while being in the design phase.

Via a global flag for the graph, the checks could be switched on or off. If the checks are switched on all objects and relationships are evaluated on the fly and possible errors are displayed in real-time.

If the live checks speed up or slow down the design process depends on the complexity of the graph (and therefore on the power of the used computer). If too many objects are present, the checks might be quite time consuming. This is because each object on the graph is evaluated if only one item is changed, added or deleted.

There are two solutions for this issue. Either the test scripts are extended to focus only on specific parts of the graph and these tests are switched on and off by using

#### 6.12.4.1 TEST BENCH CONFIGURATION STATUS OBJECT CHECK GENERATORS

more global flags. The second solution is to enable the checks at the end of the creation of the graph. Then, possible errors are displayed when the design phase is closed and just need to be corrected.

The test bench status object uses generators for graphical enhancements to display a traffic lights symbol and enrich the graphical representation with error messages or tips to solve a problem.

#### 6.12.4.1 Test Bench Configuration Status Object Check Generators

The local generator `_CHECK_TBConfStatus` must be present within the current graph. If this generator does not exist, no checks are performed. This script is the entry point for all further checks.

```
1 Report '_CHECK_TBConfStatus'
2
3 /* setup variables / initial values */
4 subreport '_setInitValues' run
5
6 /* get static variables for this graph */
7 subreport '_setStaticInformation' run
8
9 /* include global helper functions */
10 subreport '_getset_environment_helperfunctions' run
11
12 /* Checks:
13 *   check for right count of applications configurations
14 *   check for same application name
15 *   check for same port number on same host
16 *   check if simulated system has defined behaviour
17 *
18 */
19
20 /* check for right count of applications configurations */
21 subreport '_CHECK_TBConfStatus_App1' run
22
23 /* check for same application name */
24 subreport '_CHECK_TBConfStatus_App2' run
25
26 /* check for same port number on same host */
27 subreport '_CHECK_TBConfStatus_App3' run
28
29 /* check if simulated system has defined behaviour */
30 subreport '_CHECK_TBConfStatus_App4' run
31
32 endreport
```

Listing 6.3: Local Script `_CHECK_TBConfStatus`

First of all, specifics for this graph are defined. This is accomplished by the first two sub reports. The first report will initialize all local variables.

```
1 Report '_setInitValues'
2
```

```
3 /* error code and message - "return" values */
4 $errorMessage=''
5 $errorCode=''
6
7 /* local variables */
8 $counter='0'
9 $cnt='0'
10 $correctThisError='F'
11
12 endreport
```

Listing 6.4: Local Script `__setInitValues`

The second report defines variables with static information which are only valid for this type of graph.

```
1 Report '_setStaticInformation'
2
3 /* set count of systems */
4 $maxCntOfSystems='6'
5
6 endreport
```

Listing 6.5: Local Script `__setStaticInformation`

The third sub report is valid for all graphs. Therefore, no local generator is used but MetaEdit+ will reuse the one defined on a higher level. This reuse is indicated by the name of the graph where this generator can be found. In this case the word “Graph” is displayed in green letters on the right of the name of the generator (see figure 6.76 *Test Bench Generator Hierarchy*).

This report defines little helpers which are called “translators”. For instance, unwanted characters can be replaced in a string. Also, a string containing upper and lower case characters can be transformed into a lower case string. This feature is heavily used while creating target files (see chapter 9 *Creation of the Target*). While it is recommended to use mixed case identifiers in the model, this is not wanted in the automatically generated code. Such generators are described in detail in chapter 9.1.1 *Prepare local environment* (see listing 9.1 *Local Script \_\_getset\_environment\_helperfunctions*).

The checks for this graph are described in lines 13–16 (see listing 6.6 *Local Script \_\_CHECK\_TBConfStatus\_App1*). In this case, the test bench configuration status object is placed onto the system deployment and configuration graph (see chapter 6.8 *System Deployment and Configuration Graph*). Four different checks shall be run in order to ensure a well-formed graph:

1. Check for right count of application configurations — is the right amount of configurations present on this graph?
2. Check for same application name — is one name more than once present?



3. Check for same port number on same host — is the same port number used in different configurations?
4. Check if simulated system has defined behavior — each system needs to have its behavior (see chapter 6.9 *Behavior Graph*) attached to its configuration.

```

1 Report '_CHECK_TBConfStatus_App1'
2
3 /* get count of application objects */
4 foreach .ApplicationConfiguration
5 {
6   variable 'counter' append '+1' close
7 }
8
9 /* evaluate that equation and store value in variable act */
10 variable 'cnt' write math $counter evaluate close
11
12 /* check for right count of applications configurations */
13 if $cnt <> $maxCntOfSystems num then
14
15   if $correctThisError='F' then
16     /* set error code */
17     variable 'errorCode' write 'F' close
18
19     /* set error message */
20     variable 'errorMessage' write
21       'Wrong count of systems: expected '
22       '$maxCntOfSystems
23       ' got '
24       $cnt
25     close
26
27     $correctThisError='T'
28   endif
29
30 endif
31 Endreport

```

Listing 6.6: Local Script `_CHECK_TBConfStatus_App1`

The general workflow is explained by describing the generator “Local Script `_CHECK_TBConfStatus_App1`” as this check is rather simple and the general structure of the test can be comprehended easier than in a more complex example.

The variable *counter* is initialized with the value 0 (see line 8 in listing 6.4 *Local Script `_setInitValues`*). The variable *maxCntOfSystems* is initialized with the value 6 (see line 4 in listing 6.5 *Local Script `_setStaticInformation`*). All variables are of type string.

The `foreach` section loops through all objects of type `ApplicationConfiguration` (see chapter 6.8.1 *System Deployment and Configuration Object*). For each iteration, the variable *counter* is extended by the string `+1` (line 6 in listing 6.6 *Local Script `_CHECK_TBConfStatus_App1`*). This variable should contain the string

0+1+1+1+1+1 (as there should be six applications in this graph). In line 10, this variable is evaluated mathematically and the value 6" is stored in the variable *cnt*.

In line 13, it is checked whether the counted applications (*cnt*) differs from the expected count (*maxCntOfSystems*). The keyword *num* indicates that the comparison is based on a mathematical and not a lexical interpretation of "<>".

As only one error should be displayed, the guard variable *correctThisError* is checked whether it was set to *true* before. If this was not the case, then this is the first error which is reported. The variable *errorCode* is set to *F* and a textual description of the error is assigned to the variable *errorMessage*. Afterwards, the guard variable is set to *T*. Both variables are evaluated by the test bench configuration status object. They are used to display the error message and to indicate the status by using the traffic lights (see chapter 6.12.4.2 *Test Bench Configuration Status Object Indicators*).

The second test loops through all applications and compares the name of the current object with all other applications (except with itself). If a match is found, the error message is created and the guard flag is set.

```
1 Report '_CHECK_TBConfStatus_App2'
2
3 /* check for same application name */
4 foreach .ApplicationConfiguration
5 {
6   $CheckName = :Full Name;
7   $ActualOID = oid;
8   foreach .ApplicationConfiguration
9   {
10    /* Do not check same object */
11    if $ActualOID <> oid; then
12    if $CheckName = :Full Name; then
13    if $correctThisError='F' then
14
15      /* set error code */
16      variable 'errorCode' write 'F' close
17
18      /* set error message */
19      variable 'errorMessage' write
20      'Application [' :Full Name; ']' is used twice'
21      close
22      $correctThisError='T'
23    endif
24  endif
25 endif
26 }
27 }
28
29 endreport
```

Listing 6.7: Local Script `_CHECK_TBConfStatus_App2`

The third check loops through all applications and compares the IP address and port numbers of the current object with all other applications (except with itself). If

a match is found, the error message is created and the guard flag is set.

```

1 Report '_CHECK_TBConfStatus_App3'
2
3 /* check for same port number on same host */
4 foreach .ApplicationConfiguration
5 {
6   $CheckIP   = :IP;
7   $CheckPort = :Port;
8   $ActualOID = oid;
9   foreach .ApplicationConfiguration
10  {
11    /* Do not check same object */
12    if $ActualOID <> oid; then
13      if $CheckIP = :IP; then
14        if $CheckPort = :Port; then
15          if $correctThisError='F' then
16
17            /* set error code */
18            variable 'errorCode' write 'F' close
19
20            /* set error message */
21            variable 'errorMessage' write
22              'Port [' :Port; ' ] is used twice on IP [' :IP; ']'
23            close
24            $correctThisError='T'
25          endif
26        endif
27      endif
28    endif
29  }
30 }
31
32 endreport

```

Listing 6.8: Local Script `_CHECK_TBConfStatus_App3`

The fourth check loops through all applications and checks if the application is defined as simulated equipment. Each of these applications needs to have a behavior graph attached where the logic is described. If this is not the case, no code for the simulation can be created and executed in the test environment later on. If an attached graph is not found, the error message is created and the guard flag is set. Any original equipment is disregarded and is not further analyzed by this check.

```

1 Report '_CHECK_TBConfStatus_App4'
2
3 /* check if simulated system has defined behaviour */
4 foreach .ApplicationConfiguration
5 {
6   if :Simulated System; = 'T' then
7     $cnt = '0'
8     do decompositions
9     {
10      $++cnt%null
11    }
12
13    if $cnt = '0' then
14      if $correctThisError='F' then
15        /* set error code */

```

```

16     variable 'errorCode' write 'F' close
17     /* set error message */
18     variable 'errorMessage' write
19     'Simulated System [' :Full Name; ' ] has no defined behaviour'
20     close
21     $correctThisError='T'
22     endif
23   endif
24
25 endif
26 }
27 endreport

```

Listing 6.9: Local Script `_CHECK_TBConfStatus_App4`

#### 6.12.4.2 Test Bench Configuration Status Object Indicators

In this chapter, the interaction of the outcome of the tests and the display of the results is explained in detail. There are four generator scripts which are numbered consecutively (see figure 6.78 *Test Bench Configuration Status Object - Indication Generators*).

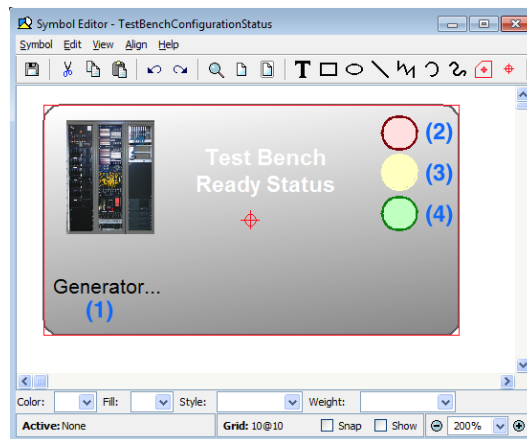


Figure 6.78: Test Bench Configuration Status Object - Indication Generators

In general, two variables are used which might have values assigned by the previously described check scripts (see chapter 6.12.4.1 *Test Bench Configuration Status Object Check Generators*).

The variable *errorMessage* contains a textual description of the error. This is used to explain the fault and guide the modeler to the associated object. Each graph possesses the property *LiveChecks*. If this property is set to *true*, all checks for this graph are executed and the results are displayed via a text object (1 - “Generator. . .”).

As this might be obstructive at early stages it can be switched off. If the model is in an advanced stage it is highly encouraged to use the checks.

If live checks should not be performed, the text “No Live Checks” is displayed to inform the user about the current status. Otherwise the script “`_CHECK_TBConfStatus`” is executed and if an error message is present, it will be displayed.

```

1 if :LiveChecks;1='F' then
2 'No Live Checks'
3 else
4 /* run checks */
5 subreport '_CHECK_TBConfStatus' run
6
7 /* display message (if any) */
8 $errorMessage
9 newline
10 endif
11 endreport

```

Listing 6.10: Local Script Generator for Label

Each of the traffic light colors (2–4) consist of two bulbs, one is illuminated and one is displayed as switched-off. Both objects are located at the same position so that only one of each color is shown. Each object evaluates the variable `errorCode` and depending on the content, the object is either displayed or hidden. By using a complementary evaluation, either the switch-on or switch-off version of the light is shown.

The configuration for displaying the red illuminated indicator (2) is displayed in figure 6.79 *Test Bench Configuration Object - Red Indicator*. A generator is attached to the condition (whether it will be displayed) of the object. If the generator evaluates to *F*, the object is shown, else it will be hidden.

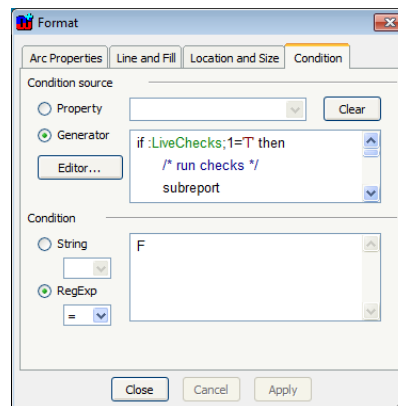


Figure 6.79: Test Bench Configuration Object - Red Indicator

The script to be executed is the same which is executed in order to display a possible error message. If live checks are enabled, the checks are executed and the variable

*errorCode* is inspected. If it evaluates to  $F$  (an error is detected), the outcome of this embedded generator is also  $F$ , else  $T$ .

```

1  if :LiveChecks;1='T' then
2  /* run checks */
3  subreport '_CHECK_TBConfStatus' run
4
5  /* evaluate return code */
6  if $errorCode = 'F' then
7  'F'
8  else
9  'T'
10 endif
11 else
12 'T'
13 endif
14 endreport

```

Listing 6.11: Local Script Red Indicator

The yellow indication (3) is much simpler as the red (or green) indicator. It only checks if the variable *LiveChecks* for this graph is set. Depending on its value, either the illuminated or the switched-off state is displayed. If live checks are disabled ( $F$ ), the switch-on indicator is shown.

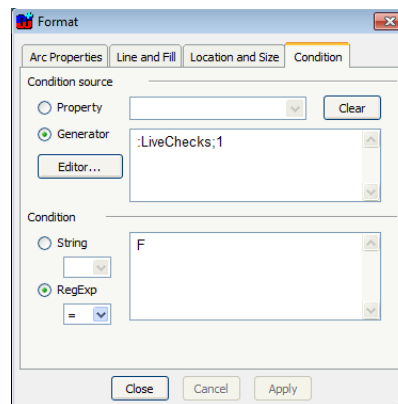


Figure 6.80: Test Bench Configuration Object - Yellow Indicator

The green indication (4) is similar to the red indication. In fact, the RegExp is set to the opposite value ( $T$  in this case) in order to create an alternating display (in this case, the switch-on state of the green indication).

Also, the attached generator is similar to the red indication generator. If the variable *errorCode* is set to  $F$  (fault is detected), the outcome of this generator is also  $F$  (and  $T$  otherwise). As the RegExp is set to match  $T$ , the indication will only be shown if no error is present.

```

1  if :LiveChecks;1='T' then
2  /* run checks */

```

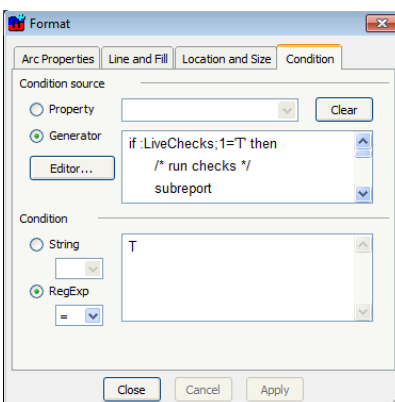


Figure 6.81: Test Bench Configuration Object - Green Indicator

```

3  subreport '_CHECK_TBConfStatus' run
4
5  /* evaluate return code */
6  if $errorCode = 'F' then
7    'F'
8  else
9    'T'
10 endif
11 else
12 'F'
13 endif
14 endreport

```

Listing 6.12: Local Script Green Indicator

### 6.12.5 Model to Text

One class of generators is used for generating model to text translations. This means that the information stored in the graphs are extracted and written into different kinds of files. This class can be subcategorized into the three following subtasks:

**Create target code** The actual code is produced for the target system. In this case, annotated C code is produced (enriched by the RT-Tester Language).

**Create intermediate format** The tool-chain consists of several tools. Each of these tools has its own specific input format which is created from the model. In this case, it is used for test data generation.

**Create configuration** The configuration of the tests and the test bench is also produced from the model. This includes the partitioning of the abstract machines to the CPU cores of the test bench. Also, the configuration for the interface cards are prepared (for instance, the AFDX configuration).

### 6.12.5.1 Create target code

For creating the target code, the model is parsed and several types of files are produced:

- Simulations
- Signal Definitions
- Interface modules (IFM)
- Oracles

These types are explained in the following chapters.

#### 6.12.5.1.1 Simulations

If simulations are specified within the model, they are created as configured there. All needed information is stored in the model, like behavior (see chapter 6.9 *Behavior Graph*) or probabilities (see chapter 6.7 *Airport Layout Graph*) and the configuration, like IP address (see chapter 6.8 *System Deployment and Configuration Graph*).

Two types of simulations are present in the model. First, there are static simulations which are built from the behavior graphs. These systems are static in the sense, that only the corresponding C code is produced. For instance, the check-in service receives two message types and reacts like described in the behavior graph (reactive system).

In contrast, there are simulations which can be regarded as dynamic as not only the code is generated. In addition, input values are extracted and calculated. For instance, if passengers are simulated, their paths through the airport are created from the model.

#### 6.12.5.1.2 Signal Definitions

A signal is an event which is transferred between systems or processes. By this definition, there are two types of signals. Firstly, messages which are transferred between systems are defined in the template graph (SOAP messages for instance). Such a message contains — at least — one or more items. Each item can be regarded as a signal. The message itself is a container for several signals which belong to each other.



Secondly, there are implicit signals which are needed for handling information within the test bed (RT-Tester signals). For instance, within the test, properties of each passenger object must be passed to the check-in application at the beginning. This transfer must not be defined explicitly in the template graph as this information can be extracted automatically. This generation of these type of signals is done during the creation of the target (see chapter 9 *Creation of the Target*).

### 6.12.5.1.3 Interface Module

IFMs are used to provide an interface between a specified function or interface and the test bed. The interface modules are static test bed functions which enrich the communication possibilities.

For each hardware interface of the test bench, a specific IFM exists. For instance, the interface module for AFDX handles all incoming and outgoing AFDX data, another interface module is responsible for sending and retrieving CAN messages. One of the IFM sends and receives push notifications (e.g. SMS) through a mobile carrier network.

In this context, there is also an interface between the test bed and the model. This IFM handles the communication to and from the model. Information like the position of a passenger in the airport layout graph is transferred from the currently running simulation to the model where the icon is displayed.

### 6.12.5.1.4 Oracle

Each test needs a specification to determine the final test result. If the outcome of the test meets the specification, the final test result is pass and failed otherwise.

Therefore, the oracles represent parts of the specification which are necessary to conclude this result. Similar to the generation of simulations, the necessary information to create oracles is already included in the model and needs to be extracted.

In order to check against the specification, the oracle needs to record events and messages. Therefore, the oracle entity is included in the target test bed.

### 6.12.5.2 Create intermediate format

The model is parsed and the necessary information for generating test data is extracted. This data is then written to a file, which is formatted for the next tool in the workflow. In this case, it's the generator for test data which reads the created

output and writes the calculated test data into a file. The probabilities from the airport layout (see chapter 6.7 *Airport Layout Graph*) are collected and rehashed for the next tool.

Additional external tools can be integrated by creating another intermediate format. Then, two files would be created, each as an input file for the next steps of the tool-chain. This way, even closed systems, with no influence on the structure of the input format for the modeler, can be integrated.

### 6.12.5.3 Create configuration

The configuration of the test bed is done implicitly through the generation of the test bed code. The necessary information is already stored within the model.

For instance, defined AFDX messages in the model lead to an AFDX configuration which needs to be used in order to configure the AFDX card. In most cases, the driver of the card accepts only messages from the network which are known to it. Therefore, the card needs to be instantiated with all defined messages before running the tests.

Furthermore, if IP ports are defined, the test bench needs to open and listen to these ports. This is also a part of the configuration which is derived from the model.

### 6.12.6 Trigger external actions

One class of generators is used to trigger external actions. Some tasks like copying files to the test bench or executing tests can be triggered through the modeling software, in this case MetaEdit+. Here, three different generators are executed to perform these tasks. These generators are explained in detail in chapter 9.1.6 *Create Target on Test Bench*.

The first generator will transfer the generated files to the final destination on the test bench. This deployment will take place when the export of the model is activated and there are no known issues within the model.

The second generator initiates the generation of the test data on the test bench. The associated shell script was created from the model and transferred to the test bench by the first generator. When executed, it parses the intermediate input files and creates the test data which are necessary for the third generator.

When the deployment has taken place, the third generator calls the compiler scripts on the test bench. Here, the intermediate files are read and the test data are calculated. These new created files must also be deployed on the test bench. This

takes place after the successful generation. Another part is to compile the created simulations, oracles, interface modules and test data into a test executable. This is the last step before the test execution.



---

---

## CHAPTER 7

---

# Testing Scenario

Within the testing activities, scenarios [31] have been implemented, which cover all needed requirements. At this stage, the main graph “Scenario” has been created and the needed sub-graphs are filled with objects and their relations. The layouts are defined and the guidance behavior is set up according to the specific airport. One click on the generator button will now create all necessary files, copy them to the right location on the test bench where the necessary tools will be executed (see chapter 9 *Creation of the Target*).

The following chapters describe the setup for a specific test. This includes all entities of the model (graphs, objects and their relationships) and their configuration. This scenario covers one part of the overall scenario which is described in chapter 5.3.1 *Test Scenario*. In this setting, some team members use the guidance system in order to find their way through the airport perimeter to their plane (see chapter 5.4.5 *Position Application* and 5.4.3 *Notification System*). Passengers check-in into the necessary systems, walk around the airport and receive messages in case they are running late. Some of these passengers are simulated and their movement is calculated by a test data generator, while other passenger’s movement is not automated and manual intervention is performed.

Within the following chapters, the setup of the scenario will be explained step-by-step and the view upon it will be widened with each further section.

### 7.1 Testing Scenario - Location Service and Guidance System

Generally speaking, the following activities will happen in this scenario. Passengers arrive at the airport and register themselves to the airport systems by using the check-in. During this process, they decide that they will use the guidance function which is present at this airport. They also check-in their baggage which will be transferred to the aircraft. Afterwards, the passengers will proceed to the gate in order to board the aircraft. While they walk to the gate, they visit several localities like shops or restaurants. If one of them is about to be running late, a text message is sent to his device. They will be unregistered from all airport systems if they either are on-board in time or if they cannot reach their flight anymore.

Four passengers are defined within the terminal graph (see chapter 6.3 *Terminal Domain Graph*). In this case, two passengers have been defined as simulated ones (Mrs. Jolie and Mr. Lennon). While Mrs. Jolie will reach her flight in time, Mr. Lennon will be too late to catch his plane. As they are simulated, movement data will be generated for both of them. In contrast, the two other passengers (Mr. Bond and Mr. Mustermann) are defined but not marked as simulated, meaning that the passengers will be checked-in but will not move around the airport perimeter automatically. All passengers, except Mr. Mustermann, opted in for guidance help on the airport perimeter.

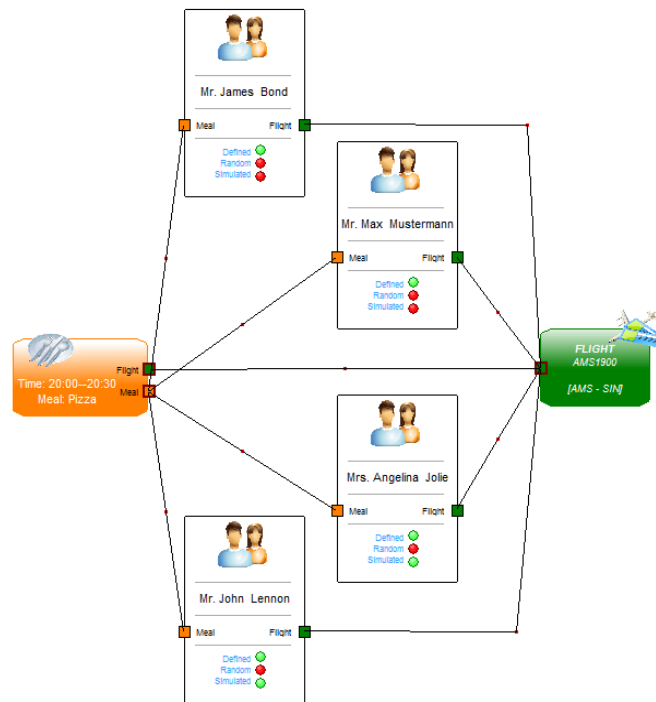


Figure 7.1: Test Scenario Passenger Definition

The airport Amsterdam is the only airport in the scenario (see chapter 6.2 *Main Scenario Graph*) which has an airport layout attached (see figure 7.2 *Test Scenario Airport Layout and Areas*). In order to show up on the airport layout, each passenger needs to be checked in at the airport systems. If this is accomplished, a pin with the name of the passenger (see chapter 6.7.7 *Passenger Move Object*) is placed next to the start point on the graph.



### 7.3 Simulated Original Equipment

The system configuration for the simulated systems consists of two parts, the behavior graphs and their location where they are executed. As they are all reactive systems, they wait for incoming messages and initiate a response to it.

The setup for each system is configured by system deployment and configuration objects (see chapter 6.8 *System Deployment and Configuration Graph*). In this test run, all systems are simulated<sup>1</sup> and are running on the test bench. Therefore, the simulated system flag is set to *true*. The IP addresses of the test bench are used for all systems and different port numbers are specified (see chapter 7.6 *Deployment*). In this case, one node of the test bench is sufficient to handle the amount of data for the scenario provided.

A behavior graph is attached for each system in order to specify the response to incoming messages (see chapter 13.2 *SOAP Message Examples*). This is depicted in the following sections.

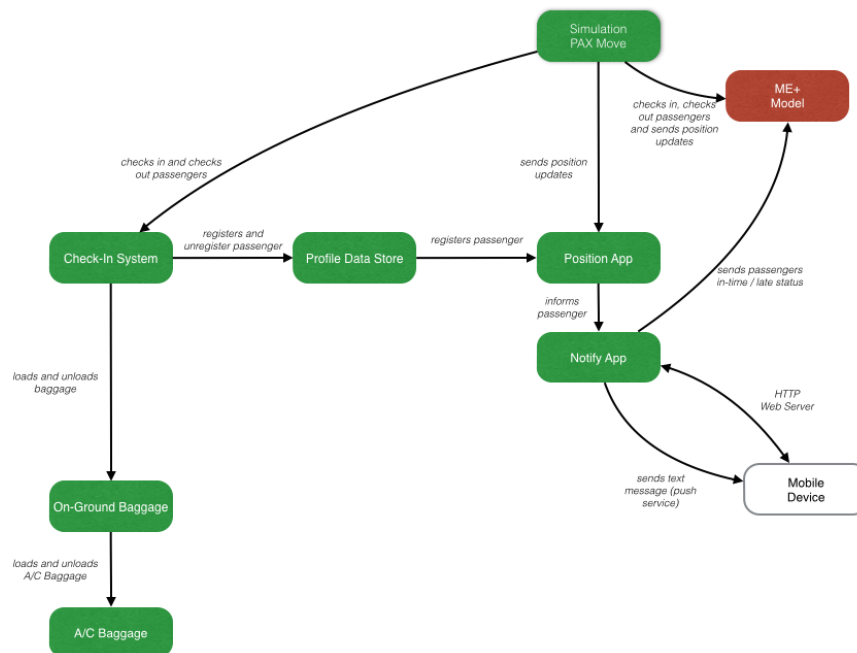


Figure 7.3: Scenario Overview

All systems communicate on a messaging basis. As these messages should only contain the information which are necessary for each system to work, each system

<sup>1</sup>However, some or all of them could be present as original systems without altering the flow of messages.



needs to store this data locally in a database. This database is the source for all further computing of data. The messages should be designed that the sending of data is on a need-to-know basis and only relevant data is transmitted. Temporary keys are used in order to provide privacy.

The initial check-in message is sent from the passenger movement simulation to the check-in system which will initiate the loading of the baggage to the aircraft and also send a register message to the profile data store (see figure 7.3 *Scenario Overview*). The profile data store will then send another register message to the position application which is responsible for the guidance function. Also, the check-in command is sent to the MetaEdit+ model so that the mentioned pin is displayed.

The simulation will run the preprocessed steps for the simulated passenger. This means that a position update message is sent for each event from the simulation to the position application. In addition, a second signal containing this information is sent to the model so that the pin is placed accordingly. In case of the manual placed passenger, the Position Updater Tool is used which runs the exact same steps as the simulation and sends two messages to the position application and the model.

The position application calculates the late status of each passenger and — if necessary — sends a notification to the passenger’s mobile device. Another message is sent to the model so that the color of the pin of the associated passenger can be changed accordingly (e.g. “green” if passenger is in time and “red” if passenger is too late). Also, the passenger can update the own guidance setting by visiting the E-Cab website and alter or confirm the preferences.

The overview of this flow of information is depicted in figure 7.3 *Scenario Overview*.

### 7.3.1 Behavior - Profile Data Store

The profile data store application receives two types of messages. If a register message is detected, the content of the message is stored in a local database. Furthermore, this event is reported to the register position application. It will also receive a register message.

In case an unregister message is received, the locally stored data is deleted.

### 7.3.2 Behavior - On-Ground Baggage System

The on-ground baggage system receives two different messages. If a load baggage message is received, the content is stored and the A/C baggage system is also informed about the loading of luggage.

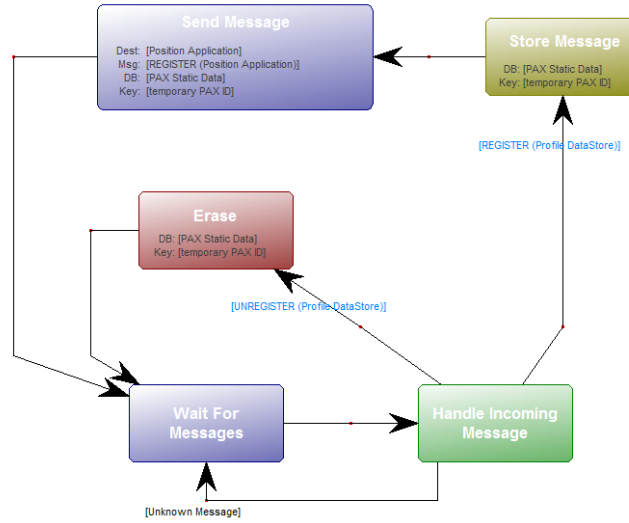


Figure 7.4: Behavior Graph - Profile Data Store

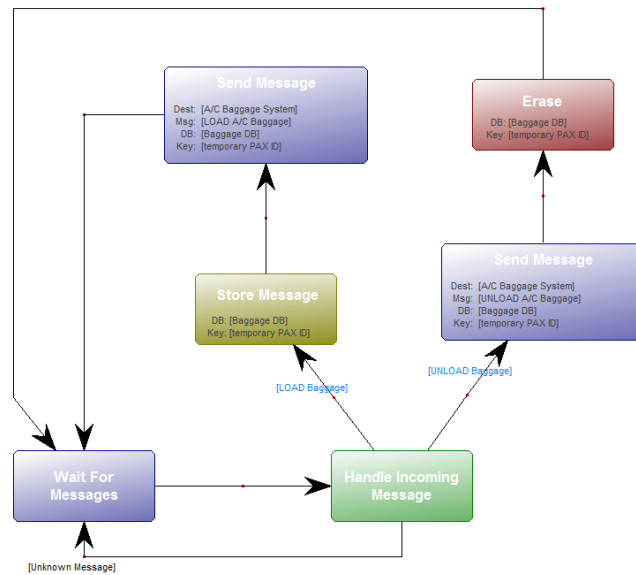


Figure 7.5: Behavior Graph - On-Ground Baggage System

If the unload baggage message is received, this information is relayed to the A/C baggage system. Afterwards, the locally stored content is deleted.

### 7.3.3 Behavior - Notification System

The notification system receives only one type of message. If this inform passenger request message is received, it is calculated if the mentioned passenger needs to be contacted. As this application is only sending out information to passengers, no local stored data is used. Therefore, no more systems need to be informed or data needs to be stored nor erased.

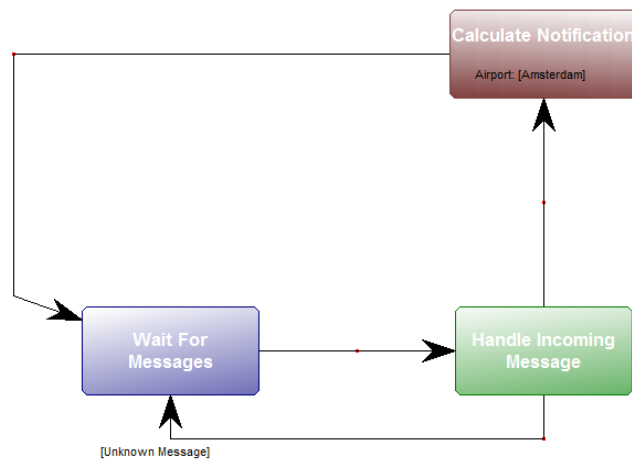


Figure 7.6: Behavior Graph - Notification System

### 7.3.4 Behavior - Check-In System

The check-in system receives either a check-in passenger message or a check-out message. If a check-in request is received, the content of the message is stored in a local database. Afterwards, two messages are sent out. The profile data store is provided with the received information. Later on, the on-ground baggage system is advised to load the luggage of this passenger.

Both systems are also noticed in case that the check-out message is received. In this case, a unregister message is sent out to the profile data store. The on-ground baggage system is informed afterwards that the luggage needs to be unloaded. If both tasks are accomplished, the locally stored data is removed from the database.

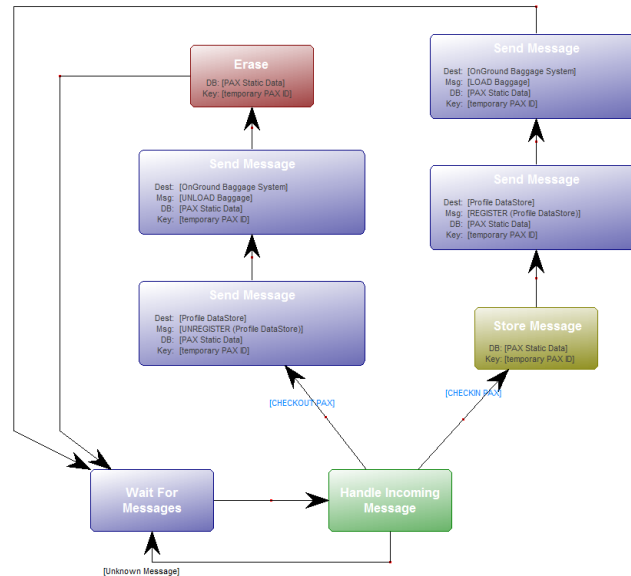


Figure 7.7: Behavior Graph - CheckIn System

### 7.3.5 Behavior - A/C Baggage System

The A/C baggage system receives messages which are either of type load or unload A/C baggage. As this is the last system regarding the sequence of baggage related messages, no further systems are involved. So in case of a load message, the data is stored in the database while it would be removed if an unload request message is received.

### 7.3.6 Behavior - Position Application

The position application needs a register message in order to store the needed data in a local database. If it receives an update airport location message, this information is processed and leads to sending a message to the notification system.

## 7.4 Position Updater Tool

The position updater tool is used to inject additional passengers movement. The tool loads a file which contains the static information of passenger. These passengers have been defined as “real”, meaning that their behavior and movement is not simulated (see chapter 6.3.1 *Passenger Object*).

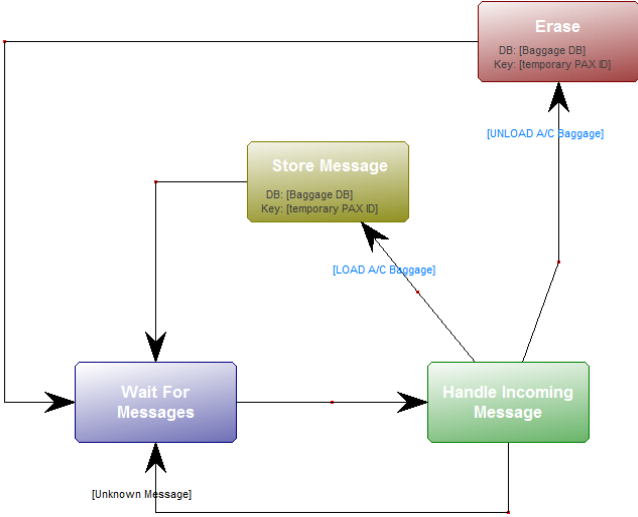


Figure 7.8: Behavior Graph - AC Baggage System

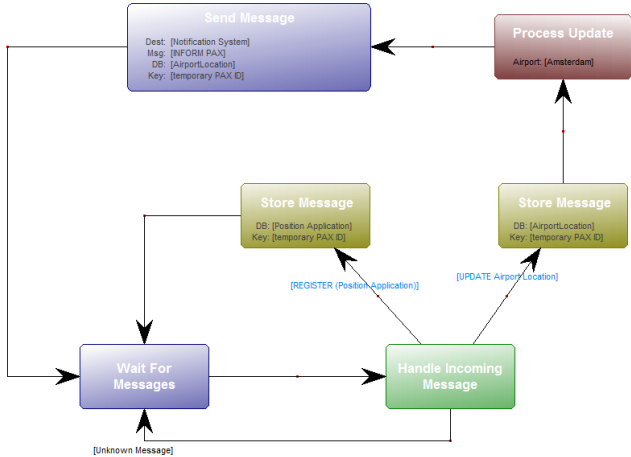


Figure 7.9: Behavior Graph - Position Application

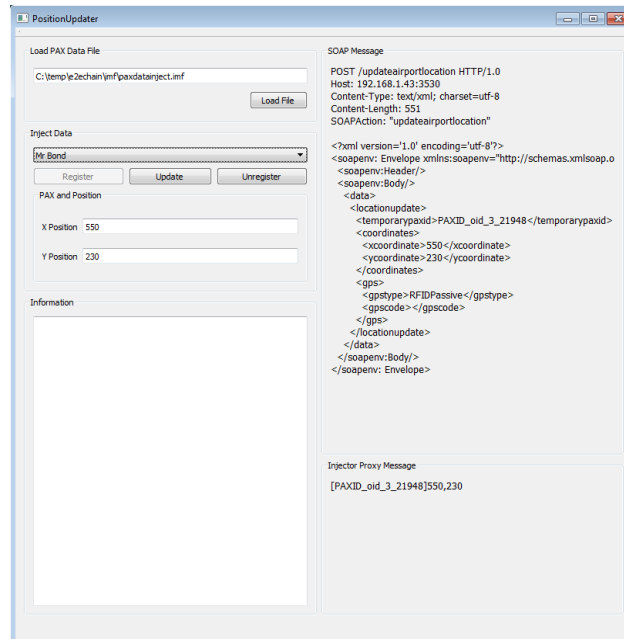


Figure 7.10: Position Injector Tool

The static information is extracted by one of the tasks to create the target (see chapter 9.1.2 *Create Intermediate Files*). If the file is loaded, the input data selection box is populated by the names of these passengers. This is a convenient way of identifying a specific passenger. If a passenger is switched, its current position data is stored and inserted when this passenger is selected again.

On the right side, the current SOAP message (like it is defined in the template graph, see chapter 6.10 *Template Graph*) and signal (see chapter 9.1.3.3 *Signal Description*) are displayed. This information will be updated each time the tester enters new values for the X- and Y-position. The connection status to the different servers is displayed on the bottom left of the window.

The tester can interact with the test and simulation by registering, unregistering or updating the location for a certain passenger. Only the buttons are enabled which can be selected to a given point of time. For instance, as the passengers have been registered by the test already, the “register” button is disabled. If the tester unregisters a passenger, the “register” button will be enabled while both buttons “update” and “unregister” will be disabled.

If a button is clicked, a connection is established to the associated server (depending on the type of message) and the message is transferred. In addition, a signal is sent to the IFM injector proxy. This signal is tunneled through a TCP connection and then relayed as an RT-signal to its destination (oracle and/or IFM MetaEdit+).

## 7.5 MetaEdit Bridge

The IFM MetaEdit+ is running as part of the test bed on the test bench, while MetaEdit+ and the model are running on a standard Windows PC. The MetaEdit+ bridge brings both parts together by relaying commands from the IFM to the model. MetaEdit+ has a built-in SOAP server which needs to be running additionally on the modeler's computer. The server can export a Web Services Description Language (WSDL) which fits exactly to the model. If the model is modified, the WSDL needs to be exported again in order to match.

The bridge itself is a java application which instantiates and incorporates the WSDL (and needs to be compiled again if the model and therefore WSDL changes). On the other hand, it features a simple clear text protocol<sup>2</sup> which is accessible via a standard TCP connection. This connection is utilized by the IFM.

When the test is started, the IFM will run an initialization procedure in order to set up the internal database of the bridge. The command `connect()` is sent to the bridge which will then connect to MetaEdit+ (all commands are explained in detail in chapter 13.1.3 *MetaEdit Bridge Commands*). Two internal databases (airports and passengers) are filled with the values from the model. This includes the names and the temporary ID so that the mapping between the running test and the associated graphical representation can be accomplished.

The next step is to register the passengers at the airport. The IFM sends the command `register()` which signs in the passenger with the given ID at the airport. The graph is refreshed immediately so that the pin is displayed instantly. If a movement event needs to be displayed, the command `setposition()` is sent to the bridge. It contains the ID and the position in order to place the pin accordingly. The command `setstatus()` is used to signal the passenger's late status. The color of the pin will then change if the new value differs from the last received one.

At the end, the bridge receives the command `unregister()` for each passenger which removes the pin from the graph.

## 7.6 Deployment

The target itself consists of several parts and therefore lots of different files need to be created. First of all, there are several entities to compile the test target binary. It consists of simulations, interface modules, oracles, and the test bed framework. The

---

<sup>2</sup>As everything is running in an isolated test lab network, security is not of a major concern here. This is why an extra layer of security was omitted. However, such a layer can be established and security can be upgraded.

test data generation framework (see chapter 8 *Test Data Generation*) produces the complete inputs for the systems under test, in this case the guidance system at the airport. Unique paths for passengers at the airport are created, according to time, path constraints and probabilities (as described in chapters 6.7.5 *Airport Location Object*, 6.7.6 *Airport Waypoint Object* and 6.7.8 *Airport Area Object*). In addition, two passengers have been defined as not simulated which means that their position will be updated manually. In order to inject additional data, extra tools are used and their data needs to be created as well (see chapter 7.4 *Position Updater Tool*). This chapter describes where these files are deployed (see also chapter 9 *Creation of the Target*).

The scenario overview shows the information flow between the simulated systems (see figure 7.3 *Scenario Overview*). This needs to be extended as not only SOAP messages are exchanged between simulations but also signals between additional Real-Time Tester Entity (RTTE) (e.g. oracles and proxies) and tools. The previous given overview is refined so that all instances are shown. For instance, the IFMs are used to convert information between different types of systems and are therefore in between logical flow of data.

The passenger movement simulation communicates via the IFM SOAP with simulations. It also relays the position information to the requirements oracle. The simulation notify application does not communicate directly with the mobile device. It sends its request to the IFM SMS which is providing the text messages to the mobile device. In this case, the service pushover has been used to deliver push notifications [59]. The simulation also sends this information to the oracle SMS. The web service is part of the IFM notify system (which is connected to the oracle and the IFM SMS).

The IFM MetaEdit+ receives commands, notifications and position updates from the passenger movement simulation, the simulation notify application and the IFM injector proxy.

The position update injector tool sends the same type of messages as the movement simulation but contacts the check-in system and the position application directly and does not utilize the IFM SOAP. This is due to the fact, that the injector tool is not part of the test bench framework and does not use signals as the movement simulation does. It also sends all signal data, encapsulated in a simple clear text notation (“signal over TCP”), to the IFM injector proxy.

This means that at least three instances are needed for the deployment: the test bench, a modeler’s PC and at least one mobile device. This breakdown is part of the next two sections. As the mobile device does only need to be equipped with a web browser and a connection to the web service it is not part of the following sections. There is no need for deployment on these devices.



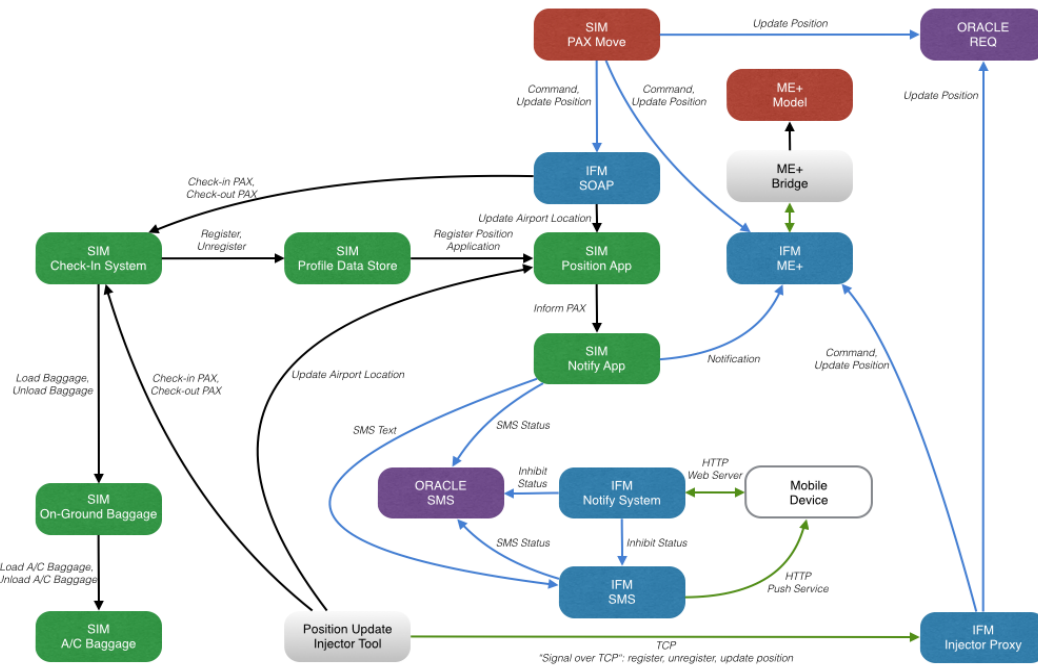


Figure 7.11: Extended Scenario Overview

## 7.6.1 Deployment Test Bench

In addition to the simulated systems, the frame work on the test bench consists of more instances. Interface modules, oracles and the passenger movement simulation are also running as part of the test bed. The involved entities consist of:

- Passenger movement simulation
- Simulations for the following systems:
  - Check-In System
  - On-Ground Baggage System
  - A/C Baggage System
  - Profile Data Store
  - Position Application
  - Notify Application
- Oracles
  - SMS

- Requirements
- Interface Modules for communication:
  - SOAP communication
  - ME+
  - Notify System (Websystem)
  - SMS
  - Injector Proxy

As the simulated systems communicate via SOAP messages, each system needs a signaling port in order to be reachable by other systems. This configuration is extracted from the model:

Table 7.1: Configuration Simulated Systems

Application	IP	Port
Profile Data Store	192.168.1.43	3500
Notification System	192.168.1.43	3510
CheckIn System	192.168.1.43	3520
Position Application	192.168.1.43	3530
OnGround Baggage System	192.168.1.43	3540
A/C Baggage System	192.168.1.43	3550

The web service is also running on the test bench (as part of the IFM Notify System).

Table 7.2: Configuration Webservice

Application	IP	Port
Websystem Notify System	192.168.1.43	8080

All other IFMs and the oracles do not need any IP configuration as they are contacted via signals (routed via Real-Time Tester (RTT) framework).

Furthermore, the project and test configurations are deployed on the test bench (see chapter 9.1.5 *Create Test Bench Configuration*). The intermediate files are also copied there (see chapter 9.1.2 *Create Intermediate Files*). As these files are input information and no RTTE, they do not need any configuration.

## 7.6.2 Deployment PC

On the modeler’s computer, MetaEdit+ is running and the API Tools must be started as well. In addition, the MetaEdit+ bridge needs to be executed in order to

relay commands from the test to the API Tool. The port of the MetaEdit+ API Tool is specified in MetaEdit+ (see figure 7.12 *MetaEdit+ API Tool Settings*).

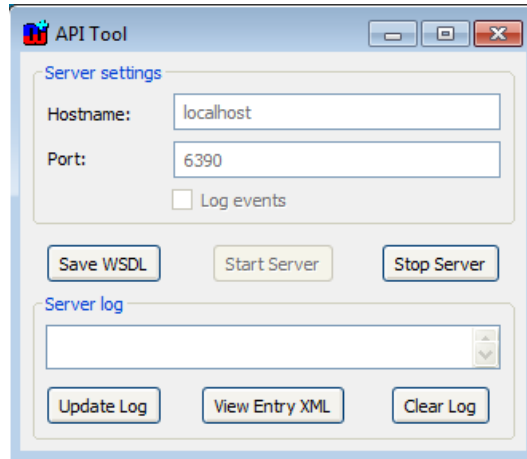


Figure 7.12: MetaEdit+ API Tool Settings

The port of the MetaEdit+ bridge is specified in the code of the bridge and is therefore fixed. The bridge needs to be run on the modeler's computer.

Table 7.3: Configuration MetaEdit+ Entities

Application	IP	Port
MetaEdit+ Proxy	192.168.1.200	7896
MetaEdit+ API Tool	192.168.1.200	6390

The position injector tool can be run on any computer which is connected to the test bench and the modeler's computer. In this scenario, it is also run on the modeler's computer. No network configuration is needed for this tool.



---

---

## CHAPTER 8

---

# Test Data Generation

In order to test systems, test stimuli are needed. In traditional projects, these test stimuli are extracted from the requirements and test procedures and test cases are written (see chapter 4.3.4 *Requirements*). Test data are coupled with test procedures and therefore cannot be regarded as single entities.

The implementer of the test procedures reads the requirements and creates normal test cases and — in most cases — a set of robustness test cases. Most of the time, the difference will lay in the test data, not the test procedure. For instance, if a range of values is given for a certain input signal, then the signal remains the same within the test procedure, but the values will be inside or outside this range.

As the requirements are written down in natural language, this is a manual task which consumes time and can be regarded as error-prone (see chapter 4.3.4.1 *Door-Status-Controller Example*). In case of changes in the requirements, all attached test procedures must be identified and adjusted. If a test procedure covers two requirements, and one of the requirements is changed, the test procedure might not be capable of covering both requirements. In this case, another test procedure must be written and the links between requirements, design and test procedures need to be adjusted (see chapter 4.3 *Development Process*).

In order to ease this situation, test procedures and test data can be extracted from the model. Regarding the workflow, the extraction of information and the creation of test data is done in two layers (see chapter 4 *Workflow*). The generator layer parses the model and stores the needed data in an Intermediate Format (IMF) while the framework layer processes these data and generates test procedures and test data.

Regarding the described testing scenario (see chapter 7 *Testing Scenario*), test data need to be generated in order to simulate passenger movement on the airport perimeter. The needed data must be gathered from the model and processed in a way so that the simulation can utilize this information.

The generation of model-based test cases, test data and test procedures is performed by an instance of the framework depicted in figure 4.1 *Generic Workflow*. The DSL model is dumped as an intermediate text representation by the second generator layer, which can be parsed by the test generator framework frontend (which is part of the third layer) and transformed into the test data representation.

The presented test data generator and its approach is tailored to this project. More sophisticated generators and approaches are described by Kyungsoo et al. [34], Peleska et al. [3], Kuhn and Gotzhein [41], and Sousa et al. [66].

## 8.1 Test Strategie

In order to check if certain paths in the model are covered, the test data must be created accordingly. As each path contains a set of objects and transitions, a list of requirements is created (the requirements are checked by the execution later on).

Each airport location object (see chapter 6.7.5 *Airport Location Object*) has ten additional properties which are used for tracing (which is needed for the final documentation, see chapter 4.3.3.2 *Documentation*) and requirements checks. First of all, two IDs (Requirement ID) are attached to each object and each transition, one for the high-level requirement (HLR) and one for the low-level requirement (LLR).

Two counters, one for each ID, will be incremented if the requirement is met during the execution (for instance, if a state is entered and/or exited or the guard of a transition was evaluated to *true*). These two counters are not a part of the property set of the object. They are applied during the execution. Generally, both would start with the initial value  $0$ . However, in some scenarios, it might be necessary to define a start value (“offset”). For instance, if it is known that a requirement is met at least  $x$  times before the test execution reaches this point, then the counters should not start with the value  $0$ . Therefore, each ID has a start value for this scenario. This property (“Counter - Start Value”) can be set for each object individually.

In order to create a path, a minimum and a maximum value (“Minimum Count” and “Maximum Count”) for each touch of an object or transition is given. This is necessary to create constraints like “this requirement should be reached at least twice but not more than six times”.

In order to introduce a “don’t care”, a flag is used which indicates if this requirement shall be checked or not. If this flag is set to *true*, the counters are evaluated during runtime and the counter constraints are taken into account. If the flag is set to *false*, the constraints are not evaluated and the minimum/maximum values are ignored. However, the internal counter is incremented each time the requirement is met.

For each high-level and low-level requirement, this set of additional properties is used:

- Requirement ID
- Counter (Start Value)

- Maximum Count
- Minimum Count
- Flag, if the requirement shall be checked

Regarding the roles of users which are used for modeling, maintaining and using the model, these properties are filled out by the modeler and the tester (see chapter 3.9 *DSL Phases*).

The modeler fills out the fields for the requirement IDs. The tester fills out the maximum and minimum counts and also the usage flag in order to create traces. Also, the start value is set by the tester. The counter if the requirement is actually met is recorded in the execution — and mapped back in the model.

## 8.2 E-Cab Workflow Test Data Generation

In this chapter, the workflow, design, and background of the test generation for passenger movement data is depicted. As all the needed information is stored in the model, it must be extracted and compiled into a target. This is done by a model to text generator (see chapter 6.12.5.2 *Create intermediate format*).

In the main scenario graph (see chapter 6.2 *Main Scenario Graph*), airport and aircraft objects (see chapter 6.2.3 *Airport Domain Object* and 6.2.5 *Aircraft Domain Object*) are present. The relationship between these two objects describes a valid flight from an origin to the destination airport. All these possible flights are collected and are present in the flight database (see chapter 6.3.3 *Flight Database Object*) which is located in the terminal domain (see chapter 6.3 *Terminal Domain Graph*). The passenger chooses one of these flights (see chapter 6.3.1 *Passenger Object*).

The creation of the test data can be subsumed in three steps:

1. Collect necessary data from the model
2. Create intermediate file
3. Call test data generator (with intermediate file as input and final test data file as output)

The first and second step is accomplished by MetaEdit+ generators (see figure 8.1 *Workflow Test Data Generation*) and are part of the workflow to create the target (see chapter 9 *Creation of the Target*). The third step is accomplished by a separate tool which uses the output of the previous steps and generates the actual test data.



Figure 8.1: Workflow Test Data Generation

The necessary information might be stored in various places in the model which is intended by all means. For instance, properties of a passenger might be used in different systems: a cell phone number in order to sent him a text to inform him about his flight or the RFID of his luggage, etc. All this information is stored in the passenger object only (see chapter 6.3.1 *Passenger Object*) — even though these bits-and-pieces could be stored in the associated systems as well. If this passenger data would be stored in various places, it would be more complicated to ensure a well-formed model. Therefore, all coherent information is stored in exactly one place.

This approach ensures a clean model and common information is gathered and well-arranged. However, this leads to the pitfall that the generator for the test data must collect the necessary information from several places in the model. Multiple objects in various graphs need to be traversed and the properties and relationships need to be exported.

Three essential parts needs to be extracted and exported from the model. First, all information about the passengers which come into question is collected. The second part consists of extracting the airport locations and waypoints. Describing the paths between these locations is done in the last part. In order to create the movement data, these tasks need to be accomplished.

Starting from the main scenario graph, all embedded airports need to be inspected if an airport layout graph is attached. If this is the case, all aircrafts and their flight routes which are attached to these airports are checked. This results in at least one entry in the flight database object on the terminal graph. All passengers which are booked on such a flight will be regarded for the test data generation — if they are marked to be simulated passengers. In this case, the necessary information about these passengers needs to be exported.

From the associated airport layout graph, all locations, waypoints and the paths between them are extracted. At the end, all this information is stored in one IMF for further processing.



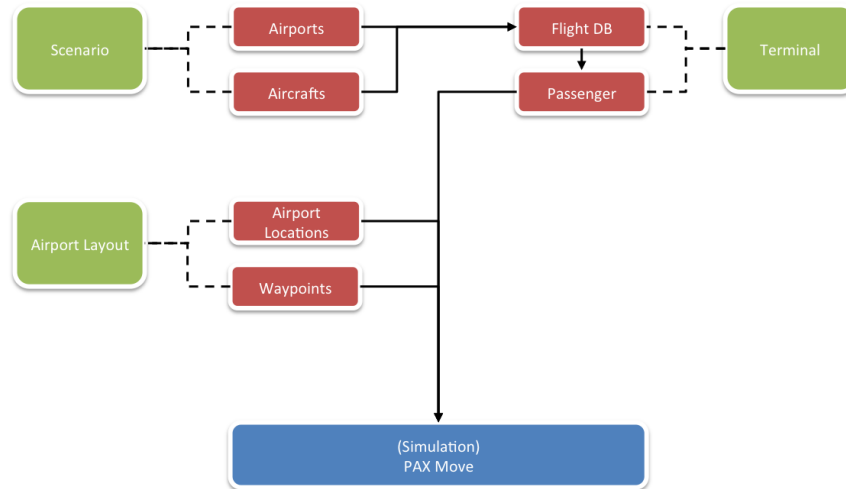


Figure 8.2: Creation of Passenger Movement Data

### 8.3 Intermediate File - Airport and Passenger Data

The MetaEdit+ generators create two Intermediate Format Files (see chapter 9.1.2 *Create Intermediate Files*) but only one of them is used for the creation of the test data. This IMF contains the condensed necessary information for describing the layout of an airport together with present passengers. In fact, the generated IMF contains information for three different purposes:

1. Location information for the creation of paths (passenger, states and paths)
2. Directives for the test data strategy (counters, flags, conditions, timing aspects)
3. Tracking of requirements (requirement ID)

The following listings are snippets of the airport intermediate format file (see chapter 13.3.4.2 *Airport Intermediate Format File*)— and therefore not complete — in order to explain the concept and the notation. This IMF itself consists of a header and footer where the content is embedded. It starts with the keyword `ecab_machine` which is followed by the name of the airport and its unique ID of the model. The content is parenthesized by curly brackets.

```

1 ecab_machine Amsterdam OID : oid3_12018 {
2   [...]
3 }
```

Listing 8.1: Snippet of Airport Export - Header

Basically, the IMF contains three different sections — as described in the chapter before — which describe passengers, locations at the airport, and paths between these locations.

Each passenger in the IMF consists of three properties. The first one is the passenger ID. This ID (`oid`) is the same ID which is used in the model. Therefore, it can be assured that each ID is unique. In case of multiple simulated passengers, the keyword `pax_multiple` is used (instead of `pax`) and the quantity is given as a parameter.

Each passenger has two additional properties. One flag indicates if this passenger will miss his flight (“late”), the second one states the departure of his flight (“departure”). The departure time is given as “seconds from midnight” (local time of the airport).

```
1 [...]
2 pax "oid3_7983";
3 late "false";
4 departure "66780";
5
6 pax "oid3_8654";
7 late "true";
8 departure "68400";
9
10 pax_multiple "4";
11 late "false";
12 departure "66780";
13 [...]
```

Listing 8.2: Snippet of Airport Export - Passenger

The airport consists of locations and paths between them (see chapter 6.7 *Airport Layout Graph*). Each location has properties like a position, name and type and also a value for a “residence probability”.

All locations (see chapter 6.7.5 *Airport Location Object*) from an airport layout (see chapter 6.7 *Airport Layout Graph*) are exported into this IMF (see listing 8.3 *Snippet of Airport Export - States*). Furthermore, both additional start- and end states are present. These two states indicate the beginning and finish of a path. As these states are different from the other ones, fixed entries for requirement ID and counters are exported as these values are not regarded during run time or the evaluation phase at the end of the execution.

Each location (or waypoint) begins with the keyword `state`, followed by its ID. Additionally, several further properties are assigned. First of all, each location needs its coordinates. In this case, the variables  $x_{1/2}$  and  $y_{1/2}$  are used to describe the upper left corner and lower right corner of the objects. Also, the minimum and maximum amount of time which a passenger will stay at this particular place is attached to each entry.

For the low- and high-level requirements, five properties are assigned per requirement. Each requirement needs an ID (LLRID/HLRID) for tracing reasons, a start counter (LLRCnt/HLRCnt), a fixed minimum and maximum of possible visits (LLRCntMin/LLRCntMax and HLRcntMin/HLRCntMax respectively) and a flag if these visits should be evaluated (LLRCntFlag/HLRCntFlag) at all.

```
1 [...]
2 startstate StartPoint  OID : oid3_16007 {
3   x1 "550"
4   y1 "230.0"
5   x2 "560"
6   y2 "240.0"
7   RESLOW "600000"
8   RESHIGH "660000"
9   LLRID "LLR_StartState"
10  LLRCnt "0"
11  LLRCntMin "0"
12  LLRCntMax "0"
13  LLRCntFlag "F"
14  HLRID "HLR_StartState"
15  HLRcnt "0"
16  HLRcntMin "0"
17  HLRcntMax "0"
18  HLRcntFlag "F"
19 }
20 endstate EndPoint  OID : oid3_30390 {
21  x1 "1440"
22  y1 "920"
23  x2 "1450"
24  y2 "930"
25  RESLOW "60000"
26  RESHIGH "120000"
27  LLRID "LLR_EndState"
28  LLRCnt "0"
29  LLRCntMin "0"
30  LLRCntMax "0"
31  LLRCntFlag "F"
32  HLRID "HLR_EndState"
33  HLRcnt "0"
34  HLRcntMin "0"
35  HLRcntMax "0"
36  HLRcntFlag "F"
37 }
38 [...]
39 state junction_oid3_16256  OID : oid3_16256 {
40  x1 "560.0"
41  y1 "565.5"
42  x2 "680.0"
43  y2 "634.5"
44  RESLOW "600000"
45  RESHIGH "900000"
46  LLRID "LLR_UpperFloor100"
47  LLRCnt "0"
48  LLRCntMin "1"
49  LLRCntMax "0"
50  LLRCntFlag "T"
51  HLRID "HLR_UpperFloor100"
52  HLRcnt "0"
53  HLRcntMin "1"
```

```
54   HLRCntMax   "0"  
55   HLRCntFlag  "T"  
56 }  
57 [...]
```

Listing 8.3: Snippet of Airport Export - States

Just like the locations, the transitions (see chapter 6.7.10 *Airport Layout Connection Types*) are also extracted from the airport layout graph (see chapter 6.7 *Airport Layout Graph*). Each transition item in the IMF starts with the keyword `transition`, followed by its ID. This ID is taken from the model so that uniqueness can be assured (see listing 8.4 *Snippet of Airport Export - Transitions*).

The property `robustness` indicates if a robustness test case should be calculated or if the calculated test data is within the specified ranges. The properties `condition` and `tolerance` are extracted from the connected source state. Both properties describe the retention time for this state. For instance, if a passenger will reside between one and twenty minutes the condition is set to 60,000 (converted to milliseconds) and the tolerance is specified as 1,200,000 (also converted to milliseconds). Actions are not used in this scenario so this property will always be empty (for this airport IMF). Of course, each transition has a start- and end point which is specified by the properties `srcstate` and `trgstate`. The property itself is aggregated of the name of the state and its ID within the model. Therefore, it is ensured that all IDs are unique. Furthermore, as the name of the state is included, it can be located in the graph as well.

The property `probability` describes the chance of a passenger walking this way. All probability values from outgoing transitions from one location must be summed up to the value 1. This value is later on used by the test data generator to create paths.

```
1  [...]  
2  transition OID : oid3_16425 {  
3    robustness "false";  
4    condition  "t >= 60000"  
5    action     ""  
6    tolerance  "120000"  
7    probability "0.5"  
8    srcstate   "StartPoint";  
9    trgstate   "junction_oid3_16238";  
10 }  
11 transition OID : oid3_16436 {  
12   robustness "false";  
13   condition  "t >= 60000"  
14   action     ""  
15   tolerance  "120000"  
16   probability "0.2"  
17   srcstate   "StartPoint";  
18   trgstate   "Quick_Check_In_Terminal_oid3_16075";  
19 }  
20 [...]
```

Listing 8.4: Snippet of Airport Export - Transitions

## 8.4 Test Data Generator Tool

The test data generator tool produces simple traces of movement data for a specific airport and passenger information. It uses the IMF which was generated from the MetaEdit+ generators (as described in the chapter before). The paths are computed based on the provided probabilities (which have been attached to the model during the modeling phase) and time information. The output of the generator tool results in a file containing position information and associated time stamps for one or more passengers.

The tool itself is called as part of the workflow during the creation of the target (see chapter 9 *Creation of the Target* and 9.1.6.1 *Helper Scripts*).

It reads the information about the passenger, airport locations and the paths between them in an internal database. Beginning from the end-point, it uses a rather simple backtracking algorithm to gain a path to the start-point. The possibility of visiting the same site a couple of times is allowed but it aborts the current generation if loops with great depth are detected. It should be avoided that paths are created where a passenger will walk to the same locations again and again.

The probabilities, which are attached to each transition, are expected to be based on the real airport layout and could be collected from observations of passenger movement. Also, the airport operator might want to influence how the passenger walks on the perimeter. For instance, due to architectonic reasons, passengers should not crowd as it would be dangerous if panic occurs. Also, in most cases, airports are segregated into areas and evacuation plans exists for these. If too many passengers are inside of such an area, it is more difficult to shift these people to a safe place.

The test data generator should therefore use these probabilities to compute traces according to this information. Nevertheless, it should also be avoided that all simulated passengers will take the same route from the start- to the end-point. The given information is used in the internal equation for computing the likelihood of a path. At the end, a path exists from the end- towards the start-point.

When all routes for all passengers have been created, a second iteration is performed. In this run, timing information is attached to each step in the path. Again, the timing conditions are used as constraints and the calculated time stamps are according to the provided data. However, as different time stamps are attached for each passenger, unique traces are generated.

Similar to the generation of the path, the second run starts from the end-point. As the time for the departing aircraft is fixed in the model, this is the reference point for all further computations. Each following time stamp is based on the last computed one.

In case the passenger is defined to be late, the time stamps are used to let him arrive at the position of the aircraft after it left already. This is accomplished by manipulating the first time stamp (departure of the flight). By adding some hours to this initial value, the passenger will not arrive in-time.

At the end, the generated data is written into an output file (see next chapter) which is used by the simulation during the test execution.

```
hartmann@testbench:~/bin> ./executeTDG
Using importFile = [ /home/hartmann/ecab//e2echain/imf/airport.imf ]
Using exportFile = [ /home/hartmann/ecab//e2echain/imf/paxmove.inc ]

Instances found:
  PAX: 2
  States: 21
  Transitions: 39

creating trace for PAX [ PAXID_oid_3_7983 ] : Done!
creating trace for PAX [ PAXID_oid_3_8654 ] : Done!
creating movement trace for PAX [ PAXID_oid_3_7983 ] : Done!
creating movement trace for PAX [ PAXID_oid_3_8654 ] : Done!
```

Figure 8.3: Test Data Generation Tool - Run Example

In this test scenario (see chapter 7 *Testing Scenario*), two passengers have been defined as simulated (with ID 3\_7983, 3\_8654). Altogether, 21 states and 39 transitions are present for this airport layout (as shown in figure 8.3 *Test Data Generation Tool - Run Example*). The computation of a trace for each passenger was done successfully in the first step. Afterwards, the timing information was added, resulting in the movement trace (step two).

## 8.5 Passenger Movement Data

The passenger movement data is the outcome of the processing of the test data generation tool. In the end, for each passenger a set of movement data is generated and time stamps are assigned accordingly. This set of data is stored in a file. It consists of a comma separated list where each line represents one movement activity.

The following listing is a snippet of the Passenger Movement Intermediate Format File (see chapter 13.3.4.1 *PAX Movement Test Data*) — and therefore not complete — in order to explain the concept and the notation.

```
1 PAXID_oid_3_7983 ,66667 ,1448 ,926
2 PAXID_oid_3_7983 ,66565 ,1160 ,853
3 PAXID_oid_3_7983 ,65614 ,900 ,891
4 [...]
5 PAXID_oid_3_7983 ,50980 ,541 ,390
6 PAXID_oid_3_7983 ,49563 ,783 ,274
7 PAXID_oid_3_7983 ,48932 ,556 ,236
```

```
8 PAXID_oid_3_8654 ,71902,1441,926
9 PAXID_oid_3_8654 ,71805,1333,904
10 PAXID_oid_3_8654 ,71199,855,958
11 [...]
12 PAXID_oid_3_8654 ,67222,569,394
13 PAXID_oid_3_8654 ,66627,323,343
14 PAXID_oid_3_8654 ,66006,554,234
```

Listing 8.5: Snippet of Passenger Movement Data

In this example, the movement for two passengers is displayed (with ID 3\_7983, 3\_8654). Each line starts with the ID of a passenger, followed by a time stamp (in milliseconds) and the position (x,y values) on the airport graph.

For each passenger, a block of movement data exists. According to the operating principle of the test data generator, a movement trace is generated for one passenger at a time, starting from the end-point. Therefore, the time stamps start with the highest value and decrease with each new entry. It is up to the simulation to order these entries (see chapter 9.1.4.3 *Test Data Stimulus*).





---

---

## CHAPTER 9

---

# Creation of the Target

In this chapter, the construction of all single parts and the creation of the overall compound is described. First of all, an overview of the target is given so that all components are clearly identified.

The workflow to create the target is initiated from within MetaEdit+ by a click on a button which is located on the main scenario in the MetaEdit+ modeler. All necessary generators are executed which will produce the necessary files, copy them to the right location, and execute further scripts in order to gain test data and to compile the test target binary. In the end, the complete target is created on the test bench. This means that no other manual intervention is necessary and “everything is in one place”.

In order to follow the principle of generating as much as possible so that manual intervention is reduced to a minimum, the complete test bed is also generated in addition to the test and test data. This means that IFMs and oracles are generated from the model as well.

For all generators which produce source code, rules resulting of the coding style guide have been followed. This includes naming conventions, indentation of source code, rules for parameters of functions, and documentation. Even the code is generated, coding smells should be avoided (see chapter 3.8 *Code and Design Smells*).

These generators are executed in a fixed chronological order. The details of this workflow are described later in this chapter (see chapter 9.1 *Workflow to Target*). All presented generators belong to the generator layer (see figure 4.1 *Generic Workflow*) — in contrast to generators which are embedded in objects of the model (see chapter 6.12.3 *GUI Enhancements*, 6.12.4 *Check Model*).

### 9.1 Workflow to Target

The workflow consists of thirteen different tasks which are depicted in figure 9.1 *E-Cab Scenario - MetaEdit+ Generator Tree*. This tree view<sup>1</sup> reflects the structure

---

<sup>1</sup>For reasons of readability, special characters are missed out. Furthermore, additional spaces are inserted. This applies for all tree views.

which is present within the model. Most of these tasks consist of several sub-tasks. However, only the top-level set of generators is presented in this chapter while the sub-tasks will be discussed later.

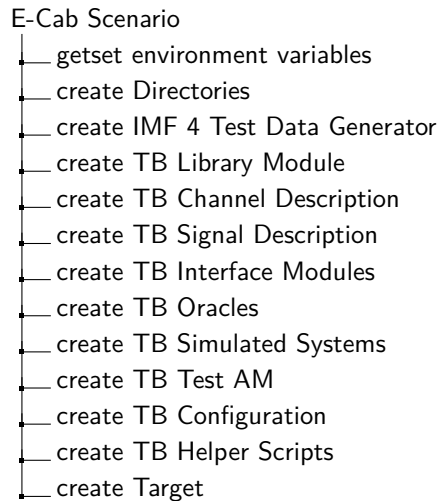


Figure 9.1: E-Cab Scenario - MetaEdit+ Generator Tree

These tasks can be categorized into six different higher level topics: the local environment needs to be prepared, intermediate files need to be generated, the test bed must be created, the test itself is extracted, the test bench configuration is assembled and — in the last step — all files (for a list of all created files see chapter 13.3.1 *Overview Created Files*) need to be copied and compiled on the test bench.

Each generator handles one specific task. This can either be a set of variables which are used later on or an implemented function. In case of such a piece of source code, the mapping from the model to the code is transparent. This means, that if such a function needs to be re-written, exchanged or extended, only this generator needs to be altered while all other generators do not need to be revised. In order to handle potentially many generators, they are partitioned into the above mentioned topics. Although this results in a possibly large tree of generators, locating a specific task is still possible by limiting each branch to a maximum of five sub-generators. Although, if recursive calls of generators are present, the depth of a maximum of five generators can be exceeded. As MetaEdit+ is unable to calculate the maximum of calls (as this depends on the graphs and objects), an “infinite” number of calls is displayed. As the name of the called generator is exactly the same name of the calling generator, this can be noticed immediately and is therefore not an issue.

These topics including all associated generators are explained in full detail within the next chapters.

### 9.1.1 Prepare local environment

The first step is to prepare the local environment. As the guideline “define everything in one place” shall be followed, all global variables, helper functions, network information, filenames and paths shall be specified at the beginning. All further generators will then make use of this pre-defined information. Also, the exact same directory structure is created on the modeler’s computer as it is necessary on the test bench.

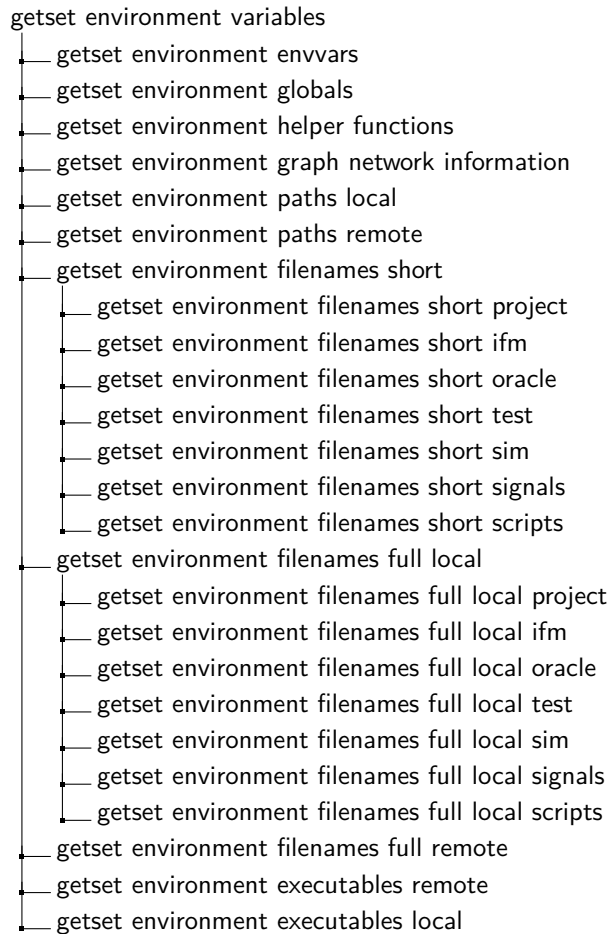


Figure 9.2: get and set environment variables - MetaEdit+ Generator Tree

In many cases, certain environment variables are already defined and used on the modeler’s computer. For instance, the *CLASSPATH* variable is set to a specified path. In order to reduce double definitions — which would need to be changed in two different places and must be synchronized manually — these variables can be incorporated into the model. This can be accomplished by exporting all these

environment variables into a temporary file which is read and parsed afterwards. As this is a simple list where key and value is divided by the “=” character, two regular expressions are used to assign the value to the same-named variable within MetaEdit+. The `envvar` generator accomplishes this task.

A set of helper functions is defined which are heavily used in nearly all following generators. For better readability variable names in the model contain spaces, hyphens or braces. However, the syntax for c++ does not allow this. By using a simple “remove unwanted characters” generator, a qualified c++ variable name is created. For instance, a full name should be as explanatory as possible. By using `:Full Name;%removeUC%removeSpaces%lower` all unwanted characters and spaces are removed from the full name. Also, only lower case letters are used. This eases and unifies the usage of long and declarative names.

```
1 Report '_getset_environment_helperfunctions'
2 [...]
3 /* translator: replace unwanted spaces and '-' with underscore */
4 to '%replaceUC' newline '\- \_ \_ ' endto
5
6 /* translator: removes unwanted characters */
7 to '%removeUC' newline '()\| \ \ \ ' endto
8 [...]
9 endreport
```

Listing 9.1: Local Script `__getset_environment_helperfunctions`

Another generator collects the network information which is defined for each airport system (see listing 9.2 *Local Script `__getset_environment_graph_networkinformation`*<sup>2</sup>). The IP address and port are stored in the system deployment and configuration object (see chapter 6.8.1 *System Deployment and Configuration Object*). Instead of traversing the graph each time this information is needed, this is done once at the beginning. The information is stored in a set of global variables which all consecutively generators can read and use.

The generator examines each airport object if a test bench configuration object is present. If this is the case, all embedded application configurations are processed. For each application, a system name is compiled depending on if it is a simulated system or original equipment. The prefix for simulated system is defined as “sim” (lines 4–11) whereas the prefix is set to “oe” for the real-existing system (lines 11–18). Followed by the prefix, the airport name (without the “unwanted” characters) is appended. Finally, the name of the system (also rectified) is trailed.

This compound is now used to generate two new global variables for each system. One variable contains the IP address while the other holds the port number (lines 21,

---

<sup>2</sup>For better readability, debug information - which is present in most generators - is not shown here. Also, some parts are left out. This is indicated by “[...]”. Furthermore, in most generators “newline” commands are not displayed.

22). For example, all further generators can now use the variable `$AllSystemIP_sim_amsterdam_profiledatastore` to gain the IP address without traversing the graph.

```

1 Report '_getset_environment_graph_networkinformation'
2 [...]
3 /* create a long string containing */
4 /* the names of the configurations */
5 if :Simulated System; = 'T' then
6   variable 'SystemName' write
7     'sim_'
8     $AirportName %lower
9     '_'
10    :Full Name;%removeUC%removeSpaces%lower
11  close
12 else
13   variable 'SystemName' write
14     'oe_'
15     $AirportName %lower
16     '_'
17    :Full Name;%removeUC%removeSpaces%lower
18  close
19 endif
20 [...]
21 /* store IP and Port */
22 /* in an associated "array" */
23 variable 'AllSystemIP_' $SystemName write :IP; close
24 variable 'AllSystemPort_' $SystemName write :Port; close
25 [...]
26 endreport

```

Listing 9.2: Local Script `_getset_environment_graph_networkinformation`

As files are written on the local drive and are copied later to the test bench, filenames and paths are needed for a large amount of generators. Instead of hard-coding each filename into each generator, a set of variables is introduced. Again, this eases the usage as only the “environment” generators need to be edited. All further generators depend on this default setting.

In order to structure this approach, it is split into several generators. First, local and remote paths are specified. These paths represent a standard RTT directory structure with the addition of a place for intermediate files. Also, a path for helper scripts is specified here. This is one part of the basis of full qualified file names.

The other part is to specify the filename itself. These generators for short filenames are structured for each “topic”: IFMs, oracles, simulations, etc. If a filename is used only for a single purpose, it is defined with the appropriate extension (e.g. *project.rtp* or *airport.ifm*). Some filenames are used for header and RTT files. In this case, the extension is left out (e.g. *ifm\_metaedit*). The suffix will be added in the following step.

In order to use full qualified filenames, both parts are put together. In case the extension was not already attached, this is also done in this task. For instance, the generator which will produce the IFM MetaEdit+ will use the variables `$filename_`

`full_local_TB_IFM_MetaEdit_Header` for the header file and `$filename_full_local_TB_IFM_MetaEdit_RTS` for the RTT file.

```
1 Report '_getset_environment_filenames_full_local_ifm'
2 [...]
3 /* Create Filename for Interface Module - MetaEdit */
4 variable 'filename_full_local_TB_IFM_MetaEdit_Header' write
5 $path_local_TB_INC
6 sep
7 $filename_TB_IFM_MetaEdit
8 '.h'
9 close
10 variable 'filename_full_local_TB_IFM_MetaEdit_RTS' write
11 $path_local_TB_Specs
12 sep
13 $filename_TB_IFM_MetaEdit
14 '.rts'
15 close
16 [...]
17 endreport
```

Listing 9.3: Local Script `__getset_environment_filenames_full_local_ifm`

The same procedure is applied for all other files — regardless if they are stored locally or remotely.

The next step is to prepare the local environment by creating the local directories. As all directories are already defined in the previous step, this generator executes the operating system command with the global variables as parameter.

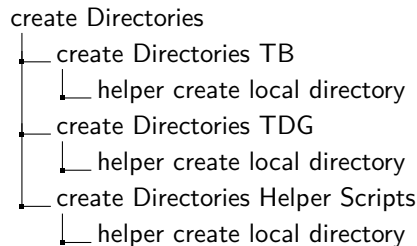


Figure 9.3: create local directories - MetaEdit+ Generator Tree

Again, in order to structure this approach, several small generators — related to topic — are introduced for this task. The actual creation of the directory is accomplished by a helper script which is called for each directory.

```
1 Report '_createDirectories_TB'
2 [...]
3 /* create directory SPECS */
4 $thisDir = $pathToSpecsDir
5 subreport '_helper_createLocalDirectory' run
6 [...]
7 endreport
```

Listing 9.4: Local Script `__createDirectories_TB`

```
1 Report '_helper_createLocalDirectory'  
2 /* create command */  
3 if $thisDir <> '' then  
4   variable 'command' write  
5     'cmd /x /c mkdir '  
6     $ThisDir  
7   close  
8   /* execute command */  
9   external $command executeBlockinge  
10 endif  
11 endreport
```

Listing 9.5: Local Script `_helper_createDirectory`

### 9.1.2 Create Intermediate Files

The purpose of this task is to extract the necessary data from the model and preprocess them for external tools.

The task for creating the intermediate files consists of two main parts. One part is responsible for the input file for the test data generator (see chapter 8 *Test Data Generation*). As this is an external tool which is called within the tool chain, a certain input format must be produced (an example is present in chapter 13.3.4.2 *Airport Intermediate Format File*). The second part of this task is to create an intermediate file for the Position Updater Tool (see chapter 7.4 *Position Updater Tool*).

```
create IMF for Test Data Generator  
├─ create IMF for Test Data Generator PAX  
├─ create IMF for Test Data Generator Airport  
└─ create IMF for Test Data Injector PAX
```

Figure 9.4: create intermediate files, test data - MetaEdit+ Generator Tree

The first part contains information about each passenger and the airport information. Initially, the passenger information is created. Each Terminal object on the main graph is visited and all embedded passenger objects are observed. By examining the attached flight information, it is checked if an airport layout exists for this flight. If this is the case, the properties *isSimulate* and *Random* are evaluated. If neither are set to *false* an entry for this passenger is created, including his temporary ID, late status, and flight departure time.

Also, airport information is attached to this file. The main graph is traversed for all present airport objects and their embedded airport layouts. Each layout consists of one start-state and one end-state. Likely, airport locations and waypoints are also present on such a layout. Transitions connect each of these locations.

For each state, a set of properties is inserted into the file. First of all, the coordinates need to be captured. Depending on the layout property of the airport layout object, a different set is used. If “real layout” is chosen during the modeling phase, then the actual outline dimension of the object’s representation is used. On the other hand, if “relative coordinates” is selected, the values of the specified properties are used.

```
1 Report '_createIMF4TestDataGenerator_State_AirportLocations'  
2 [...]  
3 /* write coordinates */  
4 if :Layout;1 = 'Relative Coordinates' then  
5 ' x1 '' :CoordinateLeft; '''  
6 ' y1 '' :CoordinateTop; '''  
7 ' x2 '' :CoordinateRight; '''  
8 ' y2 '' :CoordinateBottom; '''  
9 else  
10 ' x1 '' left ''''  
11 ' y1 '' top ''''  
12 ' x2 '' right ''''  
13 ' y2 '' bottom ''''  
14 endif  
15 [...]
```

Listing 9.6: Local Script `_createIMF4TestDataGenerator_State_AirportLocations`

In addition to the coordinates, the duration of stay (both high and low value) for this state is also written into the file. While the values in the model are friendlier to read as they are denoted in minutes, the values for the test data generator are adjusted and “converted” to milliseconds.

Furthermore, all requirement related properties are added: the ID for the low- and high-level requirement, a flag if this requirement needs to be evaluated at the end of the test run and counter for the reachability (see chapter 8 *Test Data Generation*). This includes a preset value for adjustments before the start of the test, a minimum value (the least number of visits) and a maximum value (not more than this quantity of visits).

For all airport locations and waypoints, their values from the model are taken. However, this does not apply for the start-state and end-state. As both states have a different behavior fixed presets are used. For instance, a start-state only has outgoing transitions and therefore the count of visits is limited to the maximum of passenger objects at the start of the test — and this will never change during runtime. The same applies for the end-state.

Each transition connects one of the objects to another. It has got two entries describing the source and target state. Furthermore, and very important for the test data generator, is the probability which is attached to each transition (see chapter 8.4 *Test Data Generator Tool*).

In the second part (“create IMF for Test Data Injector PAX” generator), passenger



information is extracted from the passenger objects in the terminal graph. For each real passenger (no simulated or random passenger), the static data is extracted from the model and exported as a simple semicolon separated list.

The Position Updater Tool reads this file in order to create the matched SOAP messages which are used to inject — and therefore additionally bypass — the real moment data.

Additionally, information like the departure time, airport name and flight identity is also taken into account. Furthermore, the contact information (server address and ports) for the check-in system, the position application, and the injector proxy are embedded in this file.

For reasons of convenience, the start position is also extracted from the model and inserted in this file. If the tester chooses a passenger in the position updater tool, this position is preselected and inserted in the messages and signals.

### 9.1.3 Create Test Bed

The test bed consists of several very different elements which are described in the following chapters.

It is not necessary to implement a library which is part of the test bed — but it can be a convenient way to store often used functions in one place. By using such a library, commonly used code fragments can be made available for all instances of the target. So called IFMs are used to transfer data between abstract machines (AMs) and external applications or systems. Such an IFM translates information from a RTTE to a different protocol (possibly on a specified bus system) and vice versa.

As AMs need to communicate between each other, channels and signals are used to transfer data from one instance to another one. These channels and signals need to be defined and are part of the test bed.

#### 9.1.3.1 Library Module

The library module contains functions which are needed by several AMs. This library is used to collect and store all these procedures in one place. The range of functions consists of small helpers to convert strings to integers and vice versa, extract a certain substring from a message or format a phone number to a standardized format. But also larger function sets are present. For instance, a set of client-server methods are implemented here: creation of server ports, sending data over TCP and parsing answers for a sent message.

Also, a scheduler is part of this library. In order to gain more influence on the execution, three different scheduling frameworks have been implemented. The modeler chooses one of the execution schedulers (“Time Switch”) during the modeling phase. This is done by picking one of the properties of the main graph:

- Local time on test bench
- Specify time within model
- No Time (As fast as possible)

```
1 Report '_createTBLibraryModule_MyLibrary_RTS_getCurrentScheduler'  
2 '  
3 '@func modelScheduler_t getCurrentModelScheduler ()  
4 {  
5 '  
6 if :Time Switch; = 'Local time on Test Bench' then  
7   ' return MODEL_SCHEDULER_LOCALTIME;'  
8 endif  
9  
10 if :Time Switch; = 'Specify time within model' then  
11   ' return MODEL_SCHEDULER_SETTIME;'  
12 endif  
13  
14 if :Time Switch; = 'No Time AFAP' then  
15   ' return MODEL_SCHEDULER_ASAP;'  
16 endif  
17 '  
18 }  
19 '  
20 endreport
```

Listing 9.7: Local Script `_createTBLibraryModule_MyLibrary_RTS_getCurrentScheduler`

If the “Local time on test bench” is specified, then the current time of the test bench will be mapped 1:1 to the specified time of the flights. For example, if the first flight is scheduled for 4 pm and the test is started in the morning, nothing will happen for quite some time.

Therefore, two additional scheduling possibilities can be used. If “Specify time within model” is checked, two more properties of the main graph must be filled out: hours and minutes need to be defined. In this case, the waiting time can be shortened by setting the hour value to some point in the future.

The third option is used as a fast forward mechanism and is intended to be used mainly during the modeling phase. By choosing this option, there is no waiting time as the starting time is set to the first pre-calculated timestamp. Furthermore, the execution is accelerated to the ratio 1:60. This means that each calculated minute in real life will only last one second during the test run. If this option is used, the

created test data file (see chapter 8.5 *Passenger Movement Data*) is read in order to gain the smallest timestamp. This timestamp is also the initial value of the scheduler. It results in the immediate start of the simulation.

Depending on the chosen timing conditions a different kind of framework is embedded into the target. Each simulation can use the function `getCurrentTimeInSeconds` which returns a value depending on chosen scheduler. This means that each RTTE can use this function and calculates its behavior on a common ground.

```

1 Report '_createTBLibraryModule_MyLibrary_RTS_getCurrentTimeInSeconds'
2 '
3 @func uint32_t getCurrentTimeInSeconds ()
4 {
5     uint32_t returnValue = 0;
6     '
7     if :Time Switch; = 'Specify time within model' then
8         uint32_t actualTime = 0;
9     endif
10    if :Time Switch; = 'No Time AFAP' then
11        uint32_t actualTime = 0;
12        static uint32_t firstTime = 0;
13        uint32_t boost = 0;
14    endif
15    '
16    '
17    time_t t = time ( 0 );
18    struct tm * now = localtime ( & t );
19    '
20    // calculate actual time according to model settings
21    '
22    if :Time Switch; = 'Local time on Test Bench' then
23        returnValue = ( now->tm_hour * 60 * 60 ) + ( now->tm_min * 60 ) + now
                ->tm_sec;
24    endif
25    '
26    if :Time Switch; = 'Specify time within model' then
27        actualTime = ( now->tm_hour * 60 * 60 ) + ( now->tm_min * 60 );
28        if ( timestampDiffOfModel == -100000 )
29            {'
30                timestampDiffOfModel = actualTime - ' math '(' :ScheduledDate Hour;
                    ' * 60 * 60 ) + ( ' :ScheduledDate Minute; ' * 60 )' evaluate ';
31            }
32        returnValue = actualTime - timestampDiffOfModel + now->tm_sec;
33    endif
34    '
35    if :Time Switch; = 'No Time AFAP' then
36        if ( !isFileRead )
37            {'
38                pathToTDGFile = " $filename_full_remote_TB_TDG_IMF_PaxMove ";
39                inFile.open ( pathToTDGFile.c_str ( ) );
40                if ( !inFile )
41                    {'
42                        @printf ( "-----" );
43                        @printf ( "myLibrary: Cannot read file with test data [ '$
                    filename_full_remote_TB_TDG_IMF_PaxMove ' ]. Aborting!" );
44                    }
45                else
46                    {'
47                        while ( std::getline ( inFile, line ) )

```

```
48 '      {'
49 '          // extract values and time'
50 '          found1 = line.find ( ",", " " );'
51 '          found2 = line.find ( ",", " ", found1 +1 );'
52 '          tmpString = line.substr ( found1 +1, found2 - found1 -1 );'
53 '          tsTemp = atoi ( tmpString.c_str ( " " ) );'
54 '          // store the lowest value'
55 '          if ( tsTemp < ts )'
56 '          {'
57 '              ts = tsTemp;'
58 '          }'
59 '      }'
60 '      // close file'
61 '      inFile.close ();'
62 '      // calculate actual time'
63 '      actualTime = ( now->tm_hour * 60 * 60 ) + ( now->tm_min * 60 ) +
now->tm_sec;'
64 '      // store this timestamp (as this is a mark - from this onwards,
each second counts as a minute'
65 '      firstTime = actualTime;'
66 '      // calculate the timestamp diff'
67 '      timestampDiffOfModel = ts - actualTime;'
68 '      // smallest value is found'
69 '      isFileRead = TRUE;'
70 '      }'
71 '  }'
72 '  else'
73 '  {'
74 '      // calculate actual time'
75 '      actualTime = ( now->tm_hour * 60 * 60 ) + ( now->tm_min * 60 ) + now
->tm_sec;'
76 '      // calculate "boost": = ( actualTime - firstTime ) -> seconds till
last(first) call'
77 '      //      * 60 -> "convert" seconds to minutes'
78 '      //      - ( actualTime - firstTime ) -> remove the seconds passed
between calls, only report minutes'
79 '      boost = ( actualTime - firstTime ) * 60 - ( actualTime - firstTime )
;'
80 '  }'
81 '  returnValue = actualTime + timestampDiffOfModel + boost;'
82 endif
83 '
84 return returnValue;
85 }
86 '
87 endreport
```

Listing 9.8: Local Script `_createTBLibraryModule_MyLibrary_RTS_getCurrentTimeInSeconds`

### 9.1.3.2 Channel Description

Channels are used to transfer information from one RTTE to another one. It uses a port concept, where an output port is used to write data into a channel while the input port is used to retrieve information. Channels can be used to transfer C-like structures. Channels can be used by several RTTEs. This means that every instance

with an input port attached, can read incoming data. This works also vice versa: if an output port is defined each RTTE can write data into that channel [77, p.11]. By the use of IFMs, this data can be transferred to and from external systems.

Although no channels are used in this testing scenario, the provision for adding channels is only geared up.

### 9.1.3.3 Signal Description

Signals are used for exchanging data between RTTEs. This form of communication can be used instead or in addition to channels.

“In contrast to the RT-Tester channel based communication, which needs explicit channel declarations and port definitions in order to access channel data within a test procedure, the signal based communication provides an access which is similar to the handling of global variables.”  
[77, p.181]

Such data can be transferred to and from external systems by using IFMs. Within this testing scenario, signals are transmitted between nine RTTEs, including all four IFMs, the passenger simulation and the notification system (see chapter 7 *Testing Scenario*).

```
create TB Signal Description
├── create TB Signal Description Global
│   ├── create TB Signal Description Global Static
│   └── create TB Signal Description Global Sigdef
│       ├── create TB Signal Description Global Sigdef Header
│       └── create TB Signal Description Global Sigdef RTS
├── create TB Signal Description IFM SMS
├── create TB Signal Description IFM Notification System
├── create TB Signal Description SIM Notification System
├── create TB Signal Description IFM MetaEdit+
├── create TB Signal Description IFM Injector Proxy
├── create TB Signal Description IFM SOAP
├── create TB Signal Description SIM PaxMove
├── create TB Signal Description Oracle SMS
└── create TB Signal Description Oracle Req
```

Figure 9.5: create TB Signal Description - MetaEdit+ Generator Tree

Basically, the generators for creating the signal description can be divided into two groups: the global section with the signal definition for the test bed. The other

group contains the definitions for the AMs. These are split into several generators, one for each RTTE.

The global section itself is also segmented into two sections. One part consists of the definition of the signals. The other part implements a RTT function which is needed for the test bed.

First of all, the definition of each signal is generated. Such a definition consists of five parts. A name needs to be stated for the signal. This name is later on used in order to read from and write to this signal. The second parameter is optional. If wanted, an alias name can be introduced here. In most cases, signal names tend to be rather long as this identifier needs to be unique and also explanatory (a common structure is to embed an interface name and a clear description). As this might not be handy, alias names can be smaller and therefore make source code more readable. However, as only generated code is used and this code does not need to be extended later on, no alias names are used. This section stays empty. The third parameter defines the type of signal (e.g. a number or string). In the fourth position, the length of this signal is specified (e.g. if a string is specified, this holds a maximum of characters which can be stored here). The last parameter defines the length of the queue for this signal.

An example for such a signal definition would look like this: *shortTextMessage\_SIM\_NOTIFICATION\_2\_IFM\_SMS\_text;;char;350;10*. In this case, a string with the length of 350 characters and a queue length of 10 items is created. No alias name is present. As the name of the signal indicates, it belongs to the group “shortTextMessage” and is emitted by the notification simulation. The receiver for this signal is the IFM SMS.

The signal description includes the implementation and definition (see chapter 13.3.2 *Signal Definition Generator Files*) of the function `rtt_init_testbed_signals` and of the enumeration type `rttSignal_e` which are both needed by the RTT. This is later on compiled as part of the test bed.

The second group contains the signal definitions for each AM. One generator is assigned to each AM as one separate file is expected per AM (see chapter 9.1.5 *Create Test Bench Configuration*). The definition consists of four parts which are presented as a semicolon separated list. The first part contains the direction which can be stated as *IN* if this is an incoming signal or *OUT* if the signal is emitted by this RTTE. The second and third parameter contains the name of the signal and its alias name, if present. The last part describes the signal scope which can be either *Testbed* (global signals) or *Testprocedure* (local signals) [77, p.185].

For example, the definition *IN;shortTextMessage\_SIM\_NOTIFICATION\_2\_IFM\_SMS\_text;;Testprocedure* describes an incoming signal for the IFM SMS which was

emitted by the notification simulation. As the name further indicates, the signal belongs to the group “shortTextMessage” and contains “text”.

#### 9.1.3.4 Interface Modules

Generally speaking, an IFM translates information between systems or protocols: it acts as a proxy. Examples for IFMs could be the sending of a push notification (e.g. SMS) or transferring data via SOAP.

The general approach for each IFM is rather simple. It receives data via channels or signals and converts it. These converted data are then transferred to the target system or application. This works the other way around as well. The external system sends data, the IFM receives it and routes it via channels or signals to a RTTE. Of course, an IFM can also be used to transfer information between RTTEs.

Because the conversion is realized within the IFM, both end applications do not need to know how the data is transferred. This implies that bus and protocol are abstracted from both entities. This means that a sender must not necessarily create headers and align variable content, calculate Cyclic Redundancy Checks (CRCs) and process the right sampling time for a specific message. The sender only creates and transmits a signal to the IFM which will take care of all this low-level management. Also, if such a message arrives at the IFM, it decomposes the payload, checks the header and CRC and converts a message into signal content. The receiving entity does not even need to know where the content is originated from.

Because of the concept of IFMs and the usage of abstract signals, signals can be routed to and from all kinds of entities. Such an entity can either be original equipment, then the IFM would route the signal with the appropriate format via the attached bus system, or it can be simulated equipment. In this case, the IFM would emit a signal to the appropriate RTTE.

```
create TB Interface Modules
├── create TB Interface Module SMS
├── create TB Interface Module Notification System
├── create TB Interface Module Meta Edit
├── create TB Interface Module Injector Proxy
└── create TB Interface Module SOAP
```

Figure 9.6: create TB Interface Modules - MetaEdit+ Generator Tree

For this testing scenario, five IFMs are generated. They are described in detail within the next chapters. For each IFM, two generators exist. One generator implements the header file while the second one implements the functionality.

#### 9.1.3.4.1 IFM SMS

The IFM SMS (see figure 7.11 *Extended Scenario Overview*) sends text messages to mobile devices depending on an inhibition status. It communicates with the notification application, the IFM notify system, the oracle SMS and the mobile device. If the user has enabled the guidance system function and requests notifications, messages are sent to his mobile device.

During the init phase of the IFM, the internal inhibition list is filled. This list is generated from the settings of the model. Each terminal object is traversed and all embedded passenger objects are examined. Each passenger's phone number and guidance status preselection is read and stored in a key-value list (*inhibitMap*) where the key is the mobile phone number and the value is a boolean type. If the guidance status is wanted, *true* is set, *false* otherwise.

```
1 Report '_createTBInterfaceModule_SMS_RTS_fillMaps'
2 [...]
3 '// inhibit PAX : ' :Title; ' ' :First Name; ' ' :Second Name; ' ' :
  Surname;
4 ' inhibitMap [ formatMobileNumber ( std::string ( " ' :Mobile; " ) ) ] =
  '
5 if :Guidance Status; = 'Enabled' then
6 ' TRUE; '
7 else
8 ' FALSE; '
9 endif
10 [...]
11 endreport
```

Listing 9.9: Local Script `_createTBInterfaceModule_SMS_RTS_fillMaps`

The IFM SMS main loop is running as long as the macro `@rttIsRunning` evaluates to *true*. It checks two incoming signals. One signal, which is emitted by the notification application, contains an ID, the phone number, and the text which should be sent. The other signal is sent by the IFM notify system and also contains an ID and a phone number. Furthermore, the current inhibition status is transferred. If such an inhibition signal is received, the internal list (*inhibitMap*) is updated in case the value changed.

If a text signal is received, the *inhibitMap* is checked if the embedded mobile phone number is present and if the associated inhibition flag is set. If this is the case, the push notification message is constructed and sent out to the user's mobile device. Furthermore, a signal to the oracle SMS is emitted which contains the current inhibition status and the ID which was received from the notification application.



#### 9.1.3.4.2 IFM Notification System

The IFM notification system (see figure 7.11 *Extended Scenario Overview*) acts as a user front end where the passenger can switch on or off his current guidance status. The front end itself is realized as a web server. The passenger uses his mobile device and opens the web page. After entering his username and password, he can set up his personal guidance preferences.

During the init phase of the IFM, an internal map is populated. This data is extracted from the model as this is entered for each passenger object. Similar to the approach for the IFM SMS, all terminal objects are checked if passenger objects are present. If this is the case, the password and mobile phone number is transferred. The key-value list (*paxcredentialsMap*) uses the chosen login name as key. The value is a structure which contains the chosen password, the mobile phone number and an ID.

```

1 Report '
   _createTBInterfaceModule_IFM_NotificationSystem_RTS_fillAccessLists'
2 [...]
3 ' pax.password = " " :Password; "' ;'
4 ' pax.tempID   = "PAXID_oid_" oid "' ;'
5 ' pax.mobileNumber = formatMobileNumber ( std::string ( " " :Mobile; "'
   ) );'
6 ' paxcredentialsMap.insert ( std::make_pair ( " " :Login; "' ,'
7 '                               pax ) );'
8 [...]
9 endreport

```

Listing 9.10: Local Script `_createTBInterfaceModule_IFM_NotificationSystem_RTS_fillAccessLists`

The web server listens on the port which was specified in the main scenario graph (see chapter 6.2 *Main Scenario Graph*). This is accomplished by extracting the property `:Webserver Port`; and a generator assigns it to a variable which is used for instantiating the server socket.

The main loop of this IFM is executed as long as the macro `@rttIsRunning` evaluates to *true*. It listens for web clients to connect to the server socket. If this is the case, the login page is sent to the client and the response is evaluated (if user name and password are correct). Afterwards, two signals are emitted. One signal is sent to the oracle SMS so that the sent status can be calculated at the end of the test. The second signal is sent to the IFM SMS. It also contains the inhibition status for a specific user (see chapter 9.1.3.4.1 *IFM SMS*).

#### 9.1.3.4.3 IFM MetaEdit+

The IFM MetaEdit+ is one part of the bridging between the test on the test bench and the model on the modeler’s computer (see figure 7.11 *Extended Scenario Overview*). It receives three kinds of information from the RTTE. Firstly, there are register and unregister commands, so that passenger objects are created and deleted from the model. These objects are only visible during the test run and are dismissed afterwards. Second, passenger movement is transferred to the model so that the representations of passenger objects are displayed at different positions in the model. Last, notification information is transferred in order to change the representation within the model.

The hostname and port of the modeler’s computer are specified as properties in the main graph. The generator exports this into the IFM. Also, the name of the scenario needs to be present in order to connect the MetaEdit+ bridge to the model (see chapter 7.5 *MetaEdit Bridge*). During the init phase of the IFM, it tries to connect to the MetaEdit+ bridge and — if successful — initializes its database.

```
1 Report '_createTBInterfaceModule_IFM_MetaEdit_RTS'  
2 [...]  
3 static std::string nameOfScenario = " type ";  
4 static std::string metaEditAPIHostName = " :MetaEdit Proxy IP; ";  
5 static uint16_t metaEditAPIport = " :MetaEdit Proxy Port; ";  
6 [...]  
7 endreport
```

Listing 9.11: Local Script `_createTBInterfaceModule_IFM_MetaEdit_RTS`

The main loop of the IFM is running as long as the macro `@rttIsRunning` has the value *true*. It receives three signals from the RTTE IFM injector proxy, the notification application, and the passenger move simulation (see chapter 9.1.4.3 *Test Data Stimulus*).

The injector proxy and the passenger move simulation send register and unregister command signals to this IFM. They are evaluated and the appropriate command is sent to the MetaEdit+ bridge. In the end, new passenger objects (see chapter 6.7.7 *Passenger Move Object*) are displayed on or removed from the airport layout graph.

Also the signal for updating the current position for a passenger object is emitted by the injector proxy and the passenger move simulation. It contains the ID for a certain passenger and his coordinates. The signal is processed and the corresponding command is created and sent to the model via the MetaEdit+ bridge.

The third signal is received from the notification application. This signal contains the ID for a specific passenger and a “late” state. This state could be either “in-time”, “late” or “too late”. Depending on this information, the representation of the

passenger object on the airport layout graph changes. The “needle” which depicts the current position changes its color to green (in case of “in-time”), to orange (in case of “late”) or to red (in case of “too late”). Also, by means of using filters on the airport layout graph, it is possible to display only “too late” passengers while all other are set to be invisible so that only this kind of information can be regarded quickly.

The IFM and the bridge communicate via proprietary commands sent over a standard TCP connection (see chapter 7.5 *MetaEdit Bridge*). This is similar to a telnet session as commands are sent in clear text without any header or CRC. As both instances are running in a separate network and are only used for testing activities, this seems to be sufficient and reliable. However, other environments might need a more robust communication channel. In such a case, the communication should be secured by standard means, for instance CRC protection.

#### 9.1.3.4.4 IFM Injector Proxy

The IFM Injector Proxy (see figure 7.11 *Extended Scenario Overview*) handles the register, unregister and update position commands which are emitted by the position update injector tool. It receives these commands via a standard TCP connection in a clear text format. The commands itself carry the exact same payload which would be transmitted as signal values so that it can be regarded as “signal over TCP”.

Depending on the received command, the IFM relays the message to either the IFM MetaEdit+ or the requirement’s oracle. As the position update injector tool acts the same way as the passenger move simulation, the same signals are emitted to the same target instances. In case of a register or unregister command, the payload is relayed to the IFM MetaEdit+. If an update position command is received, it is sent to both endpoint instances: the new position is displayed in the model and also used for the requirement’s coverage calculation.

The IFM utilizes the same port as the MetaEdit+ proxy but as both proxies are running on different machines there is no clash.

```
1 Report '_createTBInterfaceModule_IFM_InjectorProxy_RTS'  
2 [...]  
3 // use the same port as the metaedit proxy (as both are running on  
   // different machines, this is not a problem!)  
4 serverPort = ':MetaEdit Proxy Port;';  
5 [...]  
6 endreport
```

Listing 9.12: Local Script `__createTBInterfaceModule_IFM_InjectorProxy__RTS_AM`

Therefore, the IFM injector proxy is a rather small RTTE without any complex internal logic.

#### 9.1.3.4.5 IFM SOAP

The IFM SOAP is a sophisticated RTTE which connects the passenger movement simulation with the simulated and/or real equipment (see figure 7.11 *Extended Scenario Overview*). It receives the emitted signals from the passenger movement simulation and creates the accompanying SOAP messages. Afterwards, the messages are sent to the target system.

Large parts of this IFM are not static code which is only written to disk but generated from the model. This means that the implementation of many functions depends greatly on the model. This includes mapping tables, for instance, a string to enum type conversion. In this case, all existing template graphs are evaluated and all embedded messages are taken into account. A `std::map` called *internalDBMessageName2MessageType* is filled with the message type as string and is associated with an enum type which starts with *SOAP\_MSGTYPE\_* and ends with the name of the message. The enum type itself is also created by the same approach (and is placed in the associated header file). This enables a generic access on the model. If new messages are defined or removed, the maps will be updated accordingly.

```
1 Report '_createTBInterfaceModule_IFM_SOAP_RTS_fillAccessLists'  
2 [...]  
3 do graphs where type = 'Templates'  
4 {  
5   foreach .TEMPLATE_Message  
6   {  
7     internalDBMessageName2MessageType [ "  
8       :Message Type;%removeUC%removeSpaces  
9       " ] = SOAP_MSGTYPE_  
10    :Message Type;%removeUC%removeSpaces  
11    ';' ;  
12  }  
13 }  
14 [...]  
15 endreport
```

Listing 9.13: Local Script `_createTBInterfaceModule_IFM_SOAP_RTS_fillAccessLists`

All created messages<sup>3</sup> are exported into this RTTE. Each message is stored in a map which uses the generated enum type as key and the complete message as value. All template graphs are traversed and all including messages are exported. The map *internalDBMessageType2MessageTemplate* is populated by a fixed SOAP header and

---

<sup>3</sup>The same approach is also used in the behavior graphs (see chapter 9.1.4.2.3 *Section 3 - Main Computation*).

footer. The content is applied by calling the generator `insertDataStructure`. The message string itself is indented like shown in the model. The variable `cnt` is set to the value 2 (as two spaces are initially needed in front of each beginning line). With each recursive call, as these message items can be nested, this variable is used to calculate leading spaces. The result is a SOAP message with a perfect layout.

```

1 Report '_createTBInterfaceModule_IFM_SOAP_RTS_fillAccessLists'
2 [...]
3 do graphs where type = 'Templates'
4 {
5   foreach .TEMPLATE_Message
6   {
7     $cnt='2'
8     ' internalDBMessageType2MessageTemplate [ SOAP_MSGTYPE_'
9     ':Message Type;%removeUC%removeSpaces
10    ' ] = "<?xml version='1.0' encoding='utf-8'?'>\n"
11    ' "<soapenv: Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
12    '   envelope/\ ">\n"
13    '   " <soapenv:Header/>\n"
14    '   " <soapenv:Body/>\n"
15    '   " <data>\n"
16    '
17    /* include all attached datastructures */
18    subreport '_insertDataStructure' run
19    '   " </data>\n"
20    '   " </soapenv:Body/>\n"
21    '   " </soapenv: Envelope>\n";'
22  }
23 }
24 [...]
25 endreport

```

Listing 9.14: Local Script `_createTBInterfaceModule_IFM_SOAP_RTS_fillAccessLists`

The called sub report `_insertDataStructure` runs through all connected data structures and starts by inserting the name of the current data structure container (see chapter 6.10.3 *Data Structure Object*) as an embedding XML entity. Afterwards, all containing entries are extracted from the model and also arranged accordingly.

```

1 Report '_insertDataStructure'
2 /* include all attached datastructures */
3 do ~InheritanceTarget~InheritanceSource.()
4 {
5   /* indent - struct name */
6   $cnt+=%null
7   $times = $cnt
8   variable 'times' append '*2' close
9   variable 'indentCnt' write math $times evaluate close
10  $indent = $indentCnt
11  $preFix
12  subreport '_indent' run
13  '<' :Name;%removeSpaces%lower '>'
14  $postFix
15
16  /* get all entries (if not empty) */

```

```
17 $cnt++%null
18 $times = $cnt
19 variable 'times' append '*2' close
20 variable 'indentCnt' write math $times evaluate close
21
22 if :1\. Item - Name; <> '' and :1\. Item - Type; <> '' then
23   $indent = $indentCnt
24   $preFix
25   subreport '_indent' run
26   '<'
27   :1\. Item - Name;%removeSpaces%lower
28   '></'
29   :1\. Item - Name;%removeSpaces%lower
30   '>'
31   $postFix
32 endif
33 [...]
34 /* insert all other attached structures! */
35 do ~InheritanceTarget-InheritanceSource
36 {
37   subreport '_insertDataStructure' run
38 }
39 [...]
40 endreport
```

Listing 9.15: Local Script `_insertDataStructure`

Although the message template and database can be extracted from the model, some hand-written code is necessary to glue parts of the model together. This is necessary as there is no link between a message template and data in the model. For example, the *register* message contains parts of the properties of a passenger (which could be mapped) but also to other data on different graphs and a set of default values. The mapping of data could be done either implicitly in the model by introducing more links between objects or by mapping variable names. A certain property of a passenger would then need the same name (or a name which can be mapped by algorithm) as an entry in the SOAP message. This could not be accomplished within the E-Cab project as too many stakeholders have been involved and all connected systems have been under design and development phases.

Furthermore, a variable in the model might be used in different messages — with different identifiers in each (or at least more than two) messages.

Also, for all variables in the model a certain naming scheme was defined which contradicts to an automated mapping. As there are several hundred variables within the model, a prefix is used to identify which variable belongs to which object. A variable name *tmpID* cannot be easily associated to exactly one object. By using the prefix *PAX\_BoardingPass\_* it is clear for the modeler that this variable belongs to a passenger object — and not to another object. A temporary ID is a common property which needs to be assigned to several objects so a clear distinction must be given.

In order to have more flexibility, an automatic mapping has been discarded. This mapping is done explicitly by one hand-written function for each outgoing message. Therefore, within this IFM three functions have been implemented to create a register, unregister and position update message. The scheme of these functions follow the same idea. First, the message template is instantiated, then the entries are filled one-by-one. Such a function uses the *message* string parameter for the returned message (an empty string is passed to the function). The template is assigned and the values are inserted between the opening and closing XML entities. As the `soapInsertXMLEntity` function returns a boolean value to indicate the success state of the call, each returned value is used to calculate the return value of this function.

```

1 Report ' _createTBInterfaceModule_IFM_SOAP_RTS_constructLocationUpdate '
2 '
3 @func static bool_t constructLocationUpdate ( std::string & message,
4         std::string & myPaxID,
5         std::string & RFIDPassive,
6         std::string & XValue,
7         std::string & YValue )
8 {
9     bool_t returnValue = FALSE;
10
11     // get message template
12     message = internalDBMessageType2MessageTemplate [
13         SOAP_MSGTYPE_UPDATEAirportLocation ];
14     // fill template manually as not all collected properties of this PAX
15     // needs to be inserted in the message
16     // location update
17     returnValue = soapInsertXMLEntity ( message, "temporarypaxid", myPaxID
18         );
19     // location update - coordinates
20     returnValue = returnValue && soapInsertXMLEntity ( message, "xcoordinate"
21         , XValue );
22     returnValue = returnValue && soapInsertXMLEntity ( message, "ycoordinate"
23         , YValue );
24     // location update - gps
25     returnValue = returnValue && soapInsertXMLEntity ( message, "gpstype", "
26         RFIDPassive" );
27     returnValue = returnValue && soapInsertXMLEntity ( message, "gpscode",
28         RFIDPassive );
29
30     return returnValue;
31 }
32 '
33 endreport

```

Listing 9.16: Local Script `_createTBInterfaceModule_IFM_SOAP_RTS_constructLocationUpdate`

The inserting of items into a message can be accomplished by a generic method. As the template consists of opening and closing tags, the content needs to be placed in between these tags. As this is independent of the specific message, it is therefore independent of the model. If the message in the model is changed (e.g. new or

deleted items, renaming of items), the task to instantiate a message and fill it with values does not need to be adapted.

```
1 Report '_createTBInterfaceModule_IFM_SOAP_RTS_soapInsertXMLEntity'  
2 '  
3 @func static bool_t soapInsertXMLEntity ( std::string & message,  
4         std::string key,  
5         std::string item )  
6 {  
7     size_t foundStartTag;  
8     std::string startTag;  
9     bool_t returnValue;  
10  
11     returnValue = FALSE;  
12     startTag = "<" + key + ">";  
13  
14     foundStartTag = message.find ( startTag ) + startTag.size ();  
15  
16     if ( foundStartTag != std::string::npos )  
17     {  
18         message.insert ( foundStartTag,  
19             item );  
20         returnValue = TRUE;  
21     } // found tag, inserted item  
22     return returnValue;  
23 }  
24 '  
25 endreport
```

Listing 9.17: Local Script `__createTBInterfaceModule_IFM_SOAP_RTS__soapInsertXMLEntity`

## 9.1.4 Create Test

The test section consists of three larger parts: oracles, simulated systems and the main test including the test stimulus. The test and the test data trigger and stimulate the behavior of all attached systems. The simulated systems are created from the model and can be considered as the environment for the test. At the end of the execution of the test, the oracles check their received signals against the expected behavior. This is explained in the following chapters.

### 9.1.4.1 Oracles

Two oracles are present which will check if certain conditions are met. If both oracles set their verdict to *passed*, the test is considered to be also *passed*.

The first oracle monitors the sent SMS to the passengers while the second one checks if the requirements are met after runtime.



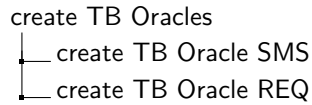


Figure 9.7: create TB Oracles - MetaEdit+ Generator Tree

#### 9.1.4.1.1 Verification of Transmitted Short Messages

This oracle checks if the count and type of short messages match the expected values. During the test execution, short messages are used to inform the passenger whether he is in time or if he needs to proceed to the next gate or if he is too late and will not catch his plane anymore. The passenger can opt-out from receiving such messages by visiting the E-Cab website and disable such notifications at any time.

The generators for this machine consist of a header and RTS part, while the latter one is subdivided into more granular generators. Each of these is responsible for a certain implemented function.

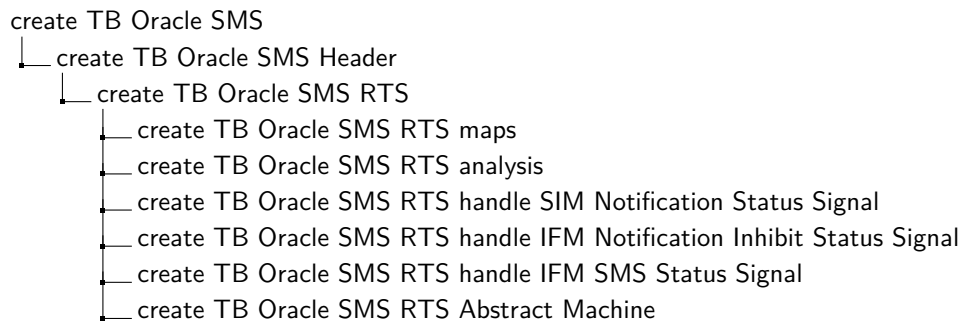


Figure 9.8: create TB Oracles - MetaEdit+ Generator Tree

This abstract machine collects the signals transmitted between IFMs notify system, SMS and the notify application simulation. It compares the expected state of signals with the actual sent state.

The simulation notify application sends a status to the oracle each time it tries to send a push notification (via the IFM SMS). Also, the IFM SMS sends a status to the oracle if this message (e.g SMS) was really sent to the mobile device, as this depends on the inhibit status for this passenger. This inhibit status is adjusted by the IFM notify system which relays the received command from the user's mobile device to the IFM SMS and the oracle (see figure 7.11 *Extended Scenario Overview*).

The main loop waits for the incoming signals and calls the appropriate `handle signal` functions. All these received signals are collected in an internal database. They will be analyzed at the end of the test run. If all RTTEs receive the `@rttStopTest;` command,

which is initiated by the passenger movement simulation (see chapter 9.1.4.3 *Test Data Stimulus*), the oracle will execute its `FINIT` section which contains the `analysis` function.

If all SMS have been sent out which should not have been inhibited and all SMS have been withheld which are set to inhibited, the oracle will set its status to *passed*.

#### 9.1.4.1.2 Verification of Requirements

This oracle checks if the requirements, which are attached to each airport location and waypoint, are met at the end of the test run.

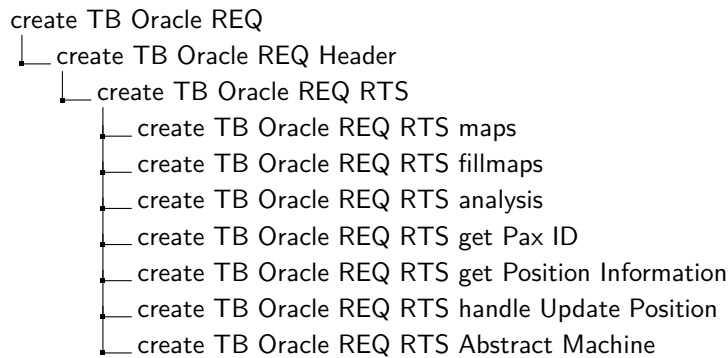


Figure 9.9: create TB Oracles - MetaEdit+ Generator Tree

During the initialization phase, all locations and waypoints of the airport layout are traversed and stored in an internal database. The name and position, either the specified values or the actual coordinates (depending on the layout setting, see chapter 6.7 *Airport Layout Graph*), for each object are saved. Also, all further requirements flags (ID, count preset, minimum and maximum count) for high-level and low-level requirements are collected.

In addition, four more properties are attached to such a data set. Two variables are defined which are used as the internal counter (one for the high-level and low-level requirement). Each time this object is entered, the timer is increased. For each requirement, a *isPassed* flag is attached which is initialized with the value *false*.

During runtime, the oracle receives the passenger movement signals and checks if a passenger is located within the boundaries of a location. If this the case, the internal counter is increased.

If the passenger movement simulation initiates the stop of the test run, the `FINIT` phase with the analysis method is executed. The internal database is evaluated and checked if the conditions for each requirement are met (e.g. is the location touched,

and if so, is the count amongst the minimum and maximum value). If this is the case for all requirements, the verdict is set to *passed*.

### 9.1.4.2 Simulated Systems

Within the model, the applications running at an airport can be set to either original equipment or simulated environment. Therefore, the main graph is traversed and all airport objects are evaluated. If a system deployment and configuration graph is present, all embedded application configurations are checked. If the property *Simulated System* of such an application is set to *false*, it is ignored (as the original system is present). Otherwise, the modeler specifies that a simulation needs to be created for this application.

Each simulated system consists of roughly three parts. The first part contains the generated items from the template graph. The second one contains static functions which need to be present in all simulated RTTEs (this includes all “helper” functions). The third part contains the generated code of the behavior graphs. As these parts are identical for all simulated systems, no distinction is made in the following sections. The generated code — and therefore the execution semantics — of the simulated systems are solely specified by the attached behavior graph. As this approach is highly generic, changes in the simulation, adding or removing more functionality or applying more simulations can be quickly done by modifying the entities in the template graph and behavior graph.

#### 9.1.4.2.1 Part 1 - Template Graph

From the template graph, all messages and databases are processed and translated into enumeration types, structures and maps. All of this is initiated by the `_createTBSimulatedSystem_variables` generator.

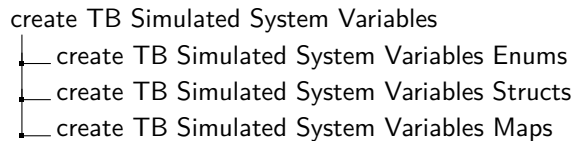


Figure 9.10: create TB Simulated System Variables - MetaEdit+ Generator Tree

Each message of the template graph is processed to an enumeration type which is later on used as a key for maps. Each entry of `soap_msgtype_t` consists of the prefix `SOAP_MSGYTPE_` followed by the name of the message (where all unwanted characters including spaces are removed). If the current behavior graph contains a

*calcNotification* object, an additional enumeration type (*messagetype\_t*) is added which contains states for a passenger being in-time, late or too late to catch a flight.

Furthermore, each database is translated to a structure where the name of the database is also used as structs name (spaces and other unwanted characters are removed). All attached data structure objects (see chapter 6.10.3 *Data Structure Object*) are traversed and the content is used as the body of this structure. At the end, a container is created which can hold data from at least one message. The idea is to merge several messages to one structure. This mechanism is used during runtime to relay messages. For instance, an application receives a message and needs to create a new message with partly additional or different data which will be sent afterwards. By combining both messages into one container, all information is stored in one place and the extraction of data can be easily performed.

In contrast to these dynamic (because they solely depend on the current model) structures, five static ones are added in case of the presence of the process update object or the calculate notification update. If the process update object is used on the behavior graph, the structure *airportarea\_t* is added which contains an areaID and four integers which describe the dimension and location of such an area. If the behavior graph contains a calculate notification object, four additional structs are inserted into the generated target file. The structure *airportAreaPaxLocation\_t* contains the last and current areaID. This is later on used for each passenger to detect and store a change of areas. The *airportAreaInformation\_t* consists of three integers and stores the average time (which is needed to walk through an area), the minimum time (which is needed at least to pass through an area) and the interval in which push notifications need to be sent. The *notificationMessage\_t* structure contains the needed information for sending and tracking messages (timestamp, type of message, and flags if certain messages have been sent out already). The last structure contains the needed information for the content of such a push notification (title and name of the passenger and the mobile number).

In order to store the created structures, several maps are added to the generated target files. There is one set of dynamic created maps which are extracted from the template graph. For each database object, one map is created with a string as key and the previously created structure as value. Also, several static maps (because they are needed for every module and do not depend on the model) are created such as storing a message template (used as value while the message type is used as key) or mapping a message type (coded as string) to an enumeration value. Similar to the creation of the structures, some more maps are added in case of the usage of the process update object or the calculate notification object. The maps are used to store instances of the previously created structures.

### 9.1.4.2.2 Part 2 - Helper Functions

The second part contains helper functions which are necessary for other functions to complete its tasks. For instance, methods are needed to create messages and send these to the target systems. SOAP entities need to be filled in a template message or read and extracted from a received message.

```

create TB Simulated System Methods
├─ create TB Simulated System SOAP insert XML entity
├─ create TB Simulated System SOAP return XML entity
├─ create TB Simulated System SOAP send message
├─ create TB Simulated System return message type
├─ create TB Simulated System get network properties
└─ create TB Simulated System handle connection

```

Figure 9.11: create TB Simulated System Helper Functions - MetaEdit+ Generator Tree

The function `soapInsertXMLEntity` receives a pointer to a message, which could be a template message or a received message via network from a different system, the key and the item which should be inserted. If the entity is found, the item is inserted and the calling function is informed by returning the boolean value *true*. The value *false* is returned otherwise. As the E-Cab SOAP messages are simple, with respect to its complexity. The used entities are expected to consist of an opening and closing tag. As no further parameters are used, the complexity of this function is rather low. To the contrary, the function `soapReturnXMLEntity` parses a given message for a specified key. Everything enclosed by this key will be returned to the calling function.

All network information was extracted from the model and stored in local maps in each RTTE. These local lookup tables are wrapped by a function which returns IP addresses and ports for a given system at an airport.

The create message helper obtains a reference to an empty string which will hold the filled out message later on. Furthermore, a message type and a map containing a list of keys with an associated value is obtained. The method retrieves an empty message template, according to the message type parameter, from the local template map and assigns it to the referenced string. Afterwards, the given map is traversed for each key/value pair and the message is filled by these entries. At the end, the string contains a valid SOAP message body.

```

1 Report '_createTBSimulatedSystem_createMessage'
2 [...]
3 '
4 // get template for message type
5 message = internalDBMessageType2MessageTemplate [ messagetype ];
6 // find and insert all items from the list into template
7 for ( mapItemsIter = mapItems.begin () ;

```

```
8  mapItemsIter != mapItems.end ();
9  mapItemsIter++ )
10 {
11  returnValue = returnValue + soapInsertXMLEntity ( message,
12  mapItemsIter->first,
13  mapItemsIter->second );
14 }
15 '
16 [...]
17 endreport
```

Listing 9.18: Local Script `_createTBSimulatedSystem_createMessage`

The `createHeader` function prepares the SOAP header for a specific message body. It calculates content length fields and inserts message type elements. Together with the `createMessage` function, this forms a valid message which will be sent out by the function `soapSendMessage`. This function is a wrapper and creates the message by using the two above mentioned methods. Afterwards, a network connection is created and the message is sent to the target system.

The `handleConnection` function depends highly on the associated behavior graphs. The main part consists of a switch-case statement where the type of the received message is used as differentiation. Each connected behavior object is traversed and a function call is inserted. The received message will always be handed over to the called function.

```
1 Report '_createTBSimulatedSystem_handleConnection'
2 [...]
3 '
4 // determin type of message
5 switch ( returnMessageType ( message ) )
6 {
7 '
8 /* dive into attached logic graph */
9 do decompositions
10 {
11 /* goto handleConnections element */
12 foreach .BEHAVIOUR_HandleConnections
13 {
14 do ~SystemBehaviourSource~SystemBehaviourTarget
15 {
16 do .()
17 {
18 if type = 'BEHAVIOUR_StoreMessage' or type = 'BEHAVIOUR_Erase' or type
19 = 'BEHAVIOUR_SendMessage' or type = 'BEHAVIOUR_Opaque' or type =
20 'BEHAVIOUR_Wait' or type = 'BEHAVIOUR_CalcNotification' or type =
21 'BEHAVIOUR_ProcessUpdate' then
22 case SOAP_MSGTYPE_ ' :MessageType;1%removeUC%removeSpaces ' :'
23 returnValue = handleSOAPMessage_ ' :MessageType;1 %removeUC%
24 removeSpaces '_' oid ' ( message );'
25 break;'
26 endif
27 }
28 }
29 }
30 }
```

```

27 [...]
28 endreport

```

Listing 9.19: Local Script `_createTBSimulatedSystem_handleConnection`

### 9.1.4.2.3 Part 3 - Behavior

The most important part of the generators for the simulated systems is the conversion of the behavior graphs to source code. The produced code highly depends on the behavior graphs and the objects from the template graph. By adjusting messages and attached data structures, the produced code will change accordingly.

The general idea for implementing the behavior is to create a “tree of calling functions”. This means that the first behavior object calls the following object and so on. Each object returns a boolean state if the processing of the own block was successful or an error occurred. This return value is used in the calling function to calculate its own return state — based on the own state concatenated with the received state (see chapter 9.1.4.2.3 *Section 4 - Function Footer*).

The received message is handed over from one function call to the next one. Although this information might not be necessary for some functions, it might be relevant for proximate function.

```

create TB Simulated System create behavior methods
├─ create TB Simulated System Store Message
├─ create TB Simulated System Erase
├─ create TB Simulated System Send Message
├─ create TB Simulated System Wait
├─ create TB Simulated System Calculate Notification
├─ create TB Simulated System Opaque
└─ create TB Simulated System Process Update

```

Figure 9.12: create TB Simulated System Create Behavior Methods - MetaEdit+ Generator Tree

The function `createBehaviorMethods` traverses through a behavior graph and inserts a method for each behavior object. It is not necessary for the generator to pay attention to the order or connection of the embedded methods. The transitions between objects are realized as function calls and as each function is ensured to gain a unique function name, the generator does only need to create blocks for each function.

```

1 Report '_createTBSimulatedSystem_createBehaviourMethods '
2 [...]
3 do decompositions

```

```
4 {
5  /* handle all StoreMessage blocks */
6  foreach .BEHAVIOUR_StoreMessage
7  {
8    $targetType = type
9    subreport '_createTBSimulatedSystem_createBehaviourMethods_StoreMessage'
        run
10 }
11 [...]
12 endreport
```

Listing 9.20: Local Script `_createTBSimulatedSystem_createBehaviourMethods`

Each generated behavior function consists of four sections. In the first section, the associated name of the message and database must be found and retrieved from the template graph. The second one constructs the function header including its name. The actual computation is covered by the third section. At the end, the next behavior object is found and the associated function call is created. These steps are explained in detail by using `sendMessage` as an example. The structure and approach is identical for all behavior objects regarding all sections except the third section (of course, as the actual behavior is placed here).

### Example Check-In System

The behavior graph of the check-in system (see figure 9.13 *Behavior Graph Check-In System*) is used in order to show the generation of an existing RTTE. The check-in system waits for incoming messages which could be either of type “Checkout Pax” or “Checkin Pax”. In case of a check-in request, the received data is analyzed and the required information is stored locally (`Store Message`). Afterwards, two messages are sent out to the profile-datastore system and the on-ground baggage system. In case the received message is of type “Checkout”, the systems profile-datastore and on-ground baggage are informed and the locally stored data is erased.

If a message is received which neither is “Checkout Pax” nor “Checkin Pax”, the `Handle Incoming Message` function is terminated and the `Wait for Messages` function is active.

This means that for this graph, six independent functions will be created for this simulated system: four `Send Message`, one `Erase`, and one `Store Message` block.

### Section 1 - Get Database

The intention of the first section is to find the associated data base for an incoming message. The type of the message is defined only at the very first transition from the `Handle Incoming Message` object (see chapter 6.9.10 *Behavior Connection Types*) but



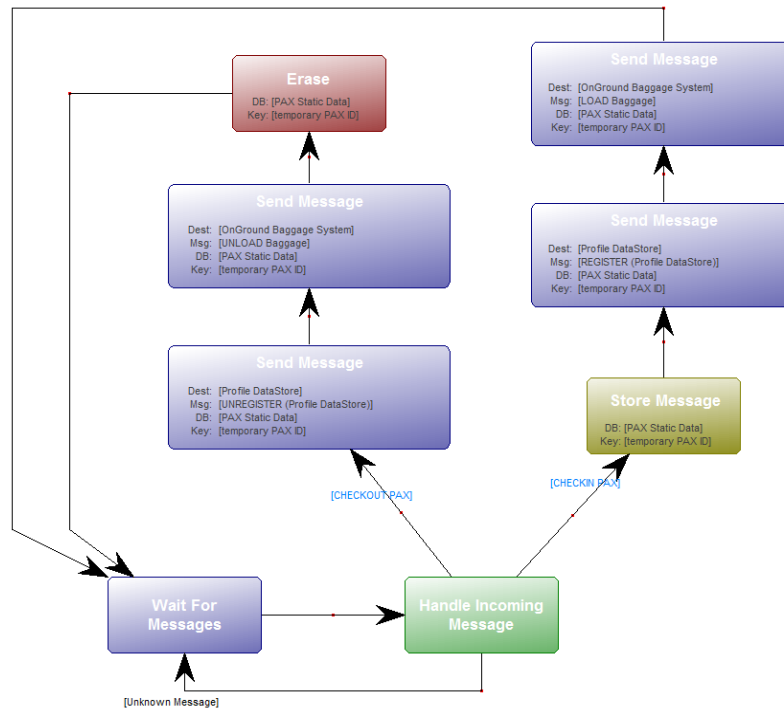


Figure 9.13: Behavior Graph Check-In System

not at the consequent transitions. The type does not change as the same message is passed from one object to the next one. Therefore, only the first transition carries the name of the message type. As this would be redundant information, it is omitted for all further transitions.

The first step is to gain the information of the message type. The generator `getMessageType` (see listing 9.21 *Local Script `_getMessageType`*) crawls from the current object from the source part of the transition to the target part (line 5) until the target object is of type “HandleConnection” (line 7). The name of the message is stored in the variable `msgType` (line 3).

```

1 Report '_getMessageType'
2
3 variable 'msgType' write ~SystemBehaviourTarget:MessageType; close
4
5 do ~SystemBehaviourTarget~SystemBehaviourSource.()
6 {
7   if type <> 'BEHAVIOUR_HandleConnections' then
8     subreport '_getMessageType' run
9   endif
10 }
11 endreport

```

Listing 9.21: Local Script `_getMessageType`

This generator `_getMessageType` is called by the `send message` generator (see line 4, listing 9.22 *Local Script \_\_createTBSimulatedSystem\_\_createBehaviourMethods\_\_SendMessage - Section 1*). As described above, the type of the source message (which is received) is now available through the variable `msgType` and used from now on. The name of the target message (which should be sent) is retrieved by the local property `:Message type`; and stored in the local variable `targetMsgType` (line 7).

The next step is to find the message on the template graph. If the message is found (line 13), the *database message connection* transition (see chapter 6.10.4 *Template Connection Types*) is used to find the attached database (line 16). The key and the name of this database is then stored into local variables `DBKey` and `DBName` (line 19, 20).

```
1 Report '_createTBSimulatedSystem_createBehaviourMethods_SendMessage'
2
3 /* get message type */
4 subreport '_getMessageType' run
5
6 /* set targetMsgType to target message type */
7 variable 'targetMsgType' write :Message Type; close
8
9 /* get associated database */
10 do graphs where type = 'Templates'
11 {
12 /* find message */
13 foreach .TEMPLATE_Message; where :Message Type; = $msgType
14 {
15 /* find associated DataBase */
16 do ~DataBaseMessageConnectionTarget~DataBaseMessageConnectionSource.()
17 {
18 /* get key and name from database */
19 variable 'DBKey' write :Key; close
20 variable 'DBName' write :Name;%removeSpaces%lower close
21 }
22 }
23 }
```

Listing 9.22: Local Script `__createTBSimulatedSystem__createBehaviourMethods__SendMessage - Section 1`

## Section 2 - Function Header

The second part creates the header of the function which mainly consists of its signature. First, a comment describing this function is inserted, including a short description of the parameter and the return value (lines 4–10, see listing 9.23 *Local Script \_\_createTBSimulatedSystem\_\_createBehaviourMethods\_\_SendMessage - Section 2*). Each behavior object returns a boolean value (line 14), describing if the computation was successful or failed. This is used later on to calculate the overall return value in the fourth section.

The naming scheme of each function name is applied as follows. First, the prefix *handleSOAPMessage\_* is used, followed by the name of the message (with removed unwanted characters). At the end, the internal MetaEdit+ ID (oid) for this object is attached in order to distinguish between methods of the same type (e.g. a send message object might be placed several times on a behavior graph). This ensures unique function names (line 14).

```

1 [...]
2 /* header of function*/
3 '
4 /** send Message
5 * uses the key of the received message as key of the 'to-send' map
6 *
7 * @param message received message as string reference
8 *
9 * @return boolean value; TRUE if successful, FALSE otherwise
10 */
11
12 /* body of function */
13 '
14 @func static bool_t handleSOAPMessage_' $msgType %removeUC%removeSpaces '_
    ' oid ' ( std::string & message )
15 {
16 #ifdef SIM_DEBUG
17 @printf ( "' $SimSystemName %removeUC ': handleSOAPMessage_' $msgType %
    removeUC%removeSpaces '_ ' oid ': START" );
18 #endif
19 [...]
```

Listing 9.23: Local Script `_createTBSimulatedSystem_createBehaviourMethods_SendMessage` - Section 2

Also, if the verbose flag was switched to *on* on the main graph (see chapter 6.2 *Main Scenario Graph*) during modeling phase, the symbol `SIM_DEBUG` is set accordingly which will lead to such a note in the log file (lines 16–18).

### Section 3 - Main Computation

If a message is received and the content is stored (via `Store Message` (see chapter 6.9.5 *Store Message Object*)), a new data base entry is created in case no entry existed before. If an entry was already present, the content is updated. Each message is linked to a data base object as defined in the template graph (see chapter 6.10 *Template Graph*). The data base consists of a structure which will hold all items from all attached messages. Therefore, if another message with the same key and which has a different type is received and stored, the entry is updated. It could be the case that some old items are overwritten or only new items are added to the current existing data set.

On the other hand, if a message is sent out, an entry of its attached data base is used to fill out the content of this message. The specified key of the message in the template graph is used as the key for the associated data base. Each message which is sent out is an aftereffect of a received message. The link between both messages is the key of each message. The content of both messages can even be stored in two different data bases.

In order to send out a message, three steps are needed. First, the key from the incoming message needs to be stored. Then, the data for the message to be sent is retrieved from the data base using the previously saved key. At the end, this data is used to fill out the template and send out the message.

In the beginning (see listing 9.24 *Local Script \_\_createTBSimulatedSystem\_\_createBehaviourMethods\_SendMessage - Section 3*), a local map (*mapItems*) which will hold the data of the outgoing message is created (line 4). Also, the iterator is defined. The variable *DBName* holds the name of the data base (line 5).

The return value is initialized with the value *false*. It is a common approach to use such a “defensive programming” statement (starting from the assumption that everything will fail): all variables are set to a default “worst case” or initial state (line 10).

Afterwards, the content of the key (*tempID*) for the access is retrieved from the received message (*message*). The helper `soapReturnXMLEntity` (line 13) is used to get the key for the database (its key was stored previously in the variable *DBKey*). However, this key can be embedded everywhere in the received message (which must be a tag of course). The link, which is created by drawing the connection between the message and the database in the template graph (see chapter 6.10 *Template Graph*), is used here to find the previously stored message.

If an entry exists for this key (line 17), all items from the data base are collected and added to the map *mapItems* (this is accomplished by the generator `pushBackStructureElements`, see listing 9.25 *Local Script \_\_pushBackStructureElements - Section 3*).

At the end, the message is sent out by calling the function `soapSendMessage` with the message type, the map containing the values and a string containing the target system as parameters (line 34). This function will get a new empty message template and fill in the content from the map (which was described in chapter 9.1.3.4.5 *IFM SOAP*).

```
1 Report '_createTBSimulatedSystem_createBehaviourMethods_SendMessage'
2 [...]
3 '
4 std::map < std::string, std::string > mapItems;
5 std::map < std::string, struct ' $DBName '_t >::iterator ' $DBName 'Iter;
6
```

```

7  bool_t    returnValue;
8  std::string tmpID;
9
10 returnValue = FALSE;
11
12 // get ID
13 tmpID = soapReturnXMLEntity ( message,
14     "' $DBKey %removeSpaces%lower ' " );
15
16 // find entry
17 ' $DBName 'Iter = ' $DBName 'Map.find ( tmpID );
18 if ( ' $DBName 'Iter != ' $DBName 'Map.end ( ) )
19 {
20 // get all items from send message, fill in values from DB entry!
21 '
22 /* get associated data structures */
23 do graphs where type = 'Templates'
24 {
25 /* find message */
26 foreach .TEMPLATE_Message; where :Message Type; = $targetMsgType
27 {
28 /* include all attached datastructures */
29 subreport '_pushBackStructureElements' run
30 }
31 }
32 '
33 // send this message
34 returnValue = soapSendMessage ( SOAP_MSGTYPE_':Message Type;%removeUC%
    removeSpaces ',
35     mapItems,
36     "' :Destination System;%removeUC%removeSpaces ' " );
37 }
38 [...]
39 '

```

Listing 9.24: Local Script `_createTBSimulatedSystem_createBehaviourMethods_SendMessage` - Section 3

The generator `pushBackStructureElements` is called for a specific message template. It traverses all attached data structures (line 4, line 16) and fills out the map `mapItems` (line 12). All entries, which are not empty, are attached to this map.

```

1 Report '_pushBackStructureElements'
2
3 /* include all attached datastructures */
4 do ~InheritanceTarget-InheritanceSource.()
5 {
6 /* get all entries (if not empty) */
7 if :1\ . Item - Name; <> '' and :1\ . Item - Type; <> '' then
8 /* find name of source target struct for finding element in map */
9 variable 'DBKey' write :1\ . Item - Name; close
10 $FoundStructName = ''
11 subreport '_getStructName' run
12 ' mapItems [ "' :1\ . Item - Name;%removeSpaces%lower' " ] = ' $DBName '
    Iter->second.' $FoundStructName %replaceUC '_' :1\ . Item - Name;%
    removeSpaces%lower ';'
13 endif
14 [...]

```

```
15 /* re-run for all attached datastructures */
16 subreport '_pushBackStructureElements' run
17 }
18 endreport
19 '
```

Listing 9.25: Local Script `_pushBackStructureElements` - Section 3

## Section 4 - Function Footer

The last part of the generator consists of calling the next function, if any outgoing transition is present, and returning to the calling function afterwards. The outgoing transition of the current object is followed and checked if the type of the target is also a behavior object. If this is the case, this function is called and its return value is composed to the locally calculated return value.

For instance, if the `store message` function is the current function (see figure 9.13 *Behavior Graph Check-In System*), the behavior relationship is followed (line 3). At its end, the attached object is evaluated (line 5) and the type is checked (line 7). In this case, the object at the end is of type `send message`. Therefore, the content of the if-clause is executed. The variable `return Value` is assigned to its current value and concatenated with the return value of the next called function. In this case, the `send message` function is called and the original received message is passed over.

The same applies at the end of the `send message` function. The next function is also of type `send message` and therefore executed. If the second `send message` block is executed, the behavior relationship is traversed and the connected object is checked. Now, the `wait for messages` object is attached only. The content of the if-clause is not executed but the current value of the variable `return Value` is given back to the previous function (line 19).

```
1 [...]
2 /* call next block (if any) */
3 do ~SystemBehaviourSource~SystemBehaviourTarget
4 {
5   do .()
6   {
7     if type = 'BEHAVIOUR_StoreMessage' or type = 'BEHAVIOUR_Erase' or type =
      'BEHAVIOUR_SendMessage' or type = 'BEHAVIOUR_ProcessUpdate' or type
      = 'BEHAVIOUR_Opaque' or type = 'BEHAVIOUR_Wait' or type =
      BEHAVIOUR_CalcNotification' then
8       // call next block'
9       returnValue = returnValue'
10      '&& handleSOAPMessage_' $msgType %removeUC%removeSpaces '_' oid ' (
      message );'
11    endif
12  }
13 }
14 '
15 #ifdef SIM_DEBUG
```

```

16  @printf ( "' $SimSystemName %removeUC ': handleSOAPMessage_' $msgType %
      removeUC%removeSpaces '_' oid ': END" );
17 #endif
18
19  return returnValue;
20 }
21 '
22 endreport

```

Listing 9.26: Local Script `_createTBSimulatedSystem_createBehaviourMethods_SendMessage` - Section 4

At the end, an entry in the log is created, stating that this function has been executed to its end. The return value, which consists of a combined status of all called functions is returned.

### 9.1.4.3 Test Data Stimulus

The abstract machine basically consists of two generators. One generator creates the header file while the second is responsible for the RTT file.

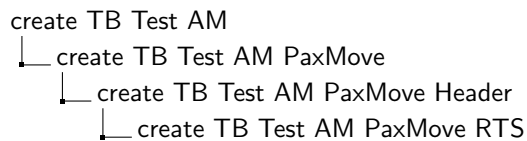


Figure 9.14: create TB Test Stimulus - MetaEdit+ Generator Tree

The goal of this simulation is to register all passengers at a certain airport, assign pre-calculated location data to each passenger and unregister these passengers afterwards. This location data needs to be injected into the systems exactly as a real guidance system would accomplish this task. In addition, this location data is also sent to MetaEdit+ in order to visualize the movement (see chapter 7 *Testing Scenario*) and to the oracle which is responsible for auditing the compliance of the requirements (see chapter 9.1.4.1 *Oracles*).

First of all, the simulation tries to read the file containing the passenger movement data which was created by the test data generator (see chapter 8.5 *Passenger Movement Data*). If the file is found, it is read and sorted ascending by timestamp. This way, it can be ensured that the list can be worked off item by item without skipping an entry.

The next step is to register all passengers if an airport layout exists for each of the passenger's flights. For accomplishing this tasks, all terminal objects are traversed. Within each terminal object all containing passenger objects are examined. As each

passenger is assigned to at least one airport, the link is followed to the attached flight database object and the airport name is extracted. Afterwards, the main graph is traversed and all containing airport objects are checked if its name is equal to the previously extracted airport name. If so, it is further checked if an airport layout graph is attached to this airport object. If this is the case, then this passenger is used in the simulation and the register commands are created.

The following task for the simulation is to inject the movement data according to the pre-calculated timestamps. The main loop is executed as long as there are unprocessed movement data.

Depending on the chosen timing conditions (Local time on test bench, Specify time within model, No Time AFAP (As fast as possible)) a different kind of framework is embedded into this simulation (see chapter 9.1.3.1 *Library Module*).

```
1 [...]
2 if :Time Switch; <> 'No Time AFAP' then
3   '   time ( & rawTime );'
4   '   timeInfo   = localtime ( & rawTime );'
5   '   actualTime = ( ( ( timeInfo->tm_hour * 60 ) + timeInfo->tm_min ) * 60
6     '               ) + timeDiff;'
7   '   actualTime = getCurrentTimeInSeconds ();'
8 endif
9
10 '   /* determine clock offset */'
11 '   /* (extracted from model) */'
12 if :Time Switch; = 'Specify time within model' then
13   /* "adjust" real time of */
14   /* test bench by offset */
15   '   timeDiff = actualTime - ( ( ( ':ScheduledDate Hour;' * 60 ) + ':
16     '           ScheduledDate Minute;' ) *60 );'
17 else
17   /* offset is set to zero */
18   '   timeDiff = 0;'
19 endif
20
21 '   // loop through list, read next event and sleep until then'
22 '   for ( iterList = timeAccessList.begin () ; iterList != timeAccessList.
23     '     end () ; iterList++ )'
24 '     {'
25 '     /* get actual time */'
26 if :Time Switch; <> 'No Time AFAP' then
27   '   time ( & rawTime );'
28   '   timeInfo   = localtime ( & rawTime );'
29   '   actualTime = ( ( ( timeInfo->tm_hour * 60 ) + timeInfo->tm_min ) * 60
30     '               ) + timeDiff;'
31 else
31   '   actualTime = getCurrentTimeInSeconds ();'
32 endif
33
34 '   // get timestamp'
35 '   if ( actualTime < * iterList )'
36 '     {'
37 '       // inform about that ( in case the sleep will take a while )'
```



```
38
39 if :Time Switch; <> 'No Time AFAP' then
40 '    @printf ( "SLEEP for [%d] sec till next event",'
41 '        ( * iterList - actualTime ) );'
42 else
43 '    @printf ( "SLEEP for [%d] sec till next event",'
44 '        ( ( * iterList - actualTime ) / 60 ) );'
45 endif
46
47 /* if test run is NOT asap, calculate time and sleep till next event */
48 if :Time Switch; <> 'No Time AFAP' then
49 '    // sleep until next event ( calculate difference )'
50 '    @rttWait ( ( ( * iterList - actualTime ) * 1000 ) _ms );'
51 else
52 '    // sleep until next event ( calculate difference -> each second
53 '        counts as a minute! )'
54 '    @rttWait ( ( ( ( * iterList - actualTime ) / 60 ) * 1000 ) _ms );'
55 endif
56 [...]
```

Listing 9.27: Local Script `_createTBTestAM_PaxMove_RTS`

After this sleep cycle, all current entries with position data for a certain passenger for this timestamp are processed. For each entry two signals are created and emitted. One signal is sent to the IFM SOAP and the second one is routed to the IFM MetaEdit+. The content for both signals are identical and contain the temporary ID of the passenger and his coordinates.

If the sending of the current set of signals is completed, the loop is further executed in case more movement data is present. If this is not the case, then the content of the test run is performed and all passengers must be unregistered from the airport. For this purpose, the same routine for identifying all passengers as for registering them is executed. Also, the unregister signals are created and emitted afterwards.

This abstract machine is the only instance which will terminate the test by calling the appropriate RTT command. All other abstract machines are executing their main loops until the termination command is triggered.

### 9.1.5 Create Test Bench Configuration

The test bench configuration consists of two types of files. One file describes the test project itself. This is the superior configuration for all tests which are organized within the project. In this case, only a small amount of information is necessary. The name of the project is extracted from the scenario graph's properties, the timers are set to microseconds and the signal definition file is declared (see chapter 13.3.3.1 *Test Bench Project Configuration File*). As there is only little information (static properties) extracted from the model, no more detailed information is given here.

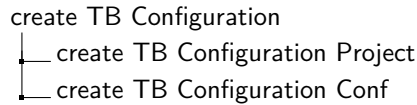


Figure 9.15: create TB Configuration - MetaEdit+ Generator Tree

The other type of file contains the configuration for a specific test. This includes the declaration of additional files, behavior when fails occur, and mainly the configuration of the RTTE.

### 9.1.5.1 Test Configuration

Only some parts of the creation process of the configuration shall be discussed here (the generated file can be found in chapter 13.3.3.2 *Test Bench Test Configuration File*), as most of the generators are rather small and only insert static information from the main graph.

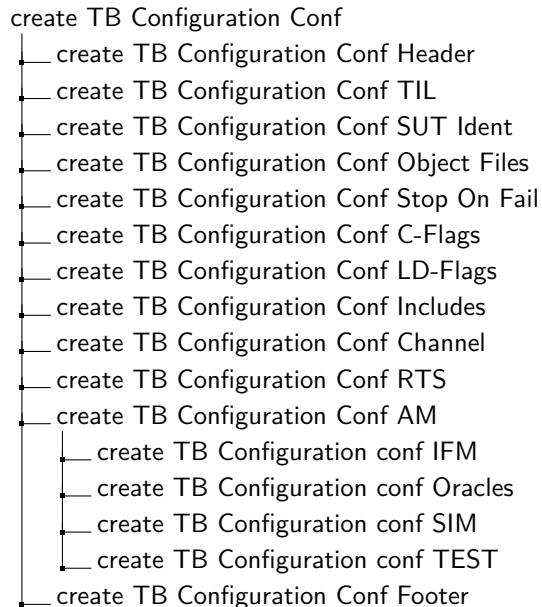


Figure 9.16: create TB Test Configuration - MetaEdit+ Generator Tree

Firstly, as described in chapter 6.2 *Main Scenario Graph*, both verbose flags are used to create or prevent debug and additional information output from the simulations, interface modules and tests. By enabling such an option, flags for the compiler are set. These flags are used by the RTT during the compilation of the target.

Secondly, all configuration information of “static” abstract machines (like IFMs, oracles, or tests) is created. As these instances are used for all derived tests from the model, and must — therefore — be present all the time. As all file names have been defined in the first tasks (see chapter 9.1.1 *Prepare local environment*), and they are also used for naming the abstract machines, filling out the configuration is a simple job.

```

1 Report '_createTBConfiguration_conf_am_IFM'
2 '/* IFM SMS Gateway */'
3 'AM ; ' $filename_TB_IFM_SMS
4 ' ; AMPROCESS ; ' $filename_TB_IFM_SMS
5 ' ; SPEC ; ' $filename_TB_IFM_SMS
6 ' ; SIGNALSET ; ' $filename_TB_SIG_IFM_SMS
7 ' ; SEED ; 23'
8 ' ; SCRATCHPAD SIZE ; 4 k'
9 ' ; ACTIVATE ; YES'
10 ' ; OUTPUTFILE ; YES'
11 ' ; OUTPUTSTDOUT ; YES'
12 [...]
13 endreport

```

Listing 9.28: Local Script `_createTBConfiguration_conf_am_IFM`

The configuration for the simulations are a bit more difficult. As the modeler decides if the original system (if present) is used or a simulation shall be incorporated into the test, the generator needs to check for the simulated flag state.

Furthermore, the file names and identifier for the abstract machines are not defined previously but are constructed depending on its state and name. The name-giving for each simulation is defined as follows. Each name starts with the prefix “sim” followed by the name of the airport (in lower case letters). The name ends with the full name of the function (also in lower case letters and spaces are also removed).

In order to collect all simulations, the graph is traversed from all existing airport graphs (line 4, listing 9.29 *Local Script \_createTBConfiguration\_conf\_am\_SIM*) and descends into each embedded test bench configuration, if such an object is present (line 11). If such a graph is attached (line 13), each present application configuration is evaluated (line 16). Depending on the simulated system flag (line 18), a configuration statement is created (lines 28–35).

```

1 Report '_createTBConfiguration_conf_am_SIM'
2
3 /* Check if there are any simulated systems */
4 foreach .Airport
5 {
6 /* store name of current airport */
7 $AirportName = :Name;%removeUC%removeSpaces
8
9 do decompositions
10 {
11 foreach .TestBenchConfiguration
12 {
13 do decompositions

```

```

14 {
15 /* create configuration entry for every simulated system */
16 foreach .ApplicationConfiguration
17 {
18   if :Simulated System; = 'T' then
19
20     /* found simulated system */
21     variable 'SimSystemName' write
22       'sim_'
23       $AirportName %lower
24       '_'
25       :Full Name;%removeUC%removeSpaces%lower
26     close
27
28     'AM ; '$SimSystemName
29     ' ; AMPROCESS ; '$SimSystemName
30     ' ; SPEC ; '$SimSystemName
31     ' ; SEED ; 23'
32     ' ; SCRATCHPAD SIZE ; 4 k'
33     ' ; ACTIVATE ; YES'
34     ' ; OUTPUTFILE ; YES'
35     ' ; OUTPUTSTDOUT ; YES'
36
37     do decompositions
38     {
39       $cnt = ''
40       foreach .BEHAVIOUR_CalcNotification
41       {
42         $cnt ++
43         if $cnt = '1' num then
44           'AM ; '$SimSystemName '_worker'
45           ' ; AMPROCESS ; '$SimSystemName '_worker'
46           ' ; SPEC ; '$SimSystemName
47           ' ; SEED ; 23'
48           ' ; SIGNALSET ; '$filename_TB_SIG_SIM_NotificationSystem
49           ' ; SCRATCHPAD SIZE ; 4 k'
50           ' ; ACTIVATE ; YES'
51           ' ; OUTPUTFILE ; YES'
52           ' ; OUTPUTSTDOUT ; YES'
53         endif
54       }
55     }
56     endif
57   }
58 }
59 }
60 }
61 }

```

Listing 9.29: Local Script `_createTBConfiguration_conf_am_SIM`

The behavior “Calc Notification” needs to be handled differently as a second abstract machine is needed. If such an object is embedded in the current behavior graph, a “worker” thread is added. This is due to the fact that two main loops are necessary: one loop is listening for RTT signals while the second loop needs to listen for incoming TCP connections.

## 9.1.6 Create Target on Test Bench

The target is built by creating sever helper scripts, copying all necessary files to the test bench and executing helper scripts there.

### 9.1.6.1 Helper Scripts

Four helper scripts are generated and written to the temporary directory. These four scripts prepare and install the test.

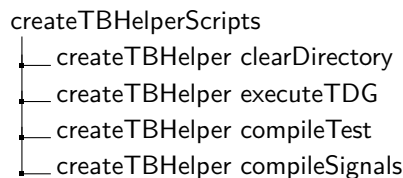


Figure 9.17: Generator Tree - createTBHelperScripts

By creating these scripts, it can be ensured that the target will be created without errors. All paths and filenames have be defined and can be used within the scripts. In the next step, these scripts need to be transferred to the test bench and executed there. Some of these scripts are only performing one litte job which is not really sophisticated. However, it eases the process of creating the target by removing every manual intervention during the process.

In order to ensure a clean startup, all associated files and directories are deleted. Afterwards, the directory structure is generated from ground-up so that an empty set of directories is present.

```
1 Report '_createTBHelper_cleanDirectory'
2
3 /* create channel description */
4 filename $filenameTBHelper_executeCleanDirectories write
5
6 '#!/bin/bash'
7 '# set the RTT_TESTCONTEXT'
8 'export RTT_TESTCONTEXT="" :TB RTT Testcontext; '''
9 '# clean simulation'
10 'rtt-clean-test ' $version ' --global'
11
12 '# clean all old test directories'
13 '# clean conf directory'
14 'if [ -d "' $path_remote_TB_Conf '" ]; then'
15 '  rm -rf "' $path_remote_TB_Conf ' "'
16 'fi'
17 [...]
18 '# create empty directories'
19 'mkdir "' $path_remote_TB_Conf ' "'
20 'mkdir "' $path_remote_TB_Specs ' "'
```

```
21 'mkdir "' $path_remote_TB_INC '"'
22 'mkdir "' $path_remote_TB_SIG '"'
23 'mkdir "' $path_remote_TB_IMF '"'
24
25 close
26 endreport
```

Listing 9.30: Local Script `_createTBHelper_cleanDirectory`

Another script executes the test data generator. Also, the knowledge of the paths and filenames are used to create an executable script which fits exactly to the test bed.

The third script creates the Makefile for compiling the signals. This is used by the next script which compiles the target.

As the modeler is not necessarily working on an Unix environment, it must be ensured that the file contains the right line feed format. The last script converts all copied files (test, test bed and intermediate files) from DOS to Unix format. Afterwards, the script compiles everything to the final test executable.

```
1 /* create channel description */
2 filename $filenameTBHelper_executeCompileSimulation write
3
4 '#!/bin/bash'
5 '# compile the test and the corresponding framework'
6
7 '# set the RTT_TESTCONTEXT'
8 'export RTT_TESTCONTEXT="" :TB RTT Testcontext; '''
9
10 '# fix files before further processing'
11 'if [ -d "' $path_remote_TB_Conf '"' ]; then'
12 '  dos2unix ' $path_remote_TB_Conf '/*.conf'
13 'fi'
14
15 [...]
16
17 '# execute signals compile'
18 'if [ -d "' $path_remote_TB_SIG '"' ]; then'
19 '  cd ' $path_remote_TB_SIG
20 '  make'
21 'fi'
22
23 '# execute simulation compile'
24 'rtt-compile-test ' $version
25
26 close
27 endreport
```

Listing 9.31: Local Script `_createTBHelper_compileTest`

### 9.1.6.2 Create Target

The last step transfers all generated files to the test bench and executes the helper scripts which are created in the previous step.

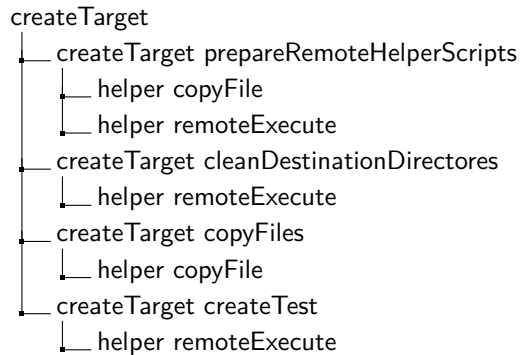


Figure 9.18: Generator Tree - createTarget

First of all, the helper files are copied, the line feeds are adjusted and the file permission flags are set to enable its execution. Afterwards, the script to clean all target directories is executed. The next step copies all other files to the new created directory structure. At the end, only the set of files which are necessary for this test are present in the destination directories on the test bench.

The final steps are to invoke the test data generator script and execute the compile test script.





---

---

## CHAPTER 10

---

# Test Results

The overall test scenario consists of four passengers which use the guidance function for a specific airport perimeter (see chapter 7 *Testing Scenario*). Two passengers are simulated so that their movement at the airport is controlled by the movement simulation. One passenger (Mrs. Jolie) is specified to reach the airplane in time while the second one (Mr. Lennon) is too late and will miss his flight.

Furthermore, two more passengers will be present (Mr. Bond and Mr. Mustermann). They are not set to be simulated so the position updater tool is used to alter their position on the perimeter.

All except Mr. Mustermann opted in for the guidance system and they expect to receive notifications on their mobile devices.

As the scheduler for this test is set to “as fast as possible” (see chapter 9.1.3.1 *Library Module*), the movement of the passengers is faster as real time. Therefore, the messages arrive shortly after each other. This effect can be noticed by looking at the time stamps on the provided screen shots of their mobile devices.

### 10.1 Course of Action

After all further necessary files have been created and deployed to their final destination on the test bench, the target is compiled. The last step is the starting of this test scenario by initiating the appropriate RTT command.

The test starts and the simulation will register all four passengers at the airport Amsterdam. For each passenger, a pin is displayed on the airport layout graph to indicate the current position for each passenger individually.

Mrs. Jolie, which is driven by the movement simulation will start walking and proceeds from the starting area to the aircraft. Each time she enters a new area, she receives a message. The second simulated passenger (Mr. Lennon) remains at the starting position and starts walking later as he is defined to be a ‘too late’ passenger.

Mr. Bond and Mr. Mustermann are not positioned by the movement simulation. Instead, the update position tool is used to place them on the airport graph.

After the simulation has made all position updates for Mr. Lennon, it unregisters all passengers from the airport and the test terminates. The oracles calculate the overall test result and the log files are finalized.

### 10.1.1 Mrs. Jolie

Initially, Mrs. Jolie opts in for the guidance function and requests notifications. As she arrives at the airport, the system recognizes her and sends an initial message. This is displayed on the lock screen of her smart phone (see figure 10.1 *Mrs. Jolie - Message on lock screen*).

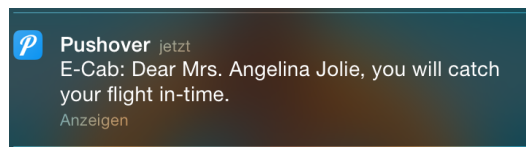


Figure 10.1: Mrs. Jolie - Message on lock screen

As she walks further through the perimeter, she receives notifications each time she enters a new airport area. Each message is stored on her device and can be viewed by her (see figure 10.2 *Mrs. Jolie - received messages*).

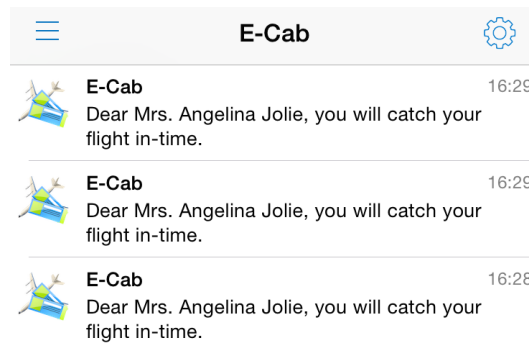


Figure 10.2: Mrs. Jolie - received messages

She decides to switch of the notifications as she arrives in the last area right before the gate. Therefore, she uses her mobile device, opens a web-browser and calls the E-Cab website. She enters her name and password and chooses the option to prevent further notifications (see figure 10.3 *Mrs. Jolie - disable notifications via webservice*). Afterwards, she receives a confirmation for the new settings (see figure 10.4 *Mrs. Jolie - confirmation of settings*).

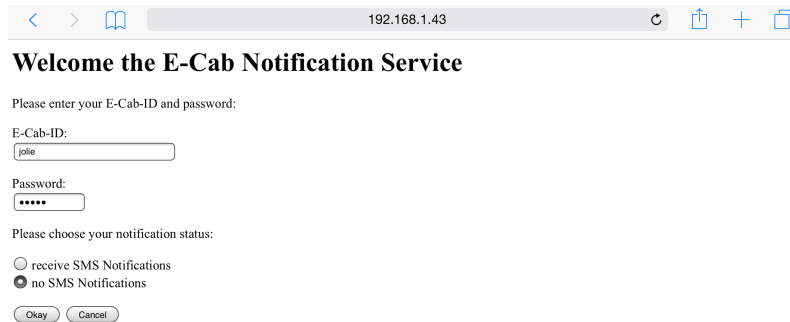


Figure 10.3: Mrs. Jolie - disable notifications via webservice

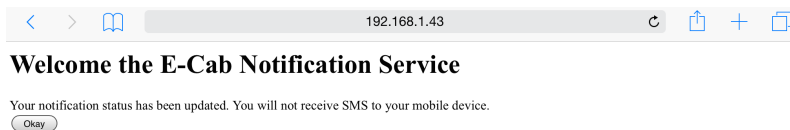


Figure 10.4: Mrs. Jolie - confirmation of settings

### 10.1.2 Mr. Bond

Mr. Bond decides to use the guidance function and therefore receives a notification as he enters the first area (see figure 10.5 *Mr. Bond - first notification on lock screen*). The update position tool is used to simulate the passenger movement of Mr. Bond. It is used to place him in the last area — and not any further.

As he is waiting there and the time of his flight departure is approaching, he receives notifications that he needs to proceed to the gate. Because he is waiting for too long and does not move any further, he is too late for catching his flight. The notification system notices this and sends him a message that he is too late (see figure 10.6 *Mr. Bond - all messages on lock screen*). All these messages appear on the lock screen of his device.

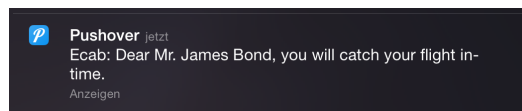


Figure 10.5: Mr. Bond - first notification on lock screen

### 10.1.3 Mr. Lennon

Mr. Lennon also opts in for the guidance function. Unfortunately, something happens on his way to the airport and he arrives much too late for his flight. The first — and



Figure 10.6: Mr. Bond - all messages on lock screen

also last — message he receives from the notification system tells him that he will not be in time to reach the aircraft before it departs (see figure 10.7 *Mr. Lennon - first and last message*).

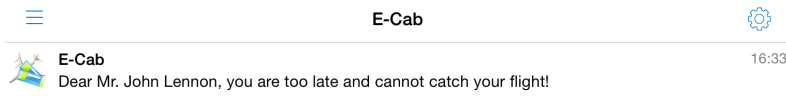


Figure 10.7: Mr. Lennon - first and last message

### 10.1.4 Mr. Mustermann

Mr. Mustermann knows the airport perimeter and therefore opts out of the guidance system right from the beginning. He will therefore not receive any messages, regardless if he is in time or if he is running late. Also, he does not change his mind while being at the airport and does not use the web service.

## 10.2 Results

When the test finishes, a short summary is displayed. Both oracles evaluate to *pass* which will lead to the overall test result *pass*.

```

rtt-get-verdict: verdict of test 'e2echain' computed.
## CONTENTS OF e2echain/testdata/VERDICT.txt
-----
ifm_sms | WARNINGS: 0, FAILURES: 0, VERDICT: ---
ifm_notificationsystem | WARNINGS: 0, FAILURES: 0, VERDICT: ---
ifm_metaedit | WARNINGS: 0, FAILURES: 0, VERDICT: ---
ifm_soap | WARNINGS: 0, FAILURES: 0, VERDICT: ---
ifm_injectorproxy | WARNINGS: 0, FAILURES: 0, VERDICT: ---
am_oracle_sms | WARNINGS: 0, FAILURES: 0, VERDICT: PASS
am_oracle_req | WARNINGS: 0, FAILURES: 0, VERDICT: PASS
sim_amsterdam_acbaggagesystem | WARNINGS: 0, FAILURES: 0, VERDICT: ---

```

---

```

sim_amsterdam_checkinsystem      | WARNINGS: 0, FAILURES: 0, VERDICT: ---
sim_amsterdam_notificationsystem | WARNINGS: 0, FAILURES: 0, VERDICT: ---
sim_amsterdam_notificationsystem_worker | WARNINGS: 0, FAILURES: 0, VERDICT: ---
sim_amsterdam_ongroundbaggagesystem | WARNINGS: 0, FAILURES: 0, VERDICT: ---
sim_amsterdam_positionapplication | WARNINGS: 0, FAILURES: 0, VERDICT: ---
sim_amsterdam_profiledatastore   | WARNINGS: 0, FAILURES: 0, VERDICT: ---
am_paxmove                       | WARNINGS: 0, FAILURES: 0, VERDICT: ---
-----
total                             | WARNINGS: 0, FAILURES: 0, VERDICT: PASS
-----
rtt-run-test: [e2echain] finished with code 0.

```

The oracle `am_oracle_sms` calculates its state by comparing the actual sent messages with the intended sent messages (see chapter 9.1.4.1.1 *Verification of Transmitted Short Messages*). If it is the case that both matches, it returns *pass* as this is the case for this test scenario. In fact, the previously shown messages have been calculated by the involved systems and only the intended ones have been sent out to the passengers' devices.

The test data generator creates paths for passengers on the airport perimeter but lacks of analyzing the requirement information stored in the intermediate file (see chapter 12.10 *Test Data Generation Improvements*, 8.4 *Test Data Generator Tool*). Therefore, one waypoint which is touched by (at least) each simulated passenger is used for requirement checks. The location is positioned between two airport areas which is the only connection between both. Therefore, it must be crossed in order to move from one area to the next one. This location is marked as to be tested with at least one passenger entering it. The movement data is collected and evaluated by the `am_oracle_req` which returns *pass* as its verdict.

The `debug` flags have been enabled on the main scenario graph for this test run which leads to more debug output which is present in the console and the log files. In case of a failed verdict, these logs could then be analyzed. Each sent SOAP message is displayed and written to disk. As it is decided to use the same temporary ID for all systems for one passenger, the flow of information can be comprehended (see chapter 12.8 *Temporary ID*).



---

---

## CHAPTER 11

---

# Evaluation and Conclusion

In this chapter, the evaluation of the realized work is depicted. This includes considerations about the used approach and its implementation. Furthermore, observations, trials and the evolution of the way to achieve the objective is presented.

### 11.1 Approach

Unfortunately, the testing in this work was done using DSL only and no equivalent “traditional” testing was conducted at the same time. A reliable comparison of both approaches (End-To-End Chain and “traditional”) regarding effort, time or money cannot be stated for this project.

However, an educated guess is possible. All domains and parts of the project were under development. Of course, as many new technologies were developed and integrated, not everything was successful from the start. There were drawbacks and adjustments throughout the complete project phase. Each stakeholder refined his own parts which also had impacts on tasks of other project partners.

This results in the fact, that most parts of the project were not stable for quite some time. This also means that not all parts of the equipment were either present or fully operational. By using the presented approach, it was simple to switch between original and simulated equipment.

Furthermore, the content of the transferred messages were not stable and fixed during the project. In a conventional approach, several parts of the test bed, test software, test bench configuration and simulations would have had to be adjusted manually. The IFM, which translates internal signals to external messages, would have had to be modified, as entries of the XML body of the SOAP message were deleted or added. Also, the type of the content was changed for some items.

Each simulation needs the description of the message in order to insert or extract the necessary information. This means that at least three parts of a simulation need an update: the message definition, the parser of the incoming message and the creator of the outgoing message. Therefore, each simulation has to be changed as well.

All these tasks would have needed to be performed manually. This means that all

the necessary files would have been altered, configured in the repository system and afterwards compiled into a target. It is an error prone task to keep several files in manually sync (adjusting the content of messages), so it can be assumed that re-work is necessary after the first editing.

In contrast to all these steps the adaptation of a new message definition for the model was a simpler task. The data structure was changed and the content of the associated message was updated automatically. By creating a new target, all necessary tasks (updating test bed, test software, test bench configuration and simulation) are performed within a few minutes.

In conclusion, this example shows that there are huge benefits from using the presented End-To-End Chain approach in contrast to the “traditional” one. Therefore, it can be assumed that time was saved using this attempt.

## 11.2 Implementation

The implementation of the DSL, the concrete model and the creation of the generators was done in an iterative approach. This means that it was not possible from the start to create a final architecture which was fixed until the end of the project. As mentioned before, many systems were under development which resulted in a constant change of interfaces and message content.

As the testing phase is at the end of a development cycle (in the sense of the processes described by Airbus Directives ABD100/ABD200 or DO178B, this might not be the case for agile development), all delays from the previous phases disrupt the planned test sessions. This means, that each mitigation — in the sense of saving time — is welcome.

Due to the approach to define everything in one place and re-use this data in different target files, the task of synchronization is limited to a minimum. For instance, in a traditional approach, each change of the payload of a message results in changing all interface modules, the passenger movement simulation and all simulations. Each file must be updated to reflect such a change. It’s a great effort to keep all files in sync. And again, as this is a manual task, introducing typos and errors is common practice.

In contrast, such changes are really simple. As all messages are defined in the template graph, only the messages need to be updated. As everything else is generated from this information automatically, no manual synchronization effort needs to be spent.



## 11.3 Model Checks

The most time consuming — and frustrating task — if code is generated from a model, is during compilation time. Each reported fault from the compiler must be traced manually to the code generator and from there back to the model. This means that at least two sources of errors exist: either in the model or in the code generator.

Of course, all errors in the generated source code (typos, wrong parameters, wrong usage of symbols, etc) originate from an error in the generator which is responsible for this piece of code.

The first step is to find the root cause for the compilation error. Not every error message can be mapped directly to the faulty position. In some cases, the whole function needs to be analyzed. This is mainly the case if missing braces or commas are present. The error is likely presented at a completely different location.

The next step is to find the associated faulty commands in the code generator. Depending on the organization of the generators, locating the faulty generator can be considered an easier task, assuming some kind of logic and structuring has been done.

If the generator is faulty in the sense that it just produces wrong code as braces are missing or typos are present, the appropriate section needs to be fixed and the next iteration of compiling the target can be initiated.

It could be the case that the generator produces syntactically correct code but the model is unfinished. This means that the generated program is missing some logic. For instance, if a simulated passenger object is not connected to a flight object, the generator would not know the time of the departing aircraft. This information would then be missing the generated code or intermediate file and no passenger movement data could be created. Also, if the IP addresses and ports of the airport systems do not differ from each other, code could be generated which would also compile. The fault would be detected during runtime, if only one system can start its server port while the other system instances would claim that they could not open their ports.

All these examples lead to the idea to avoid as many of these problems as possible. By introducing checks right at the beginning, it can be assumed that these types of faults will not make their way into the source code. Even if not all faults can be avoided, reducing the task of fixing such occurrences is worthwhile.

Two types of checkers have been evaluated and tested. The first method checks the model when the *create target* button is pressed. Before the files are created, the generators traverse each graph and check its content. While this approach is simple,

it was rejected as this is uncomfortable for the user. By clicking the button, the modeler expects the model to be finished so that the target can be created. Instead, the checker informs the modeler that this is not the case and direct him through the graphs and points at errors.

The other method is to introduce so called live checks. Such tests are performed by checker objects which are placed on every graph. Each checker examines the current graph and gives intermediate feedback to the modeler. So if two identical IP addresses are inserted, an error message is calculated instantaneously. This approach was found to be more efficient and less frustrating<sup>1</sup> than checking everything just once before the create target process.

These observations resulted in the airport layout checker (see chapter 6.7.11 *Airport Layout Checks*) which was refined to the generalized test bench object (see chapters 6.11.2 *Test Bench Configuration Status Object*), 11.7 *Evolution of Objects*).

## 11.4 Organization of Generators

In order to trace back and forth from the code generator to the source code, a mapping needs to be established. This is an additional outcome from the experience with faulty code during compilation time due to the lack of existing checkers within the model at the beginning (see chapter 11.3 *Model Checks*).

Also, this tracing is needed if a wrong functioning of one component is detected and the behavior of this entity needs to be changed. As code generators can be called from within code generators, some kind of structure is necessary to identify the root for some generated code.

The first attempts were on a file basis. One larger code generator produces one file. In addition some embedded generators produce smaller parts. These are necessary as looping and recursion might be necessary to complete such generator tasks. Within such an approach, no direct mapping from generator to code — or vice versa — can be accomplished easily (see left part in figure 11.1 *Organisation of Generators - First and Second Approach*).

During the project, this approach was refined. At least one generator produces one piece of source code (see right part in figure 11.1 *Organisation of Generators - First and Second Approach*). Of course, the embedded generators for loops and recursion are also present but are attached specifically for parts of an associated function.

This reorganization leads to smaller and more specific generators. By using some “wrapper” generators which only call other generators, a tree-like structure can be

---

<sup>1</sup>This is the personal opinion of the author as no survey was conducted.

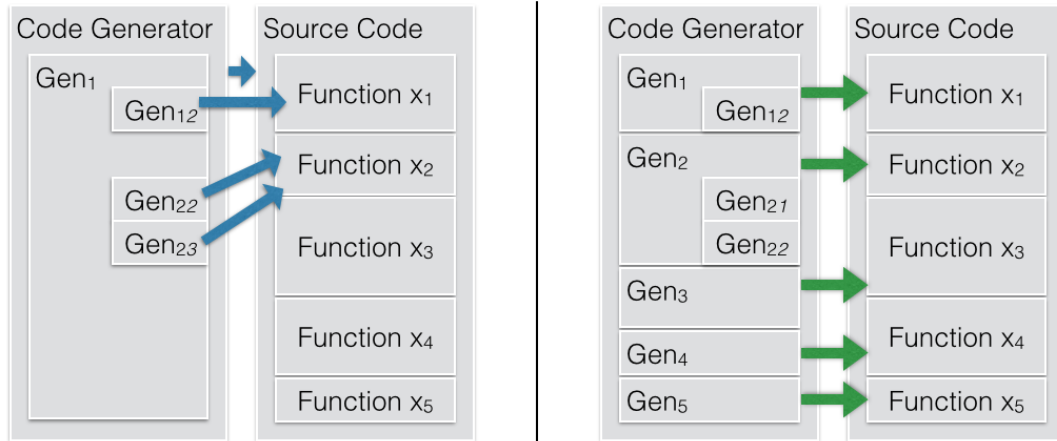


Figure 11.1: Organisation of Generators - First and Second Approach

established. These trees are organized in sections, subsections, and so on. Each source code function can now be easily found in the generator trees. These trees have been shown in chapter 9 *Creation of the Target*.

## 11.5 Handling of Generators

Each of the internal generator can be called from every other generator. As some generators implement generic functions which will be used by different components, a re-use of generators can be achieved. This reduces the work to maintain several different blocks of code. On the other hand, it can also be hard to ensure that a change in such a re-used component does not have a negative impact on all calling generators. MetaEdit+ allows such a re-use but it might not be easily visible where such a generator is used.

For instance, a generator can also be called from an embedded generator inside a part of a compound of the concrete syntax of an object. A change in an underlying generator might therefore not be clearly visible. Therefore, a comment should be placed on this kind of general generators, explaining, where this will be used (`/* This generator is used by x, y, and z*/`).

Within this model, there are generators which are used from within checks, as constraints for displaying items of objects, and for creating target code. The usage of such comments eases the handling and extension of such generators.

This method was not used from the beginning but after some time it turned out that this is absolut essential. Due to the large amount of generators and its re-use, it was inevitable that errors occurred and operative generators were accidentally broken —

which was not detected immediately but mostly during compilation of the created target.

## 11.6 Variables

All used variables follow a naming scheme. It starts with a type which could either be the name of a graph or module, followed by the name of an object or group. The last part is the description of the content of the variable. For instance, the variable *GRAPH\_Test\_VerboseFlag* is a variable which is used as a property on the main graph and it belongs to a set of variables which are used for testing activities. If set, an additional verbose flag is set. Another example is the variable *TEMPLATE\_DataBase\_Name*. This variable obviously belongs to the graph containing the templates, is part of the database object and holds the name of this database.

This naming scheme was important as the list of all variables is presented in an alphabetically sorted list. If a new object is created and variables are attached to it, it gets more complicated to choose the right variable. If only short names are used, like *name*, it cannot be determined if the right variable is used. This is also even more important as the chosen variable has a regular expression attached to ensure a proper format — and this format is only valid for a certain type of object.

During the first modeling phase, where the first parts of the model were brainstormed, only short variables were used. But in a very short time, it was clear that this will lead to confusion. The creation and usage of a clear naming scheme was the outcome.

As the list of variables is sorted alphabetically, all variables which belong together are displayed in a row. This eases the handling of larger sets of variables.

## 11.7 Evolution of Objects

During the development of the airport layout graph — containing the airport location objects and the transitions between them — the necessity of introducing a checker was clear from the beginning. Without any doubt, the graph cannot be translated to code if pieces are missing or are not correctly connected. This would not lead to a well-formed graph.

For instance, it might not be seen directly that an airport location has only incoming transitions but no outgoing transitions. This would mean that a passenger could enter this location — and never leave it. Although this would result in proper code and can be executed, this behavior is not intended.

An automated check needs to be implemented and the result needs to be reported

to the modeler. Such a graph ready status, together with some more information, was incorporated in the airport layout info object (see chapter 6.7.9 *Airport Layout Info Object*). This object is part of the airport layout graph. When introduced, it provided the necessary information to the modeler.

As other graphs evolved, it became clear that the concept of such a checker object is needed for every graph type. The checks itself cannot be generalized as the verification is specific to the specific graph and contained objects. The checks for one graph cannot be reused in another graph.

This means that for each graph, a specific set of checks needs to be implemented. However, the results can be subsumed in the same way. Three different states were introduced for the original object which are displayed as a traffic light. The color green was used if all checks evaluated to *pass* while red was displayed if at least one of these checks failed. If the checks were not conducted, the color yellow was displayed.

This concept was reused for the next iteration of such a checker object. The new introduced test bench configuration status object (see chapter 6.11.2 *Test Bench Configuration Status Object*) uses this visual display to indicate the measured state.

Several check generators were implemented for each type of graph. During modeling phase, the red indication was displayed most of the time as the model was under development and therefore was not ready. Although this indication was helping, it was unclear many times which object or transition was not properly defined. A mechanism was needed to inform the modeler about the currently failed check. Therefore, a text field was added which should display a short text describing the faulty object and hint why some more work is needed there.

In the end, the test bench configuration status object was created as described in this work (see chapter 6.12.4 *Check Model*). It displays the current status of the graph and gives a hint about a detected problem.

The evolution from the original object to the newly created one is common as the DSL is refined and used. These steps are considered as intended (see chapter 3.10 *Evolution of DSL*).

Although, the original object could have been removed and replaced by the new object, it was explicitly intended that both objects should coexist in order to show the evolution of an object.

## 11.8 Involvement of Stakeholders

The interaction with the stakeholder can be regarded as simpler as in a “traditional” testing approach. Conventionally, test specifications are written, which results in a more or less readable, longer description of all possible cases of inputs which have to be stimulated in order to trigger a specified output. The tester needs to explain why this document reflects exactly the stakeholder’s system.

By using a DSL approach, the stakeholder recognizes his own system. This means that a discussion does not take an indirect route via a document that the stakeholder is not familiar with. Instead, his own world is used to depict a scenario. By speaking the same language, communication is much easier and misunderstandings can be cleared up quickly.

This common language approach was appreciated by the involved partners during the project.

## 11.9 Test Run

Due to the fact that all systems were under development during the phase of the project, the test environment might not be complete for scheduled test runs. In a conventional approach, such test runs could not have been executed.

By using the concept of switching between original and simulated equipment, such a problem is not serious anymore. If the original equipment is present and the system’s supplier wants to integrate the system into the test environment only some minor tweaks in the model must be done. Also, if this equipment is removed, switching to a simulated environment is as easily and quickly done as integrating the original one.

## 11.10 Conclusion

The author has shown a complete procedure and workflow to test End-To-End Chains. In the E-Cab project, a special tailored DSL was developed and refined during the testing phase. Due to automated testing, test results can be obtained quickly even after changes in the test environment.

The DSL was fitted to the project. This means that the structure of the project is reflected in the language. Each stakeholder identified himself and his own work packages on the storyboards.

The concrete syntax of the DSL was developed according to rules of visual language design. This includes all elements: design of objects, their relations and the graphs.

The basics of such a visual notation (e.g. semiotic clarity, perceptual discriminability, visual distance, perceptual popout, icons and redundant coding) were applied to ensure a quick understanding and avert misinterpretation.

Live checks were implemented also to ensure that the model is well-formed. This prevents unfinished models which cannot be further processed. A concept for implementing such constraints was defined from the beginning and refined during the project phase.

It is shown in this work that the generation of simulations, test bed, test data and test stimuli can be done without changes in the given workflow. This means that this approach does not disrupt an existing life cycle but can replace traditional test means.

Also, the benefit is that the systems under test and even the test environment can be further developed and extended. The only task for the tester is to adjust the DSL and re-run the new generated test cases. No manual intervention is needed.

Another benefit is the easy change of the storyboard and therefore a complete change of the configuration of the test bench. When a new scenario is loaded, the configuration is done automatically. No wiring needs to be changed for a complete different scenario. No original components need to be extracted from the original environment and wiring to connect it to the test bench (and be put back after the test).

As the resulting source code is generated, coding style guides were applied which are necessary for further certification activities.

The outcome of all these activities results in an improved uptime and temporizes the real testing activities. Furthermore, as many manual process oriented checks and tasks are accomplished automatically, time — and therefore money — is saved.





---

---

## CHAPTER 12

---

# Outlook

Although many topics have been covered in this work, there is always room for improvement. Some topics have been touched to prove the concept but additional work can be spent on refining.

Some thoughts on the potential growth are depicted in the following sections.

### 12.1 Extending Model Checks

The checks of the semantic correctness of the model should be extended (see chapters 6.11.2 *Test Bench Configuration Status Object* and 6.12.4 *Check Model*). At the moment, the checks are limited in the sense of proving the operative readiness of the concept.

The checks could be extended to test whether the behavior graphs are semantically correct. Currently it is not tested if one type of incoming SOAP message is processed by exactly one path of the graph. For instance, if the register profile data store message (see figure 7.4 *Behavior Graph - Profile Data Store*) is handled by two different transitions with different behavior objects, the produced code would contain two `CASE` clauses with different statements. It would be up to the compiler to either complain about this occurrence only or abort the compilation. If this code would be compiled and executed, the modeler would not know which transition is really triggered.

### 12.2 Detection of Faults during Run-Time

Within this work, no major focus has been laid on the robustness of the transmitted messages with respect to reliability. Within real-world applications, more error detection needs to be designed and implemented so that missing messages or erratic values don't lead to inconsistent behavior or dead-locks. This can be done by applying protocols between systems (e.g. a value is regarded to be confirmed if two consecutive messages carry the same value or declare a value as valid (see chapter 4.3.4.1 *Door-Status-Controller Example*)).

The next step would be to identify the faulty component in the network. If one — or even more — systems did not act according to such a protocol, it would force all other connected systems into their “error detection and repair” mode. Even if this would not influence the processing of data, it consumes (at least) more CPU resources than needed for “normal processing”. As a consequence, the faulty system should be identified in order to ensure a well-working network of systems.

In order to identify the faulty system, the transmitted messages on the network need to be analyzed. Furthermore, a fault-tree analysis must be established for each system (which should be present anyway for such systems before they “go live”). Typically, the five most common types of faults (bridging faults, signal deletion errors, stuck-at-0 faults, action failure or transition missing) can be detected during such an analysis. By falsifying each fault hypothesis, the result contains the faulty system which needs to be “blamed” and the fault which has to be corrected in a following step [27].

## 12.3 Avoiding Code Smells

The code generator can be extended so that more “code smells” are avoided. For instance, if a large amount of simulated passengers are defined in the model and a block of source code for each passenger is created, this would result in a long function (`long method`) which contains these blocks (see chapter 3.8.1 *Smells and Design Practice*). The code generator must then comment this function according to the coding standard (see chapter 3.8.2 *Coding Style Guides*). Also, the code generator could create several functions each for a couple of passengers. A wrapper function could then call all these generated smaller functions — although this might contradict the rule of avoiding `lazy classes` (see chapter 3.8.1 *Smells and Design Practice*).

## 12.4 Integration to Requirements Database

In order to track the requirements, which are attached to the objects, a generator could provide this information for other tools (see chapter 4.3.2 *Documents and Requirements*). The requirement ID can be extracted from the model and exported via an intermediate file. The external requirements tool (requirements database, e.g. DOORS) can read this file and store the content (ID).

At the end of the test, the requirements oracle will check the results. If the oracle is also extended to provide these results as an input file for the requirements database, the results can be attached to the beforehand imported ID.

It is important to mention that two files need to be created. First, all requirement IDs from the model need to be provided to the requirements database. The oracle itself is part of a test run and might not contain all IDs from the model as this could be a sub-set of the complete model. Therefore, the oracle needs to provide its results to the database as well.

This requirements database can afterwards be used to prove that all requirements have been tested or that test gaps still exist.

## 12.5 Change Protocols

The messages which are transferred via SOAP are currently sent in clear text (see chapter 13.2 *SOAP Message Examples*). This is obviously a good idea if concepts are tested and prototypes are created.

For real-world applications, this is not acceptable. First, the content of each message should be encrypted so that only communicating systems are able to read the data. Also, authentication between these systems needs to be performed in order to prevent so called man-in-the-middle attacks.

For future releases, it can be discussed whether SOAP and XML is the best solution for machine-to-machine communication. XML was invented to gain a human readable format. While this helps during integration and development, all element tags and structure lead to large messages. This additional overhead needs to be transferred via network. As each network has a certain bandwidth, the amount of transferred messages is limited. The larger the message, the less can be transferred.

On the other hand, XML can be a preferred language as checks can be performed if the XML-document is well-formed. Such a rule-set can be established between stakeholders so that the same checks are performed.

At last, it can be discussed if the payload of the message can be compressed while being exchanged. This would also lead to a leaner usage of the provided bandwidth.

## 12.6 Web Server

As the web-server does not play a greater role in this work, it is a simple service which provides not more than three different pages. It is embedded in the IFM notify system which is a part of the test bed. If this service would be used by a large number of clients, this concept must be revised. Currently, no load-balancing nor threading is used which would be a must-have.

The login and password are not encrypted via transfer as only HTTP is used as transport protocol. In a future version, this needs to be extended in order to protect the user's privacy and ensure that identify theft is prohibited.

## 12.7 Principle of Data Avoidance

The principle of data avoidance and minimization should be reiterated with respect to the content of the transferred messages. It should be checked, if this amount of data does need to be sent and if its possible to reduce the amount of data to the absolute minimum. This would not only protect personal data but is also economic as only reduced datasets are stored and processed.

For instance, with the current design the check-in system receives the same amount of data as the profile data store system. The check-in message is relayed to the profile data store by the check-in server. It can be considered to split the content of these two messages so that the information is only sent to the systems which explicitly need this kind of data.

## 12.8 Temporary ID

The concept of the usage of temporary ID needs to be refined and enlarged. At this point of time, the same temporary ID is used for a passenger by all systems. For this proof-of-concept, this 1:1 mapping was explicitly wanted so that messages — and therefore associated passengers — could easily be identified and kept apart.

However, this contradicts the usage of temporary ID. Each system should generate a new temporary ID which is assigned for the outgoing messages. The internal mapping of received ID and outgoing ID needs to be done within each system. If an action is accomplished, the ID and the entry in the internal mapping table needs to be deleted. By using such an approach, tracking is not possible between systems anymore.

## 12.9 Code Generator

Starting from the fact that the generated code is syntactically and semantically correct, coding smells were avoided and design guidelines were followed (see chapter 3.8 *Code and Design Smells*). The code — and therefore the code generator — might be far away from perfect and can still be improved.

For instance, due to generalization of transforming model to code, the source might not be as fast during runtime as it could be. This has mainly two reasons. First, the code is not elegant in a sense that the expressive power of the target language is not fully utilized. This means that a simpler code generator will translate the model to a simple and not sophisticated code. For instance, if many passengers need to be created, the current implementation of the code generator consists of a loop which results in a block of steps for each passenger. These blocks would then be executed step-by-step. If the code generator would check, how many blocks it would create, it could use target language constructs and try to parallelize such blocks. The code generator could even calculate the optimum of such a parallelization. If the target will run on a multi-processor system, the code generator could take this information into account and create balanced code blocks for each processor.

Second, the source code needs to be translated by a compiler to machine code. The compiler will likely perform optimizations during this step which are not regarded by the code generator. For instance, the code generator could use techniques to ensure fully aligned structures. Also, if it is known how the compiler internally treats types like a boolean and associated arithmetic procedures, the code generator could produce compiler friendly code. This is an important step if the target hardware is an embedded system which might not have high computing power capabilities.

A general approach would be to analyze the target and find the hot spots where the most time is consumed. The compiled code could be analyzed by static code analyzers which calculate and examine the runtime behavior (e.g. `AbsInt aiT [1]`). These hot spots can be lifted up to the source code and not optimal language constructs can be spotted during a review. This information needs to be incorporated into the code generator. By this approach, the code generator would then avoid these code blocks and better code would be produced.

## 12.10 Test Data Generation Improvements

The used test data generator is a rather simple one which tries to identify paths on the airport perimeter. It uses the airport intermediate format file and generates the passenger movement data.

It creates traces according to the *late* status of the passenger. If the passenger is defined to be too late and does not catch the flight, some time is added for each step he takes at the airport. This leads to the result that he will arrive at the position of the aircraft after it left already.

However, it does lack of creating a trace with the requirements tracking in mind. This information is currently not part of the test data generator. This means that for the current test run, only one waypoint is part of the requirements checking (the waypoint

between the upper and lower part of the airport perimeter, see figure 7.2 *Test Scenario Airport Layout and Areas*). This location will be at least entered once and so the oracle will evaluate to *pass*. All other locations are not marked to be checked in the model and therefore are not regarded during the evaluation phase of the oracle.

An enhanced test data generator would use this information and create more sophisticated traces for passengers with respect to the requirements tracing.

## 12.11 Design of the DSL

The current design of the objects and graphs might not be state-of-the-art anymore. The airport layout uses too much color and seems too cluttered. While the outcome of the visual notation and guidelines were followed (see chapter 3.5.2 *Visual Notation*, 3.6 *Guidelines*) and 6 *E-Cab Domain Specific Language*), the graphs seems crowded. Due to the chosen colors, patterns might not be recognized immediately and the graph seems confusing.

However, at that time it was a good idea to re-use the colors which had been introduced during the project. Each stakeholder could identify themselves as they could recognize their parts at a glance.

For the next iteration of the E-Cab DSL (or any other DSL), more value should be assigned to the concrete syntax. A possible solution could be to avoid bright colors and use similar colors for similar objects. For instance, the airport locations (see chapter 6.7.5 *Airport Location Object*) could use one main color with different hues depending on the type of location. In order to emphasize an aspect — either during modeling phase or during runtime — a special flag could be used. This flag would then be used by the embedded generator within the concrete syntax and a different color could be used. An airport location, with properties not well-formed or where transitions are not properly defined, could be displayed in a different color and hue as the other ones. During runtime, the airport locations could be highlighted where the requirements are or are not yet fulfilled.

---

---

## CHAPTER 13

---

# Appendix

### 13.1 MetaEdit

#### 13.1.1 Extensions and Patches

Two patches have been used for MetaEdit+.

The “reprCreationAPI” patch fixes a bug when objects are created via the Java API. In this work, this is used in the passenger simulation. Passenger objects are created and their representations are moved in the airport layout graph (see chapter 6.7 *Airport Layout Graph*).

The “transparentBitmaps” patch provides the usage of images with transparent areas. This is used for most of the graphical representations in this project. Without using this patch, most of the imported images will have a white frame. This might not only look inappropriate but also might cover some other widgets. The creation of graphical representations are more difficult without this patch.

Both patches can be obtained on the MetaCase Website [49].

#### 13.1.2 Start MetaEdit+

MetaEdit+ can be started with command line parameters. These credentials were used in a batch-script so that the work environment is set up quickly. The following parameters are used for the startup script.

**fileInPatch** Path to the patch file

**loginDB:user:password** Credentials to log into a specified database

**setProject** Name for the project, this will be loaded automatically

**startAPI** This starts the API handler so that external programs can make use out of the provided Java Interface

The batch script is implemented like this<sup>1</sup>:

```
1 @ECHO OFF
2 "C:\Programme\MetaEdit+ 4.5\mep45.exe" fileInPatch: "[path to workspace]\
  patches\reprCreationAPI.mep" fileInPatch: "[path to workspace]\patches
  \transparentBitmaps.mep" loginDB:user:password: [database] [username]
  [password] setProject: [name of project] startAPI
3 exit
```

Listing 13.1: Local Script startME.bat

### 13.1.3 MetaEdit Bridge Commands

The MetaEdit+ bridge listens on port *7896* on the modeler's computer. It uses a simple text based protocol to communicate with the MetaEdit+ model (via the API Tool). In this chapter, the commands are described.

**Terminalswitch** Switches on/off terminal mode. This is used for testing reasons when the modeler connects via a telnet session. If no parameter is given, *true* is assumed and set.

```
terminalswitch()
```

**Connect** Connects to a scenario which is identified by the name of the main graph. The name of the scenario is mandatory.

```
connect(<name of scenario>)
```

**Close** Closes the current connection.

```
close()
```

**Showairports** After the connection to the model is made, the internal airport data base of the bridge is filled with values from the model. By using the `showairports` command, they are displayed on the console.

```
showairports()
```

**Showpax** After the connection to the model is made, the internal passenger data base of the bridge is filled with values from the model. By using the `showpax` command, they are displayed on the console.

```
showpax()
```

**Register** This registers a passenger on the connected scenario on a specified airport. A pin is placed on the graph. The `paxID` and `airportName` are mandatory parameters. The third parameter is optional. If this is set to *true* the pin is

---

<sup>1</sup>The paths need to be adjusted and values for the database, user, password and the name of the project must be inserted.



displayed immediately as the graph is refreshed. If set to *false*, the passenger is registered but not displayed. The command `draw` needs to be called for this airport in order to display the passenger's representation.

```
register(<paxID>,<airportName>[,<displayImmediately=TRUE>])
```

**Unregister** This unregisters a passenger on the connected scenario on a specified airport. A pin is removed from the graph. The `paxID` is a mandatory parameter. The second parameter is optional. If this is set to *true* the pin is removed immediately as the graph is refreshed. If set to *false*, the passenger is unregistered but still displayed. The command `draw` needs to be called for this airport in order to remove the passenger's representation.

```
unregister(<paxID>[,<displayImmediately=TRUE>])
```

**Setposition** The position of a pin is set to the given values. The `paxID`, `x` and `y` coordinate are mandatory parameters. The fourth parameter is optional. If this is set to *true* the pin is set to the new position immediately as the graph is refreshed. If set to *false*, the passenger's new position is not displayed. The command `draw` needs to be called for this airport in order to display the new position of the passenger's representation.

```
setposition(<paxID>,<x-coordinate>,<y-coordinate>
[,<displayImmediately=TRUE>])
```

**Setstate** The state (late, in-time, too late) is set for a given passenger. The `paxID` and the `lateState` are mandatory parameters. The color of the pin changes immediately if the state changes.

```
setstate(<paxID>,<lateState>)
```

**Draw** The graph with the given name is updated. The parameter 'name of the graph' is mandatory. The graph is redrawn immediately.

```
draw(<name of graph>)
```

```
1 telnet 192.168.1.200 7896
2 Trying 192.168.1.200...
3 Connected to phdwin.fritz.box.
4 terminalswitch()
5 Terminal switch is set to true
6
7 connect(Ecab-Scenario)
8 ConnectionHandler: created: airportDB
9 ConnectionHandler: created: paxDB
10
11 showairports()
12 Airport 'Singapore'
13 AirportAreaID = 0
14 AirportObjectID = 0
15 Scenario 'Ecab-Scenario'
16 ScenarioAreaID = 3
```

```
17 ScenarioObjectID = 272
18
19 Airport 'Munich'
20 AirportAreaID = 0
21 AirportObjectID = 0
22 Scenario 'Ecab-Scenario'
23 ScenarioAreaID = 3
24 ScenarioObjectID = 272
25
26 Airport 'Amsterdam'
27 AirportAreaID = 3
28 AirportObjectID = 12018
29 Scenario 'Ecab-Scenario'
30 ScenarioAreaID = 3
31 ScenarioObjectID = 272
32
33 showpax()
34 paxID = PAXID_oid_3_10914
35 Title = Mr
36 FirstName = Max
37 SecondName =
38 SurName = Mustermann
39 Gender = Man
40 isRegistered = false
41 at Airport =
42
43 paxID = PAXID_oid_3_7983
44 Title = Mrs
45 FirstName = Angelina
46 SecondName =
47 SurName = Jolie
48 Gender = Woman
49 isRegistered = false
50 at Airport =
51
52 paxID = PAXID_oid_3_21948
53 Title = Mr
54 FirstName = James
55 SecondName =
56 SurName = Bond
57 Gender = Man
58 isRegistered = false
59 at Airport =
60
61 paxID = PAXID_oid_3_8654
62 Title = Mr
63 FirstName = John
64 SecondName =
65 SurName = Lennon
66 Gender = Man
67 isRegistered = false
68 at Airport =
69
70 register(PAXID_oid_3_7983, Amsterdam, true)
71 ConnectionHandler: PAX [PAXID_oid_3_7983] registered at airport [Amsterdam
   ]
72
73 setstate(PAXID_oid_3_7983, intime)
74 ConnectionHandler: PAX [PAXID_oid_3_7983] set late state [intime]
75
76 setstate(PAXID_oid_3_7983, late)
```

```

77 ConnectionHandler: PAX [PAXID_oid_3_7983] set late state [late]
78
79 setstate(PAXID_oid_3_7983,toolate)
80 ConnectionHandler: PAX [PAXID_oid_3_7983] set late state [toolate]
81
82 setposition(PAXID_oid_3_7983,500,500,true)
83 ConnectionHandler: PAX [PAXID_oid_3_7983] set to position [500/500]
84
85 draw(Amsterdam)
86 ConnectionHandler: Graph [Amsterdam] refreshed
87
88 unregister(PAXID_oid_3_7983,true)
89 ConnectionHandler: PAX [PAXID_oid_3_7983] unregistered
90
91 close()
92 ConnectionHandler: connection closed
93 Connection closed by foreign host.

```

Listing 13.2: ME Sample Session

### 13.1.4 MERL Reference

In this chapter, the MetaEdit+ language MERL reference is shown.

**STRING** “ ’ ” CHAR\* “ ’ ”, where CHAR is any character

a ’ character must be doubled

**NUMBER** (“0”..“9”)+

**NAMECHAR** “a”..“z” | “0”..“9” | “ ” | \_+[-[]? | INTLNAMECHAR | ESCAPECHAR

**INTLNAMECHAR** äëïöü | áéíóú | àèìòù | âêîôû | ñãœçÿ | ß€ | ¿

**ESCAPECHAR** “\” ECHAR, where ECHAR is anything that is not a letter number or underscore

**NAME** NAMECHAR+

If NAME contains a space, the whole name should have a “;” after it, or one of “.>~#” forming the start of the next element in a chainClause

**WILDNAME** [“^”] (NAMECHAR | “\*” | “#”)+

If WILDNAME contains a space, the whole name should have a “;” after it.

**ChainOutputClause** (propClauseWithLevel | propClause | (graphEltClause+ propClause)) [“;”][translatorNames] [“;”];

**TranslatorNames** (“%” <<\lt>>NAMECHAR>+)+;

**PropClauseWithLevel** propClause levelNumber [“;”];

**LevelNumber** [“;” “ ”\* [“-”] <<\lt>>NUMBER>+;

**PropClause** “:” (<<\lt>>NAME> | “()”);

**GraphEltClause** (objClause | relClause | roleClause | portClause) [“;”]

**ObjClause** “.” TypeChoiceClause

**RelClause** “>” TypeChoiceClause

**RoleClause** “~” TypeChoiceClause

**PortClause** “#” TypeChoiceClause

**TypeChoiceClause** NAME | “()” | “( ” WILDNAME “ | ” WILDNAME\* “)”

**Report** oldreport | newreport;

**Oldreport** “report” <<\lt>>STRING> clause\* “endreport”;

**Newreport** [newheadersection] clause\*;

**Newheadersection** <<\lt>>NAME> [“(” [<<\lt>>NAME> (“,” <<\lt>>NAME>)\*] “)” ];

**Clause** (comment | basicClause | ifClause | loop | subreportClause | fileClause | md5Clause | executeClause | promptAskClause | variableAssign | variableClause | translationClause | mathClause | chainOutputClause ;

**Comment** <<\lt>>comment>;

**BasicClause** atomicClause | iterableClause;

**AtomicClause** newlineClause | separatorClause | literal | variableRef | simpleClause;

**NewlineClause** “newline” [“;”];

**SeparatorClause** “sep” [“;”];

**literal** <<\lt>>STRING> [translatorNames][“;”];

**variableRef** “\$” <<\lt>>NAME> [translatorNames][“;”];

**simpleClause** (“id” | “type” | “metatype” | “oid” | “projectid” | “objectid” | “project”) [levelNumber][“;”] [translatorNames][“;”] (“x” | “y” | “left” | “right” | “top” | “bottom” | “centerX” | “centerY” | “width” | “height” | “area”) [levelNumber][“;”] [translatorNames][“;”];

---

**iterableClause** (“decompositions” | “explosions” | “containers” | “contents” | “stack” | “graphs”) [“;”];

**ifClause** “if” [condition] “then” [“;”] (clause\* | “;”) [“else” [“;”] clause\*] “endif” [“;”];

**condition** (“not” condition) | (condition “and” condition) | (condition “or” condition) | (“(” condition “)”) | expression;

**expression** comparison | unary;

**unary** comparableClause;

**comparison** comparableClause comp comparableClause [“num”];

**comparableClause** atomicClause | chainClause;

**comp** “<<\lt>>” | “>” | “<<\lt>>=” | “>=” | “=” | “<<\lt>>>” | “=~” | “=/”;

**loop** (“do” | “dowhile”) (chainClause | atomicClause) [whereClause][filterClause] “” clause\* “” [“;”] | “foreach” graphEltClause [“;”][whereClause] [filterClause] “” clause\* “” [“;”];

**chainClause** (chainElementClause [levelNumber][“;”])+;

**chainElementClause** graphEltClause | propClause | iterableClause;

**whereClause** “where” condition;

**filterClause** orderByClause [uniqueClause] | uniqueClause;

**orderByClause** “orderby” orderCriterion (“,” orderCriterion)\*;

**uniqueClause** “unique” [clause+ (“,” clause+)\*];

**orderCriterion** clause+ [“num”][“asc” | “desc”];

**subreportClause** (“subreport” | “subgenerator”) [“;”] clause\* “run” [“;”];

**fileClause** outputFileClause | filenameReadClause | filenamePrintClause;

**outputFileClause** “filename” [“;”] clause\* [“encoding” [“;”] clause+][“md5start” [“;”] clause+] [“md5stop” [“;”] clause+] modeClause clause\* “close” [“;”];

**modeClause** (“write” | “merge” | “append”) [“;”];

**filenameReadClause** “filename” [“;”] clause\* [“encoding” [“;”] clause+] “read” [“;”];

---

```
filenamePrintClause "filename" [“;”] clause* "print" [“;”];
md5Clause "md5id" [“;”] clause* "md5Block" [“;”] clause* "md5sum" [“;”];
executeClause ("external" | "internal") [“;”] clause* ("execute" | "executeBlock-
ing") [“;”];
promptAskClause "prompt" [“;”] clause* "ask" [“;”];
variableClause variableReadClause | variableWriteClause;
variableReadClause "variable" [“;”] clause+ "read" [“;”];
variableWriteClause "variable" [“;”] clause+ variableModeClause clause* [“;”]
"close" [“;”];
variableModeClause ("write" | "append") [“;”];
variableAssign "$" <<\lt>>NAME> "=" [“;”] (variableAssign | basicClause |
chainOutputClause);
translationClause "to" [“;”] clause* ["translate" [“;”] clause*] "endto" [“;”];
mathClause "math" [“;”] clause* "evaluate" [“;”];
```

## 13.2 SOAP Message Examples

In the following sections, examples from a successful test run are presented. All messages belong to the same person Mrs. Jolie with the ID PAXID\_oid\_3\_7983.

### 13.2.1 Register Profile Data Store Message

```
1 POST /registerprofiledatastore HTTP/1.0
2 Host: 192.168.1.43:3500
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 2657
5 SDAPAction: "registerprofiledatastore"
6 <?xmlversion='1.0'encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
8 >
9 <soapenv:Header/>
10 <soapenv:Body/>
11 <data>
12 <flightinformation>
13 <seatnumber>2C</seatnumber>
14 <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
15 <departuredefinition>
16 <processstatus>Checked-In</processstatus>
17 <value>NoValue</value>
```

### 13.2.1 REGISTER PROFILE DATA STORE MESSAGE

---

```
17     <sequencenumber>NoValue</sequencenumber>
18     <bookingreference>NoValue</bookingreference>
19     <ticketnumber>NoValue</ticketnumber>
20 </departuredefinition>
21 <flightlegstatic>
22   <flightidentity>AMS1900</flightidentity>
23   <portsofcall>AMS</portsofcall>
24   <scheduleddatetime>Mar-1_19-00</scheduleddatetime>
25   <flightdirection>Departure</flightdirection>
26 </flightlegstatic>
27 </flightinformation>
28 <guidancenotification>
29   <guidancestatus>Enabled</guidancestatus>
30   <notificationlanguage>UK</notificationlanguage>
31   <notificationlevel>High</notificationlevel>
32   <notificationmedia>Text</notificationmedia>
33   <trackingstatus>Enabled</trackingstatus>
34   <baggagestatusnotificationlevel>Information</
    baggagestatusnotificationlevel>
35 </guidancenotification>
36 <pax>
37   <basicdata>
38     <title>Mrs</title>
39     <gender>Female</gender>
40     <name1>Angelina</name1>
41     <name2>NoValue</name2>
42     <familyname>Jolie</familyname>
43     <birthdate>1.1.1970</birthdate>
44     <handicapstatus>Normal</handicapstatus>
45     <nationality>USA</nationality>
46     <passportnumber>012345</passportnumber>
47     <profession>Actor</profession>
48   </basicdata>
49   <address>
50     <city>NewYork</city>
51     <citycode>NoValue</citycode>
52     <country>USA</country>
53     <countrycode>NoValue</countrycode>
54     <postofficebox>55555</postofficebox>
55     <streetname>Pittstreet</streetname>
56     <streetnumber>10</streetnumber>
57   </address>
58   <contactdetails>
59     <email>jolie@pitt.com</email>
60     <fax>555-3456</fax>
61     <mobile>+4915154627398</mobile>
62     <phone>555-2345</phone>
63     <website>www.hollywood.com</website>
64   </contactdetails>
65   <flyerdata>
66     <frequentflyernumber>12345</frequentflyernumber>
67     <frequentflyerstatus>Gold</frequentflyerstatus>
68   </flyerdata>
69 </pax>
70 <tagcodes>
71   <rfidpassive>500</rfidpassive>
72   <rfidactive>600</rfidactive>
73   <2dbarcode>700</2dbarcode>
74 </tagcodes>
75 </data>
76 </soapenv:Body/>
```

```
77 </soapenv:Envelope>
```

Listing 13.3: SOAP Message Register Profile Data Store

### 13.2.2 Unregister Profile Data Store Message

```
1 POST /unregisterprofiledatastore HTTP/1.0
2 Host: 192.168.1.43:3500
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 305
5 SOAPAction: "unregisterprofiledatastore"
6 <?xmlversion='1.0' encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  >
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <paxid>
12      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13    </paxid>
14  </data>
15 </soapenv:Body/>
16 </soapenv:Envelope>
```

Listing 13.4: SOAP Message Un-Register Profile Data Store

### 13.2.3 Check-In Passenger Message

```
1 POST /checkinpax HTTP/1.0
2 Host: 192.168.1.43:3520
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 2614
5 SOAPAction: "checkinpax"
6 <?xmlversion='1.0' encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  >
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <flightinformation>
12      <seatnumber>2C</seatnumber>
13      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
14      <departuredefinition>
15        <processstatus>Checked-In</processstatus>
16        <value></value>
17        <sequencenumber></sequencenumber>
18        <bookingreference></bookingreference>
19        <ticketnumber></ticketnumber>
20      </departuredefinition>
21      <flightlegstatic>
22        <flightidentity>AMS1900</flightidentity>
23        <portsofcall>AMS</portsofcall>
24        <scheduleddatetime>Mar-1_19-00</scheduleddatetime>
25        <flightdirection>Departure</flightdirection>
26      </flightlegstatic>
27    </flightinformation>
28    <guidancenotification>
29      <guidancestatus>Enabled</guidancestatus>
```



```

30     <notificationlanguage>UK</notificationlanguage>
31     <notificationlevel>High</notificationlevel>
32     <notificationmedia>Text</notificationmedia>
33     <trackingstatus>Enabled</trackingstatus>
34     <baggagestatusnotificationlevel>Information</
        baggagestatusnotificationlevel>
35 </guidancenotification>
36 <pax>
37   <basicdata>
38     <title>Mrs</title>
39     <gender>Female</gender>
40     <name1>Angelina</name1>
41     <name2></name2>
42     <familyname>Jolie</familyname>
43     <birthdate>1.1.1970</birthdate>
44     <handicapstatus>Normal</handicapstatus>
45     <nationality>USA</nationality>
46     <passportnumber>012345</passportnumber>
47     <profession>Actor</profession>
48   </basicdata>
49   <address>
50     <city>NewYork</city>
51     <citycode></citycode>
52     <country>USA</country>
53     <countrycode></countrycode>
54     <postofficebox>55555</postofficebox>
55     <streetname>Pittstreet</streetname>
56     <streetnumber>10</streetnumber>
57   </address>
58   <contactdetails>
59     <email>jolie@pitt.com</email>
60     <fax>555-3456</fax>
61     <mobile>+4915154627398</mobile>
62     <phone>555-2345</phone>
63     <website>www.hollywood.com</website>
64   </contactdetails>
65   <flyerdata>
66     <frequentflyernumber>12345</frequentflyernumber>
67     <frequentflyerstatus>Gold</frequentflyerstatus>
68   </flyerdata>
69 </pax>
70 <tagcodes>
71   <rfidpassive>500</rfidpassive>
72   <rfidactive>600</rfidactive>
73   <2dbarcode>700</2dbarcode>
74 </tagcodes>
75 </data>
76 </soapenv:Body/>
77 </soapenv:Envelope>

```

Listing 13.5: SOAP Message Check-In Passenger

### 13.2.4 Check-Out Passenger Message

```

1 POST /checkoutpax HTTP/1.0
2 Host: 192.168.1.43:3520
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 305
5 SOAPAction: "checkoutpax"

```

```
6 <?xmlversion='1.0'encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  >
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <paxid>
12      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13    </paxid>
14  </data>
15 </soapenv:Body/>
16 </soapenv:Envelope>
```

Listing 13.6: SOAP Message Check-Out Passenger

### 13.2.5 Load Baggage Message

```
1 POST /loadbaggage HTTP/1.0
2 Host: 192.168.1.43:3540
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 305
5 SOAPAction: "loadbaggage"
6 <?xml version='1.0' encoding='utf-8'?>
7 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
  /">
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <paxid>
12      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13    </paxid>
14  </data>
15 </soapenv:Body/>
16 </soapenv:Envelope>
```

Listing 13.7: SOAP Message Load Baggage

### 13.2.6 Unload Baggage Message

```
1 POST /unloadbaggage HTTP/1.0
2 Host: 192.168.1.43:3540
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 305
5 SOAPAction: "unloadbaggage"
6 <?xml version='1.0' encoding='utf-8'?>
7 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
  /">
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <paxid>
12      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13    </paxid>
14  </data>
15 </soapenv:Body/>
16 </soapenv:Envelope>
```

Listing 13.8: SOAP Message Unload Baggage

### 13.2.7 Load A/C Baggage Message

```
1 POST /loadacbaggage HTTP/1.0
2 Host: 192.168.1.43:3550
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 306
5 SOAPAction: "loadacbaggage"
6 <?xml version='1.0' encoding='utf-8'?>
7 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
  /">
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <paxid>
12      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13    </paxid>
14  </data>
15 </soapenv:Body/>
16 </soapenv:Envelope>
```

Listing 13.9: SOAP Message Load A/C Baggage

### 13.2.8 Unload A/C Baggage Message

```
1 POST /unloadacbaggage HTTP/1.0
2 Host: 192.168.1.43:3550
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 306
5 SOAPAction: "unloadacbaggage"
6 <?xml version='1.0' encoding='utf-8'?>
7 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
  /">
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
11    <paxid>
12      <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13    </paxid>
14  </data>
15 </soapenv:Body/>
16 </soapenv:Envelope>
```

Listing 13.10: SOAP Message Unload A/C Baggage

### 13.2.9 Register Position Application Message

```
1 POST /registerpositionapplication HTTP/1.0
2 Host: 192.168.1.43:3530
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 475
5 SOAPAction: "registerpositionapplication"
6 <?xmlversion='1.0'encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  >
8   <soapenv:Header/>
9   <soapenv:Body/>
10  <data>
```

```
11 <boardingpass>
12 <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
13 <tagcodes>
14 <rfidpassive>500</rfidpassive>
15 <rfidactive>600</rfidactive>
16 <2dbarcode>700</2dbarcode>
17 </tagcodes>
18 </boardingpass>
19 </data>
20 </soapenv:Body/>
21 </soapenv:Envelope>
```

Listing 13.11: SOAP Message Register Position Application

### 13.2.10 Update Airport Location Message

```
1 POST /registerpositionapplication HTTP/1.0
2 Host: 192.168.1.43:3530
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 475
5 SOAPAction: "registerpositionapplication"
6 <?xmlversion='1.0'encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
8 >
9 <soapenv:Header/>
10 <soapenv:Body/>
11 <data>
12 <boardingpass>
13 <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
14 <tagcodes>
15 <rfidpassive>500</rfidpassive>
16 <rfidactive>600</rfidactive>
17 <2dbarcode>700</2dbarcode>
18 </tagcodes>
19 </boardingpass>
20 </data>
21 </soapenv:Body/>
22 </soapenv:Envelope>
```

Listing 13.12: SOAP Message Update Airport Location

### 13.2.11 Inform Passenger Message

```
1 POST /informpax HTTP/1.0
2 Host: 192.168.1.43:3510
3 Content-Type: text/xml; charset=utf-8
4 Content-Length: 387
5 SOAPAction: "informpax"
6 <?xmlversion='1.0'encoding='utf-8'?>
7 <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
8 >
9 <soapenv:Header/>
10 <soapenv:Body/>
11 <data>
12 <area>
13 <areacode>3_19849</areacode>
14 </area>
15 <locationupdate>
```

```

15     <temporarypaxid>PAXID_oid_3_7983</temporarypaxid>
16   </locationupdate>
17 </data>
18 </soapenv:Body/>
19 </soapenv:Envelope>

```

Listing 13.13: SOAP Message Inform Passenger

## 13.3 Created Files

This chapter contains generated files as described in the chapters Creation of the Target and Test Data Generation (see chapter 9 *Creation of the Target*, 8 *Test Data Generation*).

### 13.3.1 Overview Created Files

The following files have been created. The given directory names need to be appended to the `$RTT_TESTCONTEXT` variable.

Table 13.1: Created Files - Intermediate Files

Filename	Type	Size	Directory
airport.imf	intermediate format	17156	\imf\
paxdatainject.imf	intermediate format	2239	\imf\

Table 13.2: Created Files - Channels and Signals

Filename	Type	Size	Directory
channels.cnl	channel file	90	\specs\
localsignal.sig	signal file	674	\sigdef\
sigdef.h	header file	156	\sigdef\
rtt_sig_def.c	c file	540	\sigdef\
ifm_sms_signals.sig	signal file	245	\conf\
ifm_notificationsystem_signals.sig	signal file	194	\conf\
sim_notificationsystem_signals.sig	signal file	251	\conf\
ifm_metaedit_signals.sig	signal file	238	\conf\
ifm_injectorproxy_signals.sig	signal file	168	\conf\
ifm_soap_signals.sig	signal file	158	\conf\
sim_paxmove_signals.sig	signal file	293	\conf\
oracle_sms_signals.sig	signal file	252	\conf\
oracle_req_signals.sig	signal file	104	\conf\
Makefile	makefile	299	\sigdef\

Table 13.3: Created Files - Interface Modules

Filename	Type	Size	Directory
mylibrary.h	header file	2668	\inc\ 
mylibrary.rts	rt-tester file	13664	\specs\ 
ifm_sms.h	header file	1149	\inc\ 
ifm_sms.rts	rt-tester file	12239	\specs\ 
ifm_notificationsystem.h	header file	1524	\inc\ 
ifm_notificationsystem.rts	rt-tester file	20322	\specs\ 
ifm_metaedit.h	header file	1342	\inc\ 
ifm_metaedit.rts	rt-tester file	13897	\specs\ 
ifm_injectorproxy.h	header file	149	\inc\ 
ifm_injectorproxy.rts	rt-tester file	6163	\specs\ 
ifm_soap.h	header file	12051	\inc\ 
ifm_soap.rts	rt-tester file	79656	\specs\ 

Table 13.4: Created Files - Oracles

Filename	Type	Size	Directory
am_oracle_sms.h	header file	470	\inc\ 
am_oracle_sms.rts	rt-tester file	10535	\specs\ 
am_oracle_req.h	header file	1012	\inc\ 
am_oracle_req.rts	rt-tester file	24466	\specs\ 

Table 13.5: Created Files - Simulations

Filename	Type	Size	Directory
sim_amsterdam_acbaggagesystem.rts	rt-tester file	40865	\specs\ 
sim_amsterdam_checkinsystem.rts	rt-tester file	55182	\specs\ 
sim_amsterdam_notificationsystem.rts	rt-tester file	73697	\specs\ 
sim_amsterdam_ongroundbaggagesystem.rts	rt-tester file	44285	\specs\ 
sim_amsterdam_positionapplication.rts	rt-tester file	49213	\specs\ 
sim_amsterdam_profiledatastore.rts	rt-tester file	47144	\specs\ 

Table 13.6: Created Files - Test Stimulus

Filename	Type	Size	Directory
am_paxmove.h	header file	168	\inc\ 
am_paxmove.rts	rt-tester file	11527	\specs\ 

Table 13.7: Created Files - Config Files

Filename	Type	Size	Directory
project.rtp	project file	2200	\
e2echain.conf	config file	7580	\conf\

Table 13.8: Created Files - Shell Scripts

Filename	Type	Size	Directory
executeCleanDirectories	shell script	1087	\bin\
executeTDG	shell script	160	\bin\
executeCompileSimulation	shell script	1003	\bin\

### 13.3.2 Signal Definition Generator Files

In order to use the signals on the test bench, two additional files need to be present in the /sig/ folder. Both are necessary for the RTT.

```

1 Report '_createTBSignalDescription_Global_Sigdef_HEADER'
2 filename $filename_full_local_TB_SIG_RTTESTER_HEADER write
3 '#ifndef _SIGDEF_H_
4 #define _SIGDEF_H_
5 #include "rttsiglib.h"
6 typedef enum rttSignal_e
7 {
8   RTTSIG_NUM_GLOBAL_SIGNALS = 1
9 } rttSignal_t;
10 #endif
11 '
12 close
13 endreport

```

Listing 13.14: Local Script `_createTBSignalDescription_Global_Sigdef_HEADER`

```

1 Report '_createTBSignalDescription_Global_Sigdef_RTS'
2 filename $filename_full_local_TB_SIG_RTTESTER_RTS write
3 '#include "sigdef.h"
4 #include "rttsiglib.h"
5 int rtt_init_testbed_signals ( rttSigDef_t *** sigDef,
6   uint32_t *   numSig )
7 {
8   rttSigDef_t ** sigDetail;
9   sigDetail = ( rttSigDef_t ** ) calloc ( 1,
10   sizeof ( rttSigDef_t * ) * RTTSIG_NUM_GLOBAL_SIGNALS );
11   if ( sigDetail == NULL )
12   {
13     printf ( "Allocation of signal definition array failed\n" );
14     return 1;
15   }
16   * sigDef = sigDetail;
17   * numSig = RTTSIG_NUM_GLOBAL_SIGNALS;

```

```
18 return 0;
19 }
20 '
21 close
22 endreport
```

Listing 13.15: Local Script `_createTBSignalDescription_Global_Sigdef_RTS`

### 13.3.3 Configuration Files

#### 13.3.3.1 Test Bench Project Configuration File

```
1 // RT-Tester Tool Qualification Project
2 PROJECT; "e_cab_scenario"
3
4 // use c++ compiler
5 LANG; c++
6
7 // The baseline identification will be used in all
8 // test procedure and test results documents
9 BASELINE; "BASE.0.0.1"
10
11 // -----
12 // Standard include files to be used in every
13 // unit test. they are looked up under the
14 // include paths specified in the CFLAGS section above.
15 // If system-global headers are referenced, they
16 // are surrounded by <..>.
17 // -----
18
19 // -----
20 // compile flags to be used in all make files for generation
21 // of executable test cases
22 // -----
23
24 // -----
25 // linker flags
26 // -----
27
28 // -----
29 // Time resolution (millisecond/microsecond)
30 // For SWI/unit tests on PCs, only microsecond is useful.
31 // -----
32 TIMERS; microsecond
33
34 // -----
35 // Time stamp format to be used in test execution logs
36 // "absolute" generates format
37 // <year>:<month>:<day>:<hour>:<minute>:<second>.<millisecond>
38 // 2003:12:31:15:22:59.123
39 // "relative" generates format
40 // <millisecond since start of test>
41 // 000000012345
42 // The third column may contain the reference point of the time (optional)
43 // :
44 // "local" uses the local time zone (default)
45 // "gm" uses Coordinated Universal Time (UTC)
```



```
45 //    formerly known as Greenwich Mean Time (GMT)
46 // The reference is listed only at the beginning of the test log.
47 // -----
48 TIMEFORMAT; relative; local
49
50 // -----
51 // Path to local signal definition file
52 // -----
53 SIGNALDEF_LOCAL; ${RTT_TESTCONTEXT}/e2echain/sigdef/localsignal.sig
```

Listing 13.16: Local Script project.rtp

### 13.3.3.2 Test Bench Test Configuration File

```
1 //
2 //-----
3 //
4 // (C) Copyright Tobias Hartmann
5 //
6 //-----
7 //
8 // Product: E-Cab
9 //
10 //-----
11 //
12 // Description:
13 //
14 //    Configuration for standard test framework.
15 //
16 //-----
17 //
18
19 //-----
20 // Test Configuration for test framework
21 //-----
22
23 //-----
24 // Test Integration Level
25 //-----
26 TILEVEL; HSI
27
28 //-----
29 // SUT identification
30 //-----
31 TLA; System_Test
32
33 //-----
34 // Object files
35 //-----
36 //    OBJ ; ""
37
38 //-----
39 // Flag indicating whether the test4 execution should
40 // stop as soon as the FAIL verdict is set.
41 //-----
42 STOPONFAIL ; YES
43
44 //-----
45 // Additional compile flags
```

```
46 // These will be inserted before the flags defined
47 // on project level.
48 //-----
49 CFLAGS ; -I${RTT_TESTCONTEXT}/inc
50 ; -g
51 ; -Wall
52 ; -D_DEBUG
53 ; -D_SIM_DEBUG
54 ; -D_SIM_DEBUG2
55 ; -I/home/hartmann/ecab/e2echain/inc
56 ; -I/home/hartmann/ecab/e2echain/sigdef
57
58 //-----
59 // Additional linker flags
60 // These will be inserted before the flags defined
61 // on project level.
62 // -l<lib> entries for stubs will be automatically
63 // generated.
64 //-----
65 LDPATH ; -lssl
66 ; -lcurl
67 ; -L/home/hartmann/ecab/e2echain/sigdef
68 ; -lglobalsigdeflib
69 LDFLAGS ; -lglobalsigdeflib
70
71 //-----
72 // additional include files
73 //-----
74 INCLUDE ; "rttsiglib.h"
75
76 //-----
77 // Channel definitions
78 //-----
79 CHANNELS ; channels.cn1
80
81 //-----
82 // Additional RTS files
83 //-----
84 ADDITIONAL_RTS ; mylibrary.rts
85
86 //-----
87 // Abstract Machine definition
88 //-----
89
90 // IFM SMS Gateway
91 AM ; ifm_sms
92 ; AMPROCESS ; ifm_sms
93 ; SPEC ; ifm_sms
94 ; SIGNALSET ; ifm_sms_signals.sig
95 ; SEED ; 23
96 ; SCRATCHPAD_SIZE ; 4 k
97 ; ACTIVATE ; YES
98 ; OUTPUTFILE ; YES
99 ; OUTPUTSTDOUT ; YES
100
101 // IFM Notification System
102 AM ; ifm_notificationsystem
103 ; AMPROCESS ; ifm_notificationsystem
104 ; SPEC ; ifm_notificationsystem
105 ; SIGNALSET ; ifm_notificationsystem_signals.sig
106 ; SEED ; 23
```

### 13.3.3.2 TEST BENCH TEST CONFIGURATION FILE

---

```
107 ; SCRATCHPAD SIZE ; 4 k
108 ; ACTIVATE ; YES
109 ; OUTPUTFILE ; YES
110 ; OUTPUTSTDOUT ; YES
111
112 // IFM MetaEdit
113 AM ; ifm_metaedit
114 ; AMPROCESS ; ifm_metaedit
115 ; SPEC ; ifm_metaedit
116 ; SIGNALSET ; ifm_metaedit_signals.sig
117 ; SEED ; 23
118 ; SCRATCHPAD SIZE ; 4 k
119 ; ACTIVATE ; YES
120 ; OUTPUTFILE ; YES
121 ; OUTPUTSTDOUT ; YES
122
123 // IFM SOAP
124 AM ; ifm_soap
125 ; AMPROCESS ; ifm_soap
126 ; SPEC ; ifm_soap
127 ; SIGNALSET ; ifm_soap_signals.sig
128 ; SEED ; 23
129 ; SCRATCHPAD SIZE ; 4 k
130 ; ACTIVATE ; YES
131 ; OUTPUTFILE ; YES
132 ; OUTPUTSTDOUT ; YES
133
134 // IFM INJECTOR PROXY
135 AM ; ifm_injectorproxy
136 ; AMPROCESS ; ifm_injectorproxy
137 ; SPEC ; ifm_injectorproxy
138 ; SIGNALSET ; ifm_injectorproxy_signals.sig
139 ; SEED ; 23
140 ; SCRATCHPAD SIZE ; 4 k
141 ; ACTIVATE ; YES
142 ; OUTPUTFILE ; YES
143 ; OUTPUTSTDOUT ; YES
144
145 // Oracle SMS
146 AM ; am_oracle_sms
147 ; AMPROCESS ; am_oracle_sms
148 ; SPEC ; am_oracle_sms
149 ; SIGNALSET ; oracle_sms_signals.sig
150 ; SEED ; 23
151 ; SCRATCHPAD SIZE ; 4 k
152 ; ACTIVATE ; YES
153 ; OUTPUTFILE ; YES
154 ; OUTPUTSTDOUT ; YES
155
156 // Oracle REQ
157 AM ; am_oracle_req
158 ; AMPROCESS ; am_oracle_req
159 ; SPEC ; am_oracle_req
160 ; SIGNALSET ; oracle_req_signals.sig
161 ; SEED ; 23
162 ; SCRATCHPAD SIZE ; 4 k
163 ; ACTIVATE ; YES
164 ; OUTPUTFILE ; YES
165 ; OUTPUTSTDOUT ; YES
166
167 AM ; sim_amsterdam_acbaggagesystem
```

```

168 ; AMPROCESS ; sim_amsterdam_acbaggagesystem
169 ; SPEC ; sim_amsterdam_acbaggagesystem
170 ; SEED ; 23
171 ; SCRATCHPAD SIZE ; 4 k
172 ; ACTIVATE ; YES
173 ; OUTPUTFILE ; YES
174 ; OUTPUTSTDOUT ; YES
175
176 AM ; sim_amsterdam_checkinsystem
177 ; AMPROCESS ; sim_amsterdam_checkinsystem
178 ; SPEC ; sim_amsterdam_checkinsystem
179 ; SEED ; 23
180 ; SCRATCHPAD SIZE ; 4 k
181 ; ACTIVATE ; YES
182 ; OUTPUTFILE ; YES
183 ; OUTPUTSTDOUT ; YES
184
185 AM ; sim_amsterdam_notificationsystem
186 ; AMPROCESS ; sim_amsterdam_notificationsystem
187 ; SPEC ; sim_amsterdam_notificationsystem
188 ; SEED ; 23
189 ; SCRATCHPAD SIZE ; 4 k
190 ; ACTIVATE ; YES
191 ; OUTPUTFILE ; YES
192 ; OUTPUTSTDOUT ; YES
193
194 AM ; sim_amsterdam_notificationsystem_worker
195 ; AMPROCESS ; sim_amsterdam_notificationsystem_worker
196 ; SPEC ; sim_amsterdam_notificationsystem
197 ; SEED ; 23
198 ; SIGNALSET ; sim_notificationsystem_signals.sig
199 ; SCRATCHPAD SIZE ; 4 k
200 ; ACTIVATE ; YES
201 ; OUTPUTFILE ; YES
202 ; OUTPUTSTDOUT ; YES
203
204 AM ; sim_amsterdam_ongroundbaggagesystem
205 ; AMPROCESS ; sim_amsterdam_ongroundbaggagesystem
206 ; SPEC ; sim_amsterdam_ongroundbaggagesystem
207 ; SEED ; 23
208 ; SCRATCHPAD SIZE ; 4 k
209 ; ACTIVATE ; YES
210 ; OUTPUTFILE ; YES
211 ; OUTPUTSTDOUT ; YES
212
213 AM ; sim_amsterdam_positionapplication
214 ; AMPROCESS ; sim_amsterdam_positionapplication
215 ; SPEC ; sim_amsterdam_positionapplication
216 ; SEED ; 23
217 ; SCRATCHPAD SIZE ; 4 k
218 ; ACTIVATE ; YES
219 ; OUTPUTFILE ; YES
220 ; OUTPUTSTDOUT ; YES
221
222 AM ; sim_amsterdam_profiledatastore
223 ; AMPROCESS ; sim_amsterdam_profiledatastore
224 ; SPEC ; sim_amsterdam_profiledatastore
225 ; SEED ; 23
226 ; SCRATCHPAD SIZE ; 4 k
227 ; ACTIVATE ; YES
228 ; OUTPUTFILE ; YES

```

```
229 ; OUTPUTSTDOUT ; YES
230
231 // TEST AM
232 AM ; am_paxmove
233 ; AMPROCESS ; am_paxmove
234 ; SPEC ; am_paxmove
235 ; SIGNALSET ; sim_paxmove_signals.sig
236 ; SEED ; 23
237 ; SCRATCHPAD SIZE ; 4 k
238 ; ACTIVATE ; YES
239 ; OUTPUTFILE ; YES
240 ; OUTPUTSTDOUT ; YES
```

Listing 13.17: Local Script project.rtp

## 13.3.4 Intermediate Format Files

### 13.3.4.1 PAX Movement Test Data

```
1 PAXID_oid_3_7983 ,66667,1448,926
2 PAXID_oid_3_7983 ,66565,1160,853
3 PAXID_oid_3_7983 ,65614,900,891
4 PAXID_oid_3_7983 ,64645,747,1146
5 PAXID_oid_3_7983 ,64010,598,990
6 PAXID_oid_3_7983 ,62883,864,943
7 PAXID_oid_3_7983 ,61731,772,1123
8 PAXID_oid_3_7983 ,60923,407,1169
9 PAXID_oid_3_7983 ,60106,685,1139
10 PAXID_oid_3_7983 ,59114,697,946
11 PAXID_oid_3_7983 ,58523,587,727
12 PAXID_oid_3_7983 ,57875,673,590
13 PAXID_oid_3_7983 ,56676,354,608
14 PAXID_oid_3_7983 ,55994,598,385
15 PAXID_oid_3_7983 ,55165,583,609
16 PAXID_oid_3_7983 ,54373,580,403
17 PAXID_oid_3_7983 ,53749,628,602
18 PAXID_oid_3_7983 ,53213,600,751
19 PAXID_oid_3_7983 ,52459,578,618
20 PAXID_oid_3_7983 ,51592,263,626
21 PAXID_oid_3_7983 ,50980,541,390
22 PAXID_oid_3_7983 ,49563,783,274
23 PAXID_oid_3_7983 ,48932,556,236
24 PAXID_oid_3_8654 ,71902,1441,926
25 PAXID_oid_3_8654 ,71805,1333,904
26 PAXID_oid_3_8654 ,71199,855,958
27 PAXID_oid_3_8654 ,70562,848,1163
28 PAXID_oid_3_8654 ,69621,601,982
29 PAXID_oid_3_8654 ,69153,647,777
30 PAXID_oid_3_8654 ,68356,639,597
31 PAXID_oid_3_8654 ,67824,726,431
32 PAXID_oid_3_8654 ,67222,569,394
33 PAXID_oid_3_8654 ,66627,323,343
34 PAXID_oid_3_8654 ,66006,554,234
```

Listing 13.18: Passenger Movement Data

### 13.3.4.2 Airport Intermediate Format File

```
1 ecab_machine Amsterdam OID : oid3_12018 {
2   pax "PAXID_oid_3_7983";
3   late "false";
4   departure "66780";
5
6   pax "PAXID_oid_3_8654";
7   late "true";
8   departure "68400";
9
10  startstate StartPoint OID : oid3_16007 {
11   x1 "550"
12   y1 "230.0"
13   x2 "560"
14   y2 "240.0"
15   RESLOW "600000"
16   RESHIGH "660000"
17   LLRID "LLR_StartState"
18   LLRCnt "0"
19   LLRCntMin "0"
20   LLRCntMax "0"
21   LLRCntFlag "F"
22   HLRID "HLR_StartState"
23   HLRCnt "0"
24   HLRCntMin "0"
25   HLRCntMax "0"
26   HLRCntFlag "F"
27 }
28  endstate EndPoint OID : oid3_30390 {
29   x1 "1440"
30   y1 "920"
31   x2 "1450"
32   y2 "930"
33   RESLOW "60000"
34   RESHIGH "120000"
35   LLRID "LLR_EndState"
36   LLRCnt "0"
37   LLRCntMin "0"
38   LLRCntMax "0"
39   LLRCntFlag "F"
40   HLRID "HLR_EndState"
41   HLRCnt "0"
42   HLRCntMin "0"
43   HLRCntMax "0"
44   HLRCntFlag "F"
45 }
46  state Restaurant_oid3_16144 OID : oid3_16144 {
47   x1 "889.872"
48   y1 "231.625"
49   x2 "1156.63"
50   y2 "449.435"
51   RESLOW "1200000"
52   RESHIGH "5400000"
53   LLRID "LLR_4Seasons"
54   LLRCnt "0"
55   LLRCntMin "0"
56   LLRCntMax "0"
57   LLRCntFlag "F"
58   HLRID "HLR_4Seasons"
59   HLRCnt "0"
60   HLRCntMin "0"
61   HLRCntMax "0"
```

### 13.3.4.2 AIRPORT INTERMEDIATE FORMAT FILE

---

```
62   HLRCntFlag "F"
63 }
64 state Aircraft_oid3_16265 OID : oid3_16265 {
65   x1 "1140.0"
66   y1 "845.0"
67   x2 "1345.0"
68   y2 "983.0"
69   RESLOW "60000"
70   RESHIGH "120000"
71   LLRID "LLR_AV"
72   LLRCnt "0"
73   LLRCntMin "0"
74   LLRCntMax "0"
75   LLRCntFlag "F"
76   HLRID "HLR_AC"
77   HLRCnt "0"
78   HLRCntMin "0"
79   HLRCntMax "0"
80   HLRCntFlag "F"
81 }
82 state Car_Rental_Service_oid3_16192 OID : oid3_16192 {
83   x1 "685.39"
84   y1 "243.913"
85   x2 "869.39"
86   y2 "378.913"
87   RESLOW "600000"
88   RESHIGH "1800000"
89   LLRID "LLR_Cars"
90   LLRCnt "0"
91   LLRCntMin "0"
92   LLRCntMax "0"
93   LLRCntFlag "F"
94   HLRID "HLR_Cars"
95   HLRCnt "0"
96   HLRCntMin "0"
97   HLRCntMax "0"
98   HLRCntFlag "F"
99 }
100 state Bar_oid3_16121 OID : oid3_16121 {
101   x1 "250.0"
102   y1 "553.0"
103   x2 "455.0"
104   y2 "676.28"
105   RESLOW "300000"
106   RESHIGH "1200000"
107   LLRID "LLR_Coffiee"
108   LLRCnt "0"
109   LLRCntMin "0"
110   LLRCntMax "0"
111   LLRCntFlag "F"
112   HLRID "HLR_Coffiee"
113   HLRCnt "0"
114   HLRCntMin "0"
115   HLRCntMax "0"
116   HLRCntFlag "F"
117 }
118 state Shop_oid3_16311 OID : oid3_16311 {
119   x1 "270.0"
120   y1 "1125.33"
121   x2 "475.0"
122   y2 "1244.33"
```

```

123 RESLOW "300000"
124 RESHIGH "1200000"
125 LLRID "LLR_Comics"
126 LLRCnt "0"
127 LLRCntMin "0"
128 LLRCntMax "0"
129 LLRCntFlag "F"
130 HLRID "HLR_Comics"
131 HLRCnt "0"
132 HLRCntMin "0"
133 HLRCntMax "0"
134 HLRCntFlag "F"
135 }
136 state Bar_oid3_16334 OID : oid3_16334 {
137 x1 "270.0"
138 y1 "985.0"
139 x2 "475.0"
140 y2 "1097.0"
141 RESLOW "600000"
142 RESHIGH "1500000"
143 LLRID "LLR_FnC"
144 LLRCnt "0"
145 LLRCntMin "0"
146 LLRCntMax "0"
147 LLRCntFlag "F"
148 HLRID "HLR_FnC"
149 HLRCnt "0"
150 HLRCntMin "0"
151 HLRCntMax "0"
152 HLRCntFlag "F"
153 }
154 state Travel_Agency_oid3_16098 OID : oid3_16098 {
155 x1 "250.0"
156 y1 "380.238"
157 x2 "455.0"
158 y2 "549.0"
159 RESLOW "300000"
160 RESHIGH "1800000"
161 LLRID "LLR_Holiday"
162 LLRCnt "0"
163 LLRCntMin "0"
164 LLRCntMax "0"
165 LLRCntFlag "F"
166 HLRID "HLR_Holiday"
167 HLRCnt "0"
168 HLRCntMin "0"
169 HLRCntMax "0"
170 HLRCntFlag "F"
171 }
172 state Shop_oid3_16288 OID : oid3_16288 {
173 x1 "270.0"
174 y1 "845.0"
175 x2 "475.0"
176 y2 "966.0"
177 RESLOW "300000"
178 RESHIGH "900000"
179 LLRID "LLR_IceCream"
180 LLRCnt "0"
181 LLRCntMin "0"
182 LLRCntMax "0"
183 LLRCntFlag "F"

```



```
184 HLRID "HLR_IceCream"
185 HLRCnt "0"
186 HLRCntMin "0"
187 HLRCntMax "0"
188 HLRCntFlag "F"
189 }
190 state Lavatory_oid3_16167 OID : oid3_16167 {
191 x1 "1001.46"
192 y1 "553.696"
193 x2 "1162.46"
194 y2 "675.696"
195 RESLOW "300000"
196 RESHIGH "900000"
197 LLRID ""
198 LLRCnt "0"
199 LLRCntMin "0"
200 LLRCntMax "0"
201 LLRCntFlag "F"
202 HLRID ""
203 HLRCnt "0"
204 HLRCntMin "0"
205 HLRCntMax "0"
206 HLRCntFlag "F"
207 }
208 state Lavatory_oid3_16357 OID : oid3_16357 {
209 x1 "912.0"
210 y1 "1115.0"
211 x2 "1075.0"
212 y2 "1253.0"
213 RESLOW "300000"
214 RESHIGH "600000"
215 LLRID "LLR_Lav"
216 LLRCnt "0"
217 LLRCntMin "0"
218 LLRCntMax "0"
219 LLRCntFlag "F"
220 HLRID "HLR_Lav"
221 HLRCnt "0"
222 HLRCntMin "0"
223 HLRCntMax "0"
224 HLRCntFlag "F"
225 }
226 state Shop_oid3_16215 OID : oid3_16215 {
227 x1 "685.293"
228 y1 "383.696"
229 x2 "869.293"
230 y2 "505.696"
231 RESLOW "300000"
232 RESHIGH "600000"
233 LLRID "LLR_Newspaper"
234 LLRCnt "0"
235 LLRCntMin "0"
236 LLRCntMax "0"
237 LLRCntFlag "F"
238 HLRID "HLR_Newspaper"
239 HLRCnt "0"
240 HLRCntMin "0"
241 HLRCntMax "0"
242 HLRCntFlag "F"
243 }
244 state Quick_Check_In_Terminal_oid3_16075 OID : oid3_16075 {
```

```
245  x1 "250.0"
246  y1 "245.0"
247  x2 "455.0"
248  y2 "383.0"
249  RESLOW "300000"
250  RESHIGH "600000"
251  LLRID "LLR_QCI"
252  LLRCnt "0"
253  LLRCntMin "0"
254  LLRCntMax "0"
255  LLRCntFlag "F"
256  HLRID "HLR_QCI"
257  HLRCnt "0"
258  HLRCntMin "0"
259  HLRCntMax "0"
260  HLRCntFlag "F"
261  }
262  state Waiting_Area_oid3_16380 OID : oid3_16380 {
263  x1 "682.439"
264  y1 "1120.89"
265  x2 "882.439"
266  y2 "1249.89"
267  RESLOW "600000"
268  RESHIGH "1200000"
269  LLRID "LLR_Waiting"
270  LLRCnt "0"
271  LLRCntMin "0"
272  LLRCntMax "0"
273  LLRCntFlag "F"
274  HLRID "HLR_Waiting"
275  HLRCnt "0"
276  HLRCntMin "0"
277  HLRCntMax "0"
278  HLRCntFlag "F"
279  }
280  state junction_oid3_16414 OID : oid3_16414 {
281  x1 "576.0"
282  y1 "909.0"
283  x2 "704.0"
284  y2 "991.0"
285  RESLOW "600000"
286  RESHIGH "1200000"
287  LLRID ""
288  LLRCnt "0"
289  LLRCntMin "0"
290  LLRCntMax "0"
291  LLRCntFlag "F"
292  HLRID ""
293  HLRCnt "0"
294  HLRCntMin "0"
295  HLRCntMax "0"
296  HLRCntFlag "F"
297  }
298  state junction_oid3_16405 OID : oid3_16405 {
299  x1 "822.0"
300  y1 "881.0"
301  x2 "938.0"
302  y2 "959.0"
303  RESLOW "300000"
304  RESHIGH "1200000"
305  LLRID "LLR_Gate200"
```

13.3.4.2 AIRPORT INTERMEDIATE FORMAT FILE

---

```
306 LLRCnt "0"
307 LLRCntMin "1"
308 LLRCntMax "0"
309 LLRCntFlag "T"
310 HLRID "HLR_Gate200"
311 HLRCnt "0"
312 HLRCntMin "1"
313 HLRCntMax "0"
314 HLRCntFlag "T"
315 }
316 state junction_oid3_16656 OID : oid3_16656 {
317 x1 "572.0"
318 y1 "714.0"
319 x2 "702.0"
320 y2 "788.0"
321 RESLOW "300000"
322 RESHIGH "600000"
323 LLRID "LLR_MiddleFloor200"
324 LLRCnt "0"
325 LLRCntMin "1"
326 LLRCntMax "0"
327 LLRCntFlag "T"
328 HLRID "HLR_MiddleFloor200"
329 HLRCnt "0"
330 HLRCntMin "1"
331 HLRCntMax "0"
332 HLRCntFlag "T"
333 }
334 state junction_oid3_16238 OID : oid3_16238 {
335 x1 "530.0"
336 y1 "365.5"
337 x2 "610.0"
338 y2 "414.5"
339 RESLOW "600000"
340 RESHIGH "900000"
341 LLRID ""
342 LLRCnt "0"
343 LLRCntMin "0"
344 LLRCntMax "0"
345 LLRCntFlag "F"
346 HLRID ""
347 HLRCnt "0"
348 HLRCntMin "0"
349 HLRCntMax "0"
350 HLRCntFlag "F"
351 }
352 state junction_oid3_16247 OID : oid3_16247 {
353 x1 "780.5"
354 y1 "557.5"
355 x2 "879.5"
356 y2 "602.5"
357 RESLOW "300000"
358 RESHIGH "600000"
359 LLRID ""
360 LLRCnt "0"
361 LLRCntMin "0"
362 LLRCntMax "0"
363 LLRCntFlag "F"
364 HLRID ""
365 HLRCnt "0"
366 HLRCntMin "0"
```

```
367   HLRCntMax   "0"
368   HLRCntFlag  "F"
369 }
370 state junction_oid3_16256 OID : oid3_16256 {
371   x1 "560.0"
372   y1 "565.5"
373   x2 "680.0"
374   y2 "634.5"
375   RESLOW      "600000"
376   RESHIGH     "900000"
377   LLRID       "LLR_UpperFloor100"
378   LLRCnt      "0"
379   LLRCntMin   "1"
380   LLRCntMax   "0"
381   LLRCntFlag  "T"
382   HLRID       "HLR_UpperFloor100"
383   HLRCnt      "0"
384   HLRCntMin   "1"
385   HLRCntMax   "0"
386   HLRCntFlag  "T"
387 }
388 transition OID : oid3_16425 {
389   robustness "false";
390   condition  "t >= 60000"
391   action     ""
392   tolerance  "120000"
393   probability "0.5"
394   srcstate   "StartPoint";
395   trgstate   "junction_oid3_16238";
396 }
397 transition OID : oid3_16436 {
398   robustness "false";
399   condition  "t >= 60000"
400   action     ""
401   tolerance  "120000"
402   probability "0.2"
403   srcstate   "StartPoint";
404   trgstate   "Quick_Check_In_Terminal_oid3_16075";
405 }
406 transition OID : oid3_16447 {
407   robustness "false";
408   condition  "t >= 60000"
409   action     ""
410   tolerance  "120000"
411   probability "0.3"
412   srcstate   "StartPoint";
413   trgstate   "Car_Rental_Service_oid3_16192";
414 }
415 transition OID : oid3_16570 {
416   robustness "false";
417   condition  "t >= 1200000"
418   action     ""
419   tolerance  "5400000"
420   probability "0.4"
421   srcstate   "Restaurant_oid3_16144";
422   trgstate   "Lavatory_oid3_16167";
423 }
424 transition OID : oid3_16581 {
425   robustness "false";
426   condition  "t >= 1200000"
427   action     ""
```

```
428 tolerance "5400000"
429 probability "0.6"
430 srcstate "Restaurant_oid3_16144";
431 trgstate "junction_oid3_16247";
432 }
433 transition OID : oid3_30397 {
434 robustness "false";
435 condition "t >= 60000"
436 action ""
437 tolerance "120000"
438 probability "1"
439 srcstate "Aircraft_oid3_16265";
440 trgstate "EndPoint";
441 }
442 transition OID : oid3_16469 {
443 robustness "false";
444 condition "t >= 600000"
445 action ""
446 tolerance "1800000"
447 probability "1"
448 srcstate "Car_Rental_Service_oid3_16192";
449 trgstate "junction_oid3_16238";
450 }
451 transition OID : oid3_16614 {
452 robustness "false";
453 condition "t >= 300000"
454 action ""
455 tolerance "1200000"
456 probability "1"
457 srcstate "Bar_oid3_16121";
458 trgstate "junction_oid3_16256";
459 }
460 transition OID : oid3_16832 {
461 robustness "false";
462 condition "t >= 300000"
463 action ""
464 tolerance "1200000"
465 probability "0.7"
466 srcstate "Shop_oid3_16311";
467 trgstate "Waiting_Area_oid3_16380";
468 }
469 transition OID : oid3_16843 {
470 robustness "false";
471 condition "t >= 300000"
472 action ""
473 tolerance "1200000"
474 probability "0.3"
475 srcstate "Shop_oid3_16311";
476 trgstate "junction_oid3_16414";
477 }
478 transition OID : oid3_16821 {
479 robustness "false";
480 condition "t >= 600000"
481 action ""
482 tolerance "1500000"
483 probability "1"
484 srcstate "Bar_oid3_16334";
485 trgstate "junction_oid3_16414";
486 }
487 transition OID : oid3_16603 {
488 robustness "false";
```

```
489   condition "t >= 300000"
490   action ""
491   tolerance "1800000"
492   probability "1"
493   srcstate "Travel_Agency_oid3_16098";
494   trgstate "junction_oid3_16256";
495 }
496 transition OID : oid3_16810 {
497   robustness "false";
498   condition "t >= 300000"
499   action ""
500   tolerance "900000"
501   probability "1"
502   srcstate "Shop_oid3_16288";
503   trgstate "junction_oid3_16414";
504 }
505 transition OID : oid3_16854 {
506   robustness "false";
507   condition "t >= 300000"
508   action ""
509   tolerance "900000"
510   probability "1"
511   srcstate "Lavatory_oid3_16167";
512   trgstate "junction_oid3_16247";
513 }
514 transition OID : oid3_16865 {
515   robustness "false";
516   condition "t >= 300000"
517   action ""
518   tolerance "600000"
519   probability "1"
520   srcstate "Lavatory_oid3_16357";
521   trgstate "Waiting_Area_oid3_16380";
522 }
523 transition OID : oid3_16526 {
524   robustness "false";
525   condition "t >= 300000"
526   action ""
527   tolerance "600000"
528   probability "0.5"
529   srcstate "Shop_oid3_16215";
530   trgstate "junction_oid3_16247";
531 }
532 transition OID : oid3_16537 {
533   robustness "false";
534   condition "t >= 300000"
535   action ""
536   tolerance "600000"
537   probability "0.5"
538   srcstate "Shop_oid3_16215";
539   trgstate "junction_oid3_16256";
540 }
541 transition OID : oid3_16458 {
542   robustness "false";
543   condition "t >= 300000"
544   action ""
545   tolerance "600000"
546   probability "1"
547   srcstate "Quick_Check_In_Terminal_oid3_16075";
548   trgstate "junction_oid3_16238";
549 }
```

### 13.3.4.2 AIRPORT INTERMEDIATE FORMAT FILE

---

```
550 transition OID : oid3_16755 {
551   robustness "false";
552   condition "t >= 600000"
553   action ""
554   tolerance "1200000"
555   probability "0.3"
556   srcstate "Waiting_Area_oid3_16380";
557   trgstate "Lavatory_oid3_16357";
558 }
559 transition OID : oid3_16766 {
560   robustness "false";
561   condition "t >= 600000"
562   action ""
563   tolerance "1200000"
564   probability "0.3"
565   srcstate "Waiting_Area_oid3_16380";
566   trgstate "Shop_oid3_16311";
567 }
568 transition OID : oid3_16777 {
569   robustness "false";
570   condition "t >= 600000"
571   action ""
572   tolerance "1200000"
573   probability "0.4"
574   srcstate "Waiting_Area_oid3_16380";
575   trgstate "junction_oid3_16405";
576 }
577 transition OID : oid3_16700 {
578   robustness "false";
579   condition "t >= 600000"
580   action ""
581   tolerance "1200000"
582   probability "0.2"
583   srcstate "junction_oid3_16414";
584   trgstate "Shop_oid3_16288";
585 }
586 transition OID : oid3_16711 {
587   robustness "false";
588   condition "t >= 600000"
589   action ""
590   tolerance "1200000"
591   probability "0.2"
592   srcstate "junction_oid3_16414";
593   trgstate "Bar_oid3_16334";
594 }
595 transition OID : oid3_16722 {
596   robustness "false";
597   condition "t >= 600000"
598   action ""
599   tolerance "1200000"
600   probability "0.2"
601   srcstate "junction_oid3_16414";
602   trgstate "Shop_oid3_16311";
603 }
604 transition OID : oid3_16733 {
605   robustness "false";
606   condition "t >= 600000"
607   action ""
608   tolerance "1200000"
609   probability "0.4"
610   srcstate "junction_oid3_16414";
```

```
611   trgstate "Waiting_Area_oid3_16380";
612 }
613 transition OID : oid3_16788 {
614   robustness "false";
615   condition "t >= 300000"
616   action ""
617   tolerance "1200000"
618   probability "0.8"
619   srcstate "junction_oid3_16405";
620   trgstate "Aircraft_oid3_16265";
621 }
622 transition OID : oid3_16799 {
623   robustness "false";
624   condition "t >= 300000"
625   action ""
626   tolerance "1200000"
627   probability "0.2"
628   srcstate "junction_oid3_16405";
629   trgstate "junction_oid3_16414";
630 }
631 transition OID : oid3_16678 {
632   robustness "false";
633   condition "t >= 300000"
634   action ""
635   tolerance "600000"
636   probability "0.3"
637   srcstate "junction_oid3_16656";
638   trgstate "junction_oid3_16256";
639 }
640 transition OID : oid3_16689 {
641   robustness "false";
642   condition "t >= 300000"
643   action ""
644   tolerance "600000"
645   probability "0.7"
646   srcstate "junction_oid3_16656";
647   trgstate "junction_oid3_16414";
648 }
649 transition OID : oid3_16480 {
650   robustness "false";
651   condition "t >= 600000"
652   action ""
653   tolerance "900000"
654   probability "0.2"
655   srcstate "junction_oid3_16238";
656   trgstate "Travel_Agency_oid3_16098";
657 }
658 transition OID : oid3_16491 {
659   robustness "false";
660   condition "t >= 600000"
661   action ""
662   tolerance "900000"
663   probability "0.2"
664   srcstate "junction_oid3_16238";
665   trgstate "Bar_oid3_16121";
666 }
667 transition OID : oid3_16502 {
668   robustness "false";
669   condition "t >= 600000"
670   action ""
671   tolerance "900000"
```



```
672 probability "0.4"
673 srcstate "junction_oid3_16238";
674 trgstate "Shop_oid3_16215";
675 }
676 transition OID : oid3_16513 {
677 robustness "false";
678 condition "t >= 600000"
679 action ""
680 tolerance "900000"
681 probability "0.2"
682 srcstate "junction_oid3_16238";
683 trgstate "junction_oid3_16256";
684 }
685 transition OID : oid3_16548 {
686 robustness "false";
687 condition "t >= 300000"
688 action ""
689 tolerance "600000"
690 probability "0.3"
691 srcstate "junction_oid3_16247";
692 trgstate "Lavatory_oid3_16167";
693 }
694 transition OID : oid3_16559 {
695 robustness "false";
696 condition "t >= 300000"
697 action ""
698 tolerance "600000"
699 probability "0.5"
700 srcstate "junction_oid3_16247";
701 trgstate "Restaurant_oid3_16144";
702 }
703 transition OID : oid3_16592 {
704 robustness "false";
705 condition "t >= 300000"
706 action ""
707 tolerance "600000"
708 probability "0.2"
709 srcstate "junction_oid3_16247";
710 trgstate "junction_oid3_16256";
711 }
712 transition OID : oid3_16625 {
713 robustness "false";
714 condition "t >= 600000"
715 action ""
716 tolerance "900000"
717 probability "0.2"
718 srcstate "junction_oid3_16256";
719 trgstate "junction_oid3_16238";
720 }
721 transition OID : oid3_16636 {
722 robustness "false";
723 condition "t >= 600000"
724 action ""
725 tolerance "900000"
726 probability "0.4"
727 srcstate "junction_oid3_16256";
728 trgstate "junction_oid3_16247";
729 }
730 transition OID : oid3_16667 {
731 robustness "false";
732 condition "t >= 600000"
```

```
733  action ""
734  tolerance "900000"
735  probability "0.4"
736  srcstate "junction_oid3_16256";
737  trgstate "junction_oid3_16656";
738  }
739 }
```

Listing 13.19: Airport Intermediate Format

### 13.3.4.3 Position Update Tool Intermediate Format File

```
1  paxid;PAXID_oid_3_21948
2  gender;Male
3  title;Mr
4  name1;James
5  name2;
6  familyname;Bond
7  birthdate;7.7.1977
8  nationality;GB
9  passportnumber;007
10 handicapstatus;Normal
11 streetname;Bond Street
12 streetnumber;7
13 postofficebox;7007
14 city;London
15 citycode;
16 country;GB
17 countrycode;
18 mobile;007
19 phone;007
20 fax;007
21 email;007@secretsservice.co.uk
22 website;www.jamesbond.co.uk
23 frequentflyernumber;007
24 frequentflyerstatus;Gold
25 profession;Cocktail Taster
26 baggagestatusnotificationlevel;Information
27 guidancestatus;Enabled
28 notificationlanguage;UK
29 notificationlevel;High
30 notificationmedia;Text
31 trackingstatus;Enabled
32 processstatus;Checked-In
33 value;
34 sequencenumber;007
35 bookingreference;007
36 ticketnumber;007
37 temporarypaxid;PAXID_oid_3_21948
38 rfidpassive;
39 rfidactive;
40 2dbarcode;007
41 seatnumber;007
42 flightidentity;AMS1900
43 portsofcall;AMS
44 scheduleddatetime;68400
45 flightdirection;Departure
46 sourceairport;Amsterdam
47 servercheckinname;192.168.1.43
```

### 13.3.4.3 POSITION UPDATE TOOL INTERMEDIATE FORMAT FILE

---

```
48 servercheckinport ;3520
49 serverpositionapplicationname ;192.168.1.43
50 serverpositionapplicationport ;3530
51 serverinjectorproxyname ;192.168.1.43
52 serverinjectorproxyport ;7896
53 startx ;550
54 starty ;230.0
55 paxid ;PAXID_oid_3_10914
56 gender ;Male
57 title ;Mr
58 name1 ;Max
59 name2 ;
60 familyname ;Mustermann
61 birthdate ;1.1.1970
62 nationality ;German
63 passportnumber ;111970
64 handicapstatus ;Normal
65 streetname ;Musterstra?e
66 streetnumber ;1
67 postofficebox ;
68 city ;Musterstadt
69 citycode ;
70 country ;Germany
71 countrycode ;
72 mobile ;+4917154687398
73 phone ;
74 fax ;
75 email ;
76 website ;
77 frequentflyernumber ;
78 frequentflyerstatus ;Non
79 profession ;
80 baggagestatusnotificationlevel ;Information
81 guidancestatus ;Disabled
82 notificationlanguage ;UK
83 notificationlevel ;High
84 notificationmedia ;Text
85 trackingstatus ;Disabled
86 processstatus ;Checked-In
87 value ;
88 sequencenumber ;
89 bookingreference ;
90 ticketnumber ;
91 temporarypaxid ;PAXID_oid_3_10914
92 rfidpassive ;
93 rfidactive ;
94 2dbarcode ;
95 seatnumber ;
96 flightidentity ;AMS1900
97 portsofcall ;AMS
98 scheduleddatetime ;68400
99 flightdirection ;Departure
100 sourceairport ;Amsterdam
101 servercheckinname ;192.168.1.43
102 servercheckinport ;3520
103 serverpositionapplicationname ;192.168.1.43
104 serverpositionapplicationport ;3530
105 serverinjectorproxyname ;192.168.1.43
106 serverinjectorproxyport ;7896
107 startx ;550
```

```
108 starty;230.0
```

Listing 13.20: Position Update Tool Intermediate Format File

---

# Bibliography

- [1] AbsInt aiT. [www.absint.com](http://www.absint.com).
- [2] Steve Anonsen. Experiences in Modeling for a Domain Specific Language. In *Software Language Engineering*, pages 187–197. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [3] Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test automation for hybrid systems. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, page 14, New York, New York, USA, November 2006. ACM Request Permissions.
- [4] Klaus Birken, Daniel Hünig, Thomas Rustemeyer, and Ralph Wittmann. Resource Analysis of Automotive/Infotainment Systems Based on Domain-Specific Models – A Real-World Example. In *Software Language Engineering*, pages 424–433. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [5] Barry W. Boehm. Software Engineering Economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.
- [6] Richard Brath. Paper landscapes: A visualization design methodology. *Visualization and Data Analysis*, 2003.
- [7] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. Concurrent modeling in early phases of the software development life cycle. In *CRIWG'10: Proceedings of the 16th international conference on Collaboration and technology*. Springer-Verlag, September 2010.
- [8] William H. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. AntiPatterns: refactoring software, architectures, and projects in crisis. *AntiPatterns: refactoring software, architectures, and projects in crisis*, January 1998.
- [9] Tony Cant, Ben Long, Jim McCarthy, Brendan Mahony, and Kylie Williams. The HiVe Writer. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 217:221–234, July 2008.
- [10] Stuart K. Card and Jock Mackinlay. The structure of the information visualization design space. In *VIZ '97: Visualization Conference, Information Visualization Symposium and Parallel Rendering Symposium*, pages 92–99. IEEE Comput. Soc, 1997.

- [11] Gustavo Carvalho, Florian Lapschies, Uwe Schulz, and Jan Peleska. Model Based Testing from Controlled Natural Language Requirements. In *Second International Workshop on Formal Techniques for Safety-Critical Systems*, pages 195–210, October 2013.
- [12] Carla-Fabiana Chiasserini. RFC 3330 - Special-Use IPv4 Addresses. Technical report, 2002.
- [13] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. RFC 6335 - Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. Technical report, Internet Engineering Task Force, August 2011.
- [14] Gerardo de Geest, Sander Vermolen, Arie van Deursen, and Eelco Visser. Generating Version Convertors for Domain-Specific Languages. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 197–201. IEEE Computer Society, October 2008.
- [15] Marc Delpont and Reinhard Botha. Visualization Techniques for the Display of Graphs. (12201400):196, 2008.
- [16] Tom DeMarco. *Structured Analysis and System Specification*. Prentice Hall, 1979.
- [17] Edsger Wybe Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1, pages 269–277.
- [18] Christof Efkemann and Jan Peleska. Model-Based Testing for the Second Generation of Integrated Modular Avionics. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 55–62. IEEE.
- [19] Neil A. Ernst, Yijun Yu, and John Mylopoulos. Visualizing non-functional requirements. In *REV '06: Proceedings of the 1st international workshop on Requirements Engineering Visualization*, pages 2–2. IEEE Computer Society, September 2006.
- [20] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring*. Improving the Design of Existing Code. Addison-Wesley, March 2012.
- [21] Greg Freeman, Don Batory, Greg Lavender, and Jacob Neal Sarvela. Lifting transformational models of product lines: a case study. *Software & Systems Modeling*, 9(3):359–373, October 2009.

- [22] Mathias Fritzsche, Jendrik Johannes, Uwe Aßmann, Simon Mitschke, Wasif Gilani, Ivor Spence, John Brown, and Peter Kilpatrick. Systematic Usage of Embedded Modelling Languages in Automated Model Transformation Chains. *Software Language Engineering*, 5452(Chapter 9):134–150, March 2009.
- [23] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3-4):415–454, December 2009.
- [24] Kevin Hammond and Greg Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Software Language Engineering*, pages 37–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
- [25] Kevin Hammond and Greg Michaelson. The Design of Hume: A High-Level Language for the Real-Time Embedded Systems Domain. In *Software Language Engineering*, pages 127–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [26] Tobias Hartmann. End-To-End Chain Testing. In *Talk at RWTH Aachen*, pages 1–67, November 2009.
- [27] Tobias Hartmann and Christof Efekemann. Specification of Conditions for Error Diagnostics. In *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008)*, pages 97–112, Sydney, 2008. Electronic Notes in Theoretical Computer Science.
- [28] Tobias Hartmann, Claas Heidmann, Volker Hasbach, Federico Baroni, and Jan Peleska. Test Infrastructure Definition (D720-1). Technical report, June 2007.
- [29] Tobias Hartmann, Jan Peleska, Claas Heidmann, Federico Baroni, and Volker Hasbach. Test Platform Specification (D730-1). Technical report, September 2007.
- [30] Tobias Hartmann, Stefan Richter, and Matthias Benesch. Test Tools (D740-3). Technical report, March 2009.
- [31] Tobias Hartmann, Alban Roger, and Claas Heidmann. Overall Test Scenarios (D730-3). *Internal Document E-Cab*, pages 1–57, December 2007.
- [32] Anne E. Haxthausen and Jan Peleska. Formal development and verification of a distributed railway control system. In *Software Language Engineering*, pages 1546–1563. Springer Berlin Heidelberg, Berlin, Heidelberg, September 1999.
- [33] Steve Hitchman. The Details of Conceptual Modelling Notations are Important—A Comparison of Relationship Normative Language. In *Communications of the Association for Information Systems*. Communications of the Association for Information . . . , 2002.

- [34] Kyungsoo Im, Tacksoo Im, and John D McGregor. Automating test case definition using a domain specific language. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference on XX*, pages 180–185, New York, New York, USA, March 2008. ACM Request Permissions.
- [35] Ethan K. Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjørner, and Wolfram Schulte. Specifying and Composing Non-functional Requirements in Model-Based Development. In *SC '09: Proceedings of the 8th International Conference on Software Composition*, pages 72–89, Berlin, Heidelberg, June 2009. Springer-Verlag.
- [36] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. Enabling Full Code Generation. John Wiley & Sons, Hoboken, NJ, USA, April 2008.
- [37] Jessie B. Kennedy, Kenneth J. Mitchell, and Peter J. Barclay. A framework for information visualisation. *SIGMOD Record*, 25(4):30–34, December 1996.
- [38] Anneke Kleppe. Towards General Purpose, High Level, Software Languages. In *Lecture Notes in Computer Science*, pages 220–238. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.
- [39] Tomaz Kosar, Pablo E Martinez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, April 2008.
- [40] Stephen M. Kosslyn. Graphics and human information processing: A review of five books. *Journal of the American Statistical Association*, 1985.
- [41] Thomas Kuhn and Reinhard Gotzhein. Model-Driven Platform-Specific Testing through Configurable Simulations. In *Model Driven Architecture – Foundations and Applications*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [42] Vinay Kulkarni, Sreedhar Reddy, and Asha Rajbhoj. Scaling up model driven engineering-experience and lessons learnt. In *MODELS'10: Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*. Springer-Verlag, October 2010.
- [43] Philipp W. Kutter, Daniel Schweizer, and Lothar Thiele. *Integrating Domain Specific Language Design in the Software Life Cycle*, volume 1641 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, lecture notes in computer science edition, 1999.
- [44] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 1987.



- [45] Chuan-kai Lin and Andrew P. Black. DirectFlow: A Domain-Specific Language for Information-Flow Systems. In *ECOOOP 2007 – Object-Oriented Programming*, pages 299–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [46] Jason Mansell, Aitor Bediaga, Régis Vogel, and Keith Mantell. A process framework for the successful adoption of model driven development. In *SC '09: Proceedings of the 8th International Conference on Software Composition*, pages 90–100, Berlin, Heidelberg, July 2006. Springer-Verlag.
- [47] Richard E. Mayer and Roxana Moreno. Nine ways to reduce cognitive load in multimedia learning. *Educational psychologist*, 2003.
- [48] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *Computing Surveys (CSUR)*, 37(4):316–344, December 2005.
- [49] MetaCase. [www.metacase.com](http://www.metacase.com).
- [50] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):303–319, August 2006.
- [51] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In *Software Language Engineering*, pages 276–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [52] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3):345–361, May 2009.
- [53] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *Model-Driven Platform-Specific Testing through Configurable Simulations*, pages 432–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [54] Daniel L. Moody. The “Physics” of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35:756–756, December 2009.
- [55] Daniel L. Moody, Patrick Heymans, and Raimundas Raimundas Matulevičius. Visual syntax does matter: improving the cognitive effectiveness of the i\* visual notation. *Requirements Engineering*, 15(2):141–175, June 2010.

- [56] Andrew Moss and Henk Muller. Efficient code generation for a domain specific language. *Generative Programming and Component . . .*, 2005.
- [57] Chris North. Visualization Design Principles. pages 1–26, April 2001.
- [58] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *MODELS'07: Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems*, pages 635–649, Berlin, Heidelberg, September 2007. Springer-Verlag.
- [59] Pushover. [www.pushover.net](http://www.pushover.net).
- [60] Chris Raistrick and Tony Bloomfield. Model Driven Architecture – An Industry Perspective. In *Software Language Engineering*, pages 330–350. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [61] David A. Scanlan. Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Software*, 6(5):28–36, September 1989.
- [62] Bran Selic. The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective. In *Software Language Engineering*, pages 290–321. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [63] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE Comput. Soc. Press, 1996.
- [64] Herbert A. Simon. Information-processing theory of human problem solving. *Handbook of Learning and Cognitive Processes*, January 1978.
- [65] Herbert A. Simon and Allen Newell. Human problem solving: The state of the theory in 1970. *American Psychologist*, 26(2):145–159, 1971.
- [66] Helaine Sousa, Denivaldo Lopes, Zair Abdelouahab, Slimane Hammoudi, and Daniela Barreiro Claro. Building Test Cases through Model Driven Engineering. In *Innovations in Computing Sciences and Software Engineering*, pages 395–401. Springer Netherlands, Dordrecht, May 2010.
- [67] Miroslaw Staron, Ludwik Kuzniarz, and Ludwik Wallin. Case study on a process of industrial MDA realization: determinants of effectiveness. *Nordic Journal of Computing*, 11(3), September 2004.
- [68] Dominik Stein and Stefan Hanenberg. Assessing the Power of a Visual Modeling Notation – Preliminary Contemplations on Designing a Test –. In *Software Language Engineering*, pages 78–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

- [69] The Middleware Company. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach. pages 1–18, July 2003.
- [70] Scott A. Thibault, Renaud Marlet, and Charles Consel. Domain-specific languages: from design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, 1999.
- [71] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *Software Language Engineering*, pages 198–209. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [72] Edward R. Tufte. *Envisioning Information*, 1992.
- [73] Edward R. Tufte. *The Visual Display of Quantitative Information*, January 2001.
- [74] Marcel van Amstel, Mark van den Brand, and Luc Engelen. An exercise in iterative domain-specific language design? In *the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 48–57, New York, New York, USA, 2010. ACM Press.
- [75] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Little languages: little maintenance*, March 1997.
- [76] Arie van Deursen, Paul Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
- [77] Verified Systems. *RT-Tester 6.0 User Manual*, November 2011.
- [78] Jonas Völcker. *A Quantitative Analysis of Statechart Aesthetics and Statechart Development Methods*, 2008.
- [79] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin van den Berg, Kim Fleer, David Nelson, Michael Wells, and Brian Mastenbrook. Experiences in Deploying Model-Driven Engineering. In *Software Language Engineering*, pages 35–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [80] Frank Weil, Brian Mastenbrook, David Nelson, Paul Dietz, and Aswin van den Berg. Automated Semantic Analysis of Design Models. In *Lecture Notes in Computer Science*, pages 166–180. Springer Berlin / Heidelberg SN -, Berlin, Heidelberg, 2007.
- [81] Christoph Wienands and Michael Golm. Anatomy of a Visual Domain-Specific Language Project in an Industrial Context. In *Software Language Engineering*, pages 453–467. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

- [82] Alan Cameron Wills and Steven Kelly. Agile development with Domain Specific Languages scaling up agile: is Domain-Specific Modeling the key? In *XP'05: Proceedings of the 6th international conference on Extreme Programming and Agile Processes in Software Engineering*, pages 311–314. Springer-Verlag, June 2005.
- [83] Hui Wu, Jeff Gray, and Marjan Mernik. Unit Testing for Domain-Specific Languages. In *DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, pages 125–147, Berlin, Heidelberg, July 2009. Springer-Verlag.
- [84] Hao Xu. EriLex: An Embedded Domain Specific Language Generator. In *Software Language Engineering*, pages 192–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

---

# Glossary

## **3G**

3G is the abbreviation for third generation mobile cellular network.

## **AM**

Abstract Machine is a Real-Time Tester Entity.

## **AFDX**

Avionics Full-Duplex Switched Ethernet is a standardized data network used in aircrafts.

## **Airbus Directives ABD100/ABD200**

Airbus Directives ABD100/ABD200 describe Airbus internal processes and activities.

## **ARINC**

Abbreviation for Aeronautical Radio, Incorporated. The company develops and maintains standards including the aircraft domain.

## **ARINC 653**

Software specification for partition segregation and time partitioning in real-time operating systems for safety-critical avionics.

## **AWK**

Abbreviation from the authors' last names: Alfred Aho, Peter Weinberger, and Brian Kernighan

## **Black-Box Test**

Black-Box testing is a testing method where the complete system is taken into account.

## **BNF**

Backus–Naur Form is the main notation technique for describing the syntax of computer languages.

## **Bluetooth**

Short range wireless technology

**CAN**

Controller Area Network is a short range local network of controllers.

**CCL**

Communication Control Link is a high-speed link between cluster nodes to connect Real-Time Tester Entities.

**CFG**

Control Flow Graph is a representation of all paths through a program.

**CORBA**

Common Object Request Broker Architecture is an exchange standard for systems which are deployed on different platforms.

**COTS**

Commercial Off-The-Shelf are standardized products where no customization for a certain scope of application is needed nor wanted.

**Compact PCI**

Compact PCI is a local computer bus (Peripheral Component Interconnect).

**CPU**

Central Processing Unit of a computer consisting of a processor and control unit

**CRC**

Cyclic Redundancy Check detects errors using checksums.

**CSS**

Cascading Style Sheets is a style sheet language for HTML.

**DAL A**

Design Assurance Level A is the highest of five levels (DAL A-E). A failure of such a DAL A system might have catastrophic effects on the aircraft.

**DDD**

Domain-Driven Development is development based on the knowledge of a specific domain.

**DDL**

Data Distribution Layer is a logical layer for exchanging data.

**DO178B**

DO178B is a document which provides guidance for an airworthiness certification process (Software Considerations in Airborne Systems and Equipment Certification).

**DOS**

Disk Operating System is a single user operating system.

**DOT**

DOT is a plain text description language for graphs.

**DSL**

Domain Specific Language is a language which was designed for a particular domain.

**DSM**

Domain-Specific Modeling describes a software engineering methodology based on the creation of models according to a specific domain.

**E-Cab**

E-Enabled Cabin is the project name for providing airports and airlines with an environment that enables change in service concepts.

**End-To-End Chain**

End-To-End Chain describes a flow of information from one system to another one while many other systems are involved for relaying this data.

**GOPRR**

Abbreviation for Graph, Object, Property, Port, Relationship, Role Description. These are the core elements of Domain Specific Languages.

**Hardware-In-The-Loop Test**

Hardware-In-The-Loop Test is a testing method which provides a simulated environment for the system under test.

**HQL**

Hibernate Query Language is similar to SQL but provides access to hibernate data objects.

**HSI**

Hardware-Software Integration Test is a test where the software is integrated on the target hardware.

**HTML**

Hyper Text Markup Language is the standard markup language for web pages.

**HTTP**

Hypertext Transfer Protocol is an application protocol for exchanging hypertext.

**IC**

Integrated Circuit is an electronic component consisting of transistors on a small plate.

**IFE**

In-Flight Entertainment System is a built-in system for video and audio information for passengers in aircrafts.

**IFM**

Interface Module is a Real-Time Tester Entity.

**IMF**

Intermediate Format is a file format which is used to transfer data between programs or systems.

**IMR**

Intermediate Representation is a representation of data which are transferred between programs or systems.

**IP**

Internet Protocol is the foundational communication protocol for the internet.

**JPQL**

Java Persistence Query Language is similar to SQL but provides access to java data objects.

**LEX**

Lexical analyzer ("lexers")

**Make**

Make is a tool for the building process of software.

**Makefile**

Input file for Make tool. Text file which contains rules and variables in order to build a target.



**MC/DC**

Modified Condition/Decision Coverage is a code coverage criterion which needs to be addressed during test activities for certification reasons.

**MetaEdit+**

Modeling Tool for Domain Specific Languages

**MISRA C**

MISRA C is a software development guideline.

**MDSD**

Model-Driven Software-Development is a software development methodology which focuses on models in contrast to computing concepts.

**PAX**

Abbreviation for Passenger

**PHP**

Hypertext Preprocessor is a scripting language for web sites.

**PSAC**

Plan for Software Aspects of Certification, which is necessary for certification.

**RTTE**

RT-Tester Entity

**RTTL**

Real Time Tester Language is an embedded DSL, which is used for testing.

**RFID**

Radio-Frequency Identification is a short range wireless technology.

**RTT**

Real-Time Tester ensures a real-time environment and test platform.

**SMS**

Short Message Service is a standardized text messaging service of mobile communication systems.

**SOAP**

Simple Object Access Protocol is a protocol for exchanging structured information of web services.

**SOE**

Simulated Original Equipment is a virtual instance of an existing system or software.

**SQL**

Abbreviation for Structured Query Language which provides access to databases.

**SUT**

System Under Test is a system which is the target of the testing activities.

**SWI**

Software Integration Test is an integration test on software level.

**TBC**

Abbreviation for To Be Confirmed

**TBD**

Abbreviation for To Be Defined

**TCP**

Transmission Control Protocol is one of the core internet protocols.

**Test Bench**

Test Bench consists of an environment in order to test and verify a system or software.

**ULD**

Unified Loading Devices are standardized containers or pallets which are used to load freight to an aircraft.

**UML**

Unified Modeling Language is a general purpose modeling language.

**UMTS**

Universal Mobile Telecommunications System is a 3G mobile cellular system.

**Unit Test**

Unit testing is a software testing method on source code basis.

**Unix**

Multitasking, multiuser computer operating system

**V-Model**

The V-Model describes a development lifecycle model which includes the project definition, implementation, and integration and test phase.

**VPN**

Virtual Private Network establishes an encrypted virtual network over a public network.

**WIMAX**

Worldwide Interoperability for Microwave Access is a wireless communications standard.

**WSDL**

The Web Services Description Language is an XML based language which describes the functionality of a web-service.

**XMI**

XML Metadata Interchange specifies a standard for exchanging format for UML models. The format is based on XML.

**XML**

Extensible Markup Language is a markup language which is designed to be human- and machine-readable.

**YACC**

Yet Another Compiler Compiler (YACC)