Linköping Studies in Science and Technology

Dissertation No. 1547

Model-Based Verification of Dynamic System Behavior against Requirements

Method, Language, and Tool

by

Wladimir Schamai

Linköping University
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2013

Copyright © 2013 Wladimir Schamai

ISBN 978-91-7519-505-6

ISSN 0345-7524

Thesis No. 1547

October 2013

Electronic version available at:

<http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-98107>

Printed by LiU-Tryck 2013

Abstract

Modeling and simulation of complex systems is at the heart of any modern engineering activity. Engineers strive to predict the behavior of the system under development in order to get answers to particular questions long before physical prototypes or the actual system are built and can be tested in real life.

An important question is whether a particular system design fulfills or violates requirements that are imposed on the system under development. When developing complex systems, such as spacecraft, aircraft, cars, power plants, or any subsystem of such a system, this question becomes hard to answer simply because the systems are too complex for engineers to be able to create mental models of them. Nowadays it is common to use computer-supported modeling languages to describe complex physical and cyber-physical systems. The situation is different when it comes to describing requirements. Requirements are typically written in natural language. Unfortunately, natural languages fail at being unambiguous, in terms of both syntax and semantics. Automated processing of natural-language requirements is a challenging task which still is too difficult to accomplish via computer for this approach to be of significant use in requirements engineering or verification.

This dissertation proposes a new approach to design verification using simulation models that include formalized requirements. The main contributions are a new *method* that is supported by a new *language* and *tool*, along with *case studies*. The method enables verification of system dynamic behavior designs against requirements using simulation models. In particular, it shows how natural-language requirements and scenarios are formalized. Moreover, it presents a framework for automating the composition of simulation models that are used for design verification, evaluation of verification results, and sharing of new knowledge inferred in verification sessions.

A new language called ModelicaML was developed to support the new method. It enables requirement formalization and integrates UML and Modelica. The language and the developed algorithms for automation are implemented in a prototype that is based on Eclipse Papyrus UML, Acceleo, and Xtext for modeling, and OpenModelica tools for simulation. The prototype is used to illustrate the applicability of the new method to examples from industry. The case studies presented start with sets of natural-language requirements and show how they are translated into models. Then, designs and verification scenarios are modeled, and simulation models are composed and simulated automatically. The simulation results produced are then used to draw conclusions on requirement violations; this knowledge is shared using semantic web technology.

This approach supports the development and dynamic verification of cyber-physical systems, including both hardware and software components. ModelicaML facilitates a holistic view of the system by enabling engineers to model and verify multi-domain system behavior using mathematical models and state-of-the-art

simulation capabilities. Using this approach, requirement inconsistencies, incorrectness, or infeasibilities, as well as design errors, can be detected and avoided early on in system development. The artifacts created can be reused for product verification in later development stages.

This work has been supported by EADS Innovation Works, the German Federal Ministry of Education and Research (BMBF), and the Swedish Governmental Agency for Innovation Systems (Vinnova) in the ITEA2 OPENPROD and MODRIO projects, and by SSF and ELLIIT.

Populärvetenskaplig sammanfattning

Modellering och simulering är idag centrala inslag i modern produktutveckling. Man kan göra modeller av en produkt i datorn redan innan den är tillverkad och simulera dess beteende. På så sätt kan man tidigt eliminera felaktigheter på designstadiet och öka kvaliteten.

En viktig frågeställning är om en viss design kan verifieras, dvs om den uppfyller ställda krav på produkten. Detta är vanligen svårt att svara på för komplexa produkter som exempelvis bilar, flygplan och kraftverk. Anledningen är att det är svårt för en person att skapa sig en mental modell av alla aspekter för en så komplex produkt.

Idag är det vanligt att använda datorstödda modelleringsspråk för att beskriva komplexa produkter och system. Situationen är dock en annan för produktkraven. De skrivs typiskt i ett vanligt naturligt språk som svenska eller engelska. Detta är problematiskt eftersom naturliga språk är flertydiga och inte tillräckligt exakta för att krav skrivna i sådana språk entydigt ska kunna tolkas och verifieras av en dator.

För att lösa dessa problem presenterar denna avhandling en ny metod för att verifiera krav med användning av simuleringsmodeller som innehåller formaliserade krav, dvs krav uttryckta i ett exakt formaliserat datorspråk. Avhandlingens forskningsbidrag omfattar en ny metod, ett nytt verktyg med ett nytt modelleringsspråk, samt fallstudier med metoden och verktyget tillämpade på mindre industriella problemställningar.

Denna metod möjliggör automatisk verifiering av krav. Man får reda på om den tänkta produkten, beskriven som en simuleringsmodell, uppfyller kraven för de användningsfall som man modellerat. Dessutom visas hur krav och användningsfall uttryckta i naturligt språk kan skrivas om i modelleringsspråket för att bli formella och exakta. Vidare ingår ett ramverk för att automatisera sammansättning av simuleringsmodeller för designverifiering, utvärdering av verifieringsresultat, och genererande av rapporter med kunskap som erhållits under verifieringsprocessen.

Det nya språket, kallat ModelicaML, utvecklades under avhandlingsarbetet för att stödja den nya metoden. Det möjliggör exakt formell beskrivning av krav och baseras på modelleringsspråken UML, känt för mjukvarumodellering, och Modelica, mest känt för hårdvarumodellering. Därmed kan ModelicaML göra cyberfysikalisk modellering, dvs uttrycka modeller och krav för produkter som innehåller både programvara och hårdvara. Själva simuleringen utförs t.ex. med OpenModelica som är en öppen källkodsimplementation av utvecklingsverktyg för Modelicaspråket.

Sammanfattningsvis ger den nya metoden och verktyget en helhetsyn för utveckling av komplexa produktsystem innehållande både hårdvara och mjukvara. Metoden ger stöd till utvecklingsingenjörer som behöver modellera och verifiera komplexa produktmodeller på ett matematiskt exakt sätt. Med denna metod kan fel

och motsägelser i kraven samt rena designfel upptäckas och åtgärdas tidigt under produktutvecklingen.

Detta arbete har stötts av EADS Innovation Works och Vinnova samt Tyska undervisnings och forskningsdepartementet (BMBF) inom ITEA2 OPENPROD och MODRIO projekten, samt av SSF och ELLIIT.

Acknowledgements

First of all, I am deeply indebted to my main advisor, Prof. Peter Fritzson. I want to thank him for placing his confidence in me, for motivating me, and, most importantly, for *inspiring* me. He once said to me, “People do research because they believe in it.” This sentence kept me going many times when I saw no light at the end of the tunnel. His supervision helped me to become a more complete researcher.

In the same way, I want to thank Prof. Chris Paredis for his strong and effective support and guidance. A few times he had to put me back on track in order to stop me from getting lost in details and unimportant tasks. Also, Prof. Paredis said, “No one in the room knows your work better than you do.” This sentence helped me not to keel over from excitement while standing in front of an audience and to present my results adequately.

I greatly appreciate the technical and motivating discussions I had with Prof. Kristian Sandahl and Adrian Pop, who were immediately able to find the puzzle pieces I was missing when I showed up with fuzzy questions and absolutely no patience.

I thank Axel Mauritz and Martine Callot for their trust in me and for creating adequate working conditions for my research. I also want to thank my colleagues Philipp Helle, Stefan Richter, and Andreas Mitschke for being open-minded and for helping me to turn my need for discussions into productive work.

Last, but certainly not least, I thank my family—Saskia, Rocco, and Ava—for being with me. You are the light of my life.

August 2013, Hamburg, Germany
Wladimir Schamai

Table of Contents

Chapter 1	Introduction.....	1
1.1	Motivation	1
1.2	Research Method	2
1.3	Research Questions.....	3
1.4	Contributions	4
1.5	Publications	4
1.6	Dissertation Structure	6
Chapter 2	Background	9
2.1	What is Design Verification?.....	9
2.2	What is a Requirement?.....	10
2.3	Model-Based Systems Engineering (MBSE)	11
2.4	Verification Approaches.....	12
2.4.1	<i>Model Checking</i>	13
2.4.1.1	Numerical Model Checking	13
2.4.1.1	Statistical Model Checking	13
2.4.2	<i>Runtime Verification</i>	14
2.4.3	<i>Testing</i>	15
2.4.1	<i>Model-Based Testing</i>	16
2.4.1	<i>Simulation</i>	16
2.5	System Modeling and Simulation Languages	16
2.5.1	<i>UML-Based Languages for Descriptive Modeling</i>	17
2.5.2	<i>Modelica for Dynamic Behavior Modeling and Simulation</i>	18
2.6	Problem Description	19
Chapter 3	Model-Based Design Verification Method	21
3.1	Motivation	21
3.2	Scope	22
3.3	Using Models for Design Verification	22
3.4	The Role of Scenarios.....	23
3.5	Analysis Approach for Design Verification	24
3.5.1	<i>Static Analysis: Model Checking</i>	24
3.5.2	<i>Dynamic Analysis: Simulation-Based Testing</i>	25
3.6	Satisfaction vs. Violation of a Requirement	27
3.7	vVDR Method Description	28
3.7.1	<i>Overview</i>	29
3.7.2	<i>Roles</i>	32
3.7.2.1	Requirements Analyst.....	32
3.7.2.2	System Designer	33
3.7.2.3	Tester	33
3.7.2.4	Difference to Traditional Role Responsibilities	33
3.7.3	<i>Task: Formalize Requirements</i>	34

3.7.3.1	Requirement Violation Monitors	34
3.7.3.2	From Natural-Language Statement to Violation Monitor	35
3.7.3.3	Testing of Violation Monitor Models	41
3.7.3.4	Added Value	42
3.7.3.5	Requirements Preselection	44
3.7.3.6	vVDR Language Support	44
3.7.4	<i>Task: Formalize Design</i>	45
3.7.5	<i>Task: Formalize Scenarios</i>	46
3.7.5.1	Relations between Scenarios and Requirements	48
3.7.5.2	Manually vs. Automatically Generated Scenarios	48
3.7.5.3	Testing of Scenario Models	49
3.7.5.4	Added Value	49
3.7.5.5	vVDR Language Support	49
3.7.6	<i>Additional Models</i>	49
3.7.7	<i>Task: Create Verification Models</i>	52
3.7.7.1	Added Value	54
3.7.7.2	vVDR Language Support	54
3.7.8	<i>Task: Analyze Verification Models and Create Report</i>	55
3.7.9	<i>Task: Analyze Verification Results</i>	55
3.8	What Can be Automated?	56
3.9	Conclusion	56
Chapter 4	Framework for Automation	59
4.1	Introduction	59
4.2	Bindings Concept	60
4.2.1	<i>Model Instantiation</i>	61
4.2.2	<i>Basic Concepts</i>	62
4.2.3	<i>Binding Operations</i>	65
4.2.3.1	Mediator Operation	65
4.2.3.2	Provider Operation	67
4.2.3.3	Client Operation	68
4.2.3.4	Overwriting of Bindings	69
4.2.4	<i>Validity of Binding Operations</i>	69
4.2.5	<i>Preferred Bindings</i>	71
4.3	Applications for Bindings	71
4.3.1	<i>Automated Composition of Verification Models</i>	71
4.3.2	<i>Discovery of Relations between Scenarios and Requirements</i>	73
4.3.3	<i>Traceability of Requirements, Designs and Scenarios</i>	74
4.3.4	<i>Detection of Redundant or Conflicting Requirements</i>	75
4.3.4.1	Redundant Requirements	75
4.3.4.2	Conflicting Requirements	75
4.3.5	<i>Impact Analysis</i>	76
4.4	vVDR Enhancements	76
4.5	vVDR Language Support	78
4.6	Algorithm for Generating Verification Models	79
4.7	Algorithm for Inferring Bindings	86
4.8	Conclusion on Verification Results	90
4.9	Verification Session Report Generation	91

4.10	Conclusion	97
Chapter 5	Language and Tool.....	99
5.1	Introduction	99
5.2	Related Work.....	100
5.3	UML-to-Modelica Mapping	100
5.4	vVDR Support	101
5.5	Graphical Modeling with ModelicaML.....	102
5.6	Model Transformation Implementation.....	104
5.6.1	<i>ModelicaML to Modelica</i>	104
5.6.2	<i>Modelica to ModelicaML</i>	105
5.7	Resolution of UML State Machines Semantics Issues	105
5.7.1	<i>Simple Example</i>	107
5.7.2	<i>State Machines in ModelicaML</i>	109
5.7.2.1	Transformation of State Machines to Modelica Code.....	111
5.7.2.2	Interrelation with Other Class Behavior Definitions.....	112
5.7.2.3	Combining Continuous-Time, Event-Based or Discrete-Time Behavior.....	112
5.7.2.4	Event Processing (Run-To-Completion Semantics Applicability).....	114
5.7.2.5	Subset of UML2 State Machine Concepts Supported in ModelicaML	115
5.7.2.6	Support of UML State Machine Graphical Notation	115
5.7.2.7	Supported UML State Machine Concepts.....	115
5.7.3	<i>State Machines Execution Semantics Issues Discussion</i>	117
5.7.3.1	Issues with Instantaneous States: Deadlocks (Infinite Looping).....	117
5.7.3.2	Issues with Concurrency When Using Event Queues	117
5.7.3.3	Issue with Concurrent Execution in Regions	121
5.7.3.4	Issues with Conflicting Transitions.....	123
5.7.3.5	Priorities for State-Outgoing Transitions	123
5.7.3.6	Priority Schema for Conflicting Transitions	124
5.7.3.7	Issues with Inter-Level Transitions.....	125
5.7.3.8	Issues with Fork and Join.....	127
5.7.4	<i>Related Work</i>	128
5.8	ModelicaML Prototype.....	130
5.9	Conclusion	132
Chapter 6	Application Examples	133
6.1	Introduction	133
6.2	Two-Tank System.....	133
6.2.1	<i>Formalizing Requirements</i>	134
6.2.2	<i>Formalizing Design</i>	138
6.2.3	<i>Formalizing Scenarios</i>	140
6.2.4	<i>Specifying Bindings</i>	141
6.2.5	<i>Generating Verification Models</i>	143
6.2.5.1	Options for Models Generation.....	144
6.2.5.2	Discovering Relations Between Scenarios and Requirements	145
6.2.5.3	Generation of Verification Models Based on Explicit Relations	150
6.2.5.4	Generated Verification Models.....	150
6.2.6	<i>Generating Verification Session Reports</i>	154
6.2.7	<i>Traceability Between Requirements, Design, and Scenarios</i>	154

6.3	Power Plant Cooling System	155
6.3.1	<i>Formalizing Requirements</i>	156
6.3.2	<i>Formalizing Design</i>	166
6.3.3	<i>Formalizing Scenarios</i>	167
6.3.4	<i>Specifying Bindings</i>	168
6.3.5	<i>Generating Verification Models and Verification Results</i>	169
6.4	Conclusion	171
Chapter 7	Conclusion and Future Work	173
7.1	Review	173
7.2	Major Contributions.....	174
7.3	Validity	175
7.4	Future Work	175
Appendix A	ModelicaML Profile	179
A.1	Class Constructs.....	180
A.2	Composite Constructs	184
A.3	Behavior Constructs.....	189
A.4	Relations Constructs	196
A.5	Requirement Constructs.....	199
A.6	Simulation Constructs.....	200
A.7	Annotation Constructs	201
A.8	Verification	204
A.9	Bindings	206
A.10	Model References	208
A.11	ModelicaPredefinedTypes	209
A.12	ModelicaPredefinedEnumerations	210
A.13	Enhancements of UML Graphical Notation	210
Appendix B	Grammar for Binding Operations	215
	Client Operation	215
	Mediator Operation.....	216
	Provider Operation	217
Appendix C	Ontology for Verification Report	219
	vVDR Verification Ontology	219
	Examples of SPARQL Queries	222
	Violated Requirements (Partial Evidence) Query	223
	Not Violated Requirements (Partial Evidence)	224
	Not Evaluated Requirements (Partial Evidence)	225
	Scenarios for Stimulating Designs.....	225
	Scenarios for Evaluating Requirements.....	226
Bibliography		i

Table of Figures

<i>Figure 1: vVDR overview</i>	<i>31</i>
<i>Figure 2: Information flow in vVDR</i>	<i>32</i>
<i>Figure 3: Testing a requirement violation monitor</i>	<i>42</i>
<i>Figure 4: Concept of verification models</i>	<i>53</i>
<i>Figure 5: Example of instance hierarchy</i>	<i>62</i>
<i>Figure 6: Basic concept of client, mediator and provider.....</i>	<i>63</i>
<i>Figure 7: Bindings concept in vVDR.....</i>	<i>63</i>
<i>Figure 8: Illustration of the verification models generation algorithm.....</i>	<i>73</i>
<i>Figure 9: vVDR steps contributing to the specification of bindings.....</i>	<i>77</i>
<i>Figure 10: Enhanced vVDR overview</i>	<i>78</i>
<i>Figure 11: Example of verification session report generated</i>	<i>92</i>
<i>Figure 12: vVDR verification results ontology.....</i>	<i>93</i>
<i>Figure 13: vVDR verification results ontology: Inverse object properties.....</i>	<i>94</i>
<i>Figure 14: vVDR verification results ontology: Data properties.....</i>	<i>94</i>
<i>Figure 15: vVDR ontology with individuals.....</i>	<i>95</i>
<i>Figure 16: vVDR ontology with inferred data.....</i>	<i>95</i>
<i>Figure 17: SPARQL query result for "Violated Requirements" view</i>	<i>96</i>
<i>Figure 18: ModelicaML diagrams</i>	<i>104</i>
<i>Figure 19: ModelicaML prototype technology.....</i>	<i>105</i>
<i>Figure 20: Example of a simple state machine owned by a class.....</i>	<i>108</i>
<i>Figure 21: Modelica code for the simple state machine.....</i>	<i>109</i>
<i>Figure 22: State Machine of the tank</i>	<i>113</i>
<i>Figure 23: State Machine of the controller</i>	<i>113</i>
<i>Figure 24: Example of state-dependent equations</i>	<i>114</i>
<i>Figure 25: Deadlocks (infinite looping) example.....</i>	<i>117</i>
<i>Figure 26: Events queue issue 1</i>	<i>118</i>
<i>Figure 27: Events queue issue 1 simulation</i>	<i>119</i>
<i>Figure 28: Events queue issue 2.....</i>	<i>120</i>
<i>Figure 29: Events queue issue 2 simulation</i>	<i>120</i>
<i>Figure 30: Same model in ModelicaML.....</i>	<i>121</i>
<i>Figure 31: Definition of priority for parallel regions</i>	<i>122</i>
<i>Figure 32: Priority definition for state-outgoing transitions.....</i>	<i>124</i>
<i>Figure 33: Transitions at a higher level have higher priority</i>	<i>124</i>
<i>Figure 34: Inter-level transition example.....</i>	<i>126</i>
<i>Figure 35: Fork and join example.....</i>	<i>127</i>
<i>Figure 36: ModelicaML prototype GUI example.....</i>	<i>131</i>
<i>Figure 37: Application example 1: Simple two-tank system.....</i>	<i>134</i>
<i>Figure 38: Application example 1: Req. 001 formalized.....</i>	<i>135</i>
<i>Figure 39: Application example 1: Req. 002 formalized.....</i>	<i>135</i>
<i>Figure 40: Application example 1: Req. 003 formalized.....</i>	<i>136</i>
<i>Figure 41: Application example 1: Req. 004 formalized using a state machine</i>	<i>137</i>
<i>Figure 42: Example of a verification scenario</i>	<i>141</i>

Figure 43: Application example 1: Bindings for requirements	141
Figure 44: Application example 1: Bindings for design and scenarios	142
Figure 45: Application example 1: Client operation	143
Figure 46: Application example 1: Provider operation	143
Figure 47: Options dialog: the model generation options for relations discovery	145
Figure 48: Selecting scenarios	146
Figure 49: Selecting requirements	147
Figure 50: Defining bindings manually	148
Figure 51: Created verification models and simulation options	149
Figure 52: Relations discovered between scenarios and requirements	149
Figure 53: Options dialog for model generation based on explicit relations	150
Figure 54: Naming conventions for generated models	152
Figure 55: Storing bindings in Modelica	152
Figure 56: Verification session report GUI and HTML	153
Figure 57: GUI for traceability based on bindings	155
Figure 58: Application example 2: Schematic overview	156
Figure 59: Application example 2: Req. 002 properties	157
Figure 60: Application example 2: Req. 002 status	157
Figure 61: Application example 2: Req. 003 precondition	157
Figure 62: Application example 2: Req. 0083 issue	158
Figure 63: Application example 2: Req. 002 issue resolution	158
Figure 64: Application example 2: Req. 0083 properties	159
Figure 65: Application example 2: Req. 0083 functions	159
Figure 66: Application example 2: Req. 0083 status	160
Figure 67: Application example 2: Req. 0083 simulation results	160
Figure 68: Application example 2: Req. 013 properties	161
Figure 69: Application example 2: Req. 013 calculations	161
Figure 70: Application example 2: Req. 013 functions	162
Figure 71: Application example 2: Req. 013 calculations	162
Figure 72: Application example 2: Req. 013 status	162
Figure 73: Application example 2: Req. 013 simulation results	163
Figure 74: Application example 2: Req. 007 properties	163
Figure 75: Application example 2: Req. 007 calculations	164
Figure 76: Application example 2: Req. 007 status	164
Figure 77: Application example 2: Req. 007 simulation results 1	165
Figure 78: Application example 2: Req. 007 simulation results 2	166
Figure 79: Application example 2: System design model import	167
Figure 80: Application example 2: Formalized scenarios	168
Figure 81: Application example 2: Example of requirement bindings	168
Figure 82: Application example 2: Example of system model and scenario bindings	169
Figure 83: Application example 2: Verification model generated	169
Figure 84: Application example 2: Example of an additional model	170
Figure 85: Application example 2: Verification session report generated	171
Figure 86: ModelicaML class constructs	180

<i>Figure 87: ModelicaML composite constructs</i>	<i>184</i>
<i>Figure 88: ModelicaML constructs for capturing Modelica code.....</i>	<i>189</i>
<i>Figure 89: ModelicaML constructs for state machine</i>	<i>190</i>
<i>Figure 90: ModelicaML constructs for conditional equations or algorithm.....</i>	<i>191</i>
<i>Figure 91: ModelicaML relations constructs</i>	<i>196</i>
<i>Figure 92: ModelicaML requirement constructs.....</i>	<i>199</i>
<i>Figure 93: ModelicaML simulation construct.....</i>	<i>200</i>
<i>Figure 94: ModelicaML annotation constructs.....</i>	<i>201</i>
<i>Figure 95: ModelicaML constructs for design verification.....</i>	<i>204</i>
<i>Figure 96: ModelicaML bindings constructs</i>	<i>206</i>
<i>Figure 97: ModelicaML constructs for Modelica code synchronization.....</i>	<i>208</i>
<i>Figure 98: Proposed class notation</i>	<i>211</i>
<i>Figure 99: Proposed notation for the indication of incomplete view</i>	<i>211</i>
<i>Figure 100: Proposed notation for capturing “for loop” statements</i>	<i>212</i>
<i>Figure 101: Proposed notation for capturing “if” statements</i>	<i>213</i>
<i>Figure 102: Proposed notation for capturing if equations.....</i>	<i>214</i>
<i>Figure 103: SPARQL query 1 result for "Violated Requirements" view</i>	<i>223</i>
<i>Figure 104: SPARQL query 2 result for "Violated Requirements" view</i>	<i>224</i>
<i>Figure 105: SPARQL query result for "Not Violated Requirements" view</i>	<i>225</i>
<i>Figure 106: SPARQL query result for "Not Evaluated Requirements" view.....</i>	<i>225</i>
<i>Figure 107: SPARQL query result for "Scenarios for Stimulating Design" view</i>	<i>226</i>
<i>Figure 108: SPARQL query result for "Scenarios for Evaluating Requirements" view</i>	<i>227</i>

Chapter 1

Introduction

1.1 Motivation

Modeling and simulation of complex systems is at the heart of any modern engineering activity. Engineers strive to predict the behavior of the system under development in order to get answers to particular questions long before physical prototypes or the actual system are built and can be tested in real life. One important question is whether a particular system design satisfies or violates requirements that are imposed on the system under development. When developing complex systems, such as spacecraft, aircraft, cars, power plants, or any subsystem of such systems, this question becomes hard to answer simply because the systems are too complex for engineers to be able to create mental models of them.

In order to cope with complexity, engineers use machines—computers—which are better than the human brain when it comes to mathematical calculations or solving of combinatorial problems, overcoming “... one of the most fundamental bounds on human cognition: our inability to simulate mentally the dynamics of complex nonlinear systems.” (Serman, 2002). “Formal models, grounded in data and subjected to a wide range of tests, lead to more reliable inferences about dynamics and uncover errors in our mental simulations” (Serman, 2002).

Creating formal models using computers require languages with precisely defined syntax and semantics. Languages that are understandable by humans and by computers are referred to as *modeling languages*. Using modeling languages to describe systems is a common practice nowadays (Fritzson, 2004) (Cellier & Kofman, 2006).

This is different when it comes to describing requirements. Requirements are typically written in natural language (Mich, Mariangela, & Pierluigi, 2004). Natural language is understood by everyone involved in the system development or certification process. Formal languages and methods are still not widely used in

industry (Woodcock, Larsen, Bicarregui, & Fitzgerald, 2009). Moreover, formal languages are often too costly to be introduced and used by engineers untrained in formal methods. For those reasons natural language is still the main means for writing system requirements (Mich, Mariangela, & Pierluigi, 2004). Unfortunately, natural languages fail at being unambiguous, in terms of both syntax and semantics. Automated resolution of natural language ambiguities is a challenging task (Ambriola & Gervasi, 1997) that can be accomplished by computers only with great difficulty, and is therefore not of significant use in requirements engineering (Ryan, 1993, p. 240) or verification¹.

The purpose of this dissertation is first to identify what is necessary to enable model-based verification of designs against natural-language requirements, and second, to develop a method and tool to support and partially automate the task of design verification using simulations. Here when we speak about models, we refer to formal models written in some type of modeling language that can be processed by computers.

1.2 Research Method

According to (Denning, et al., 1989) research in computer science or engineering is a mixture of paradigms that are rooted in mathematical sciences, natural sciences and engineering. The fundamental question is “What can be (efficiently) automated?” (Denning, et al., 1989).

The research method used in this dissertation mainly follows the *design* paradigm rooted in engineering (Denning, et al., 1989). The steps of this method include formulating questions and requirements, describing the problem, specifying and designing the solution, and testing the solution in order to show how the problem can be solved and how questions are addressed. In this dissertation, in addition to the research questions, one main hypothesis is formed and tested in case studies using the prototype developed.

The research is performed as follows. First the work is motivated in Section 1.1. Section 1.3 presents research questions and hypothesis that define the research scope and interest. In Chapter 2 the literature is reviewed followed by the problem description. Then the problem is analyzed, a solution is designed and requirements on the system (i.e., a new modeling and simulation environment) are elicited (see Chapter 3, Chapter 4). The developed solution is prototyped (see Chapter 5) and

¹ An approach for using natural language processing for analyzing requirements or generating test cases is presented in (Boddu, Guo, Mukhopadhyay, & Cukic, 2004). It concludes that natural language processing can be used requirements analysis. However, it reports on the same difficulties with handling natural-language requirements of inadequate quality.

used in Chapter 6 for providing answers to research questions and for testing the main hypothesis. The obtained results are discussed and published (see Section 1.5) and further ideas for future work are presented (see Chapter 7).

1.3 Research Questions

The research questions addressed in this dissertation are the following, where two questions have associated sub-questions:

Research Question 1. Which model infrastructure is required to enable the verification of system designs against system requirements using models?

- What should be the characteristics of a method that allows for a systematic and efficient approach for a model-based design verification?
- What are the roles and tasks?
- What are the required modeling artifacts?

Research Question 2. How should natural-language requirements be represented in the context of model-based design verification?

- Can we model requirements so that they can be unambiguously interpreted by computers and automatically evaluated in system simulations?
- How can we determine if a particular system design can be verified against a particular requirement using models?
- How can we determine the level of detail of the system model required for the verification?

Research Question 3. Which steps of the method should be automated to increase process efficiency and avoid errors?

Research Question 4. What should be the characteristics of a language that enables modeling and simulation of complex cyber-physical systems, and facilitates the verification of system designs against requirements?

Following the fundamental question of “What can be (efficiently) automated?” (Denning, et al., 1989) (see Section 1.2), the main **hypothesis** tested in this dissertation is the following:

If a method is developed that enables engineers to do the following:

- *to formalize natural-language requirements into executable models,*

- *to model the system design such that it can provide all data needed for formalized requirements,*
- *to model scenarios that stimulate the system such that one or more requirements can be evaluated,*
- *and to capture potential interactions between requirements, system designs, and other required models,*

then it is possible to achieve the following:

- *to automate the process of creating and simulating models for the verification of system designs against requirements throughout the system design cycle,*
- *and to automatically draw conclusions based on the simulation results produced.*

The purpose of this dissertation is to address the above research questions by developing a method, language and tool to support model-based verification of system designs against formalized natural-language requirements. The practical evidence is provided by the prototype developed (see Chapter 5), which is used to test the hypothesis and to demonstrate the applicability of the new approach to industrial problems (see Chapter 6).

1.4 Contributions

This dissertation presents a new approach to design verification using simulation models, including formalized requirements. The main contributions are a new *system design verification method* (see Chapter 3) that is supported by a new *modeling language* and a new *tool* (see Chapter 4). Furthermore, new *algorithms* for an *automated model composition*, as well as an approach for *automatically drawing conclusions* based on simulation results generated and *sharing the knowledge* inferred in verification sessions, are presented in Chapter 4.

See Section 7.2 for an elaborated summary of the major contributions.

1.5 Publications

Some of the material in this dissertation is based in part on the following publications:

1. Schamai, W. (2009). Modelica Modeling Language (ModelicaML): A UML Profile for Modelica. *Technical Report*. Linköping University Electronic Press.
2. Schamai, W., Fritzson, P., Paredis, C., & Pop, A. (2009). Towards Unified System Modeling and Simulation with ModelicaML: Modeling of

Executable Behavior Using Graphical Notations. In *Proceedings 7th Modelica Conference*. Como.

- Main idea: Wladimir Schamai
 - Text and editing: Wladimir Schamai
 - Discussions, validation and proofreading: Peter Fritzson, Chris Paredis, and Adrian Pop
3. Schamai, W., Helle, P., Fritzson, P., & Paredis, C. (2011). Virtual Verification of System Designs against System Requirements. In *Models in Software Engineering* (pp. 75-89). Springer Berlin Heidelberg.
 - Main idea: Wladimir Schamai
 - Text and editing: Wladimir Schamai
 - Discussions, validation and proofreading: Philipp Helle, Peter Fritzson, Chris Paredis, and Adrian Pop
 - Philipp Helle
 - State-of-the-art research contribution
 - Editing in Latex
 4. Schamai, W., Fritzson, P., & Paredis, C. (2013). Translation of UML State Machines to Modelica: Handling Semantic Issues. In *SIMULATION* (Volume 89 Issue 4 ed., pp. 498 - 512).
 - Main idea: Wladimir Schamai
 - Text and editing: Wladimir Schamai
 - Discussions, validation, and proofreading: Peter Fritzson, Chris Paredis
 5. Schamai, W., Fritzson, P., Paredis, C., & Helle, P. (2012). ModelicaML Value Bindings for Automated Model Composition. In S. f. International (Ed.), *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*.
 - Main idea: Wladimir Schamai
 - Text and editing: Wladimir Schamai
 - Discussions, validation, and proofreading: Peter Fritzson, Chris Paredis

Moreover, the developed method and tool was used to give the following tutorials:

- Using the MDT-ModelicaML Eclipse Plugin for Modelica Development and UML-Modelica Systems Engineering. *2009 Modelica Conference*. Instructors: Adrian Pop and Wladimir Schamai.
- Model-Based Development Using the ModelicaML (Modelica-SysML) and Modelica MDT Eclipse Plugin. *2010 MODPROD*. Instructors: Wladimir Schamai and Adrian Pop.
- Model-Based Development Using the ModelicaML (Modelica-SysML) and Modelica MDT Eclipse Plugin. *2011 MODPROD*. Instructors: Wladimir Schamai and Adrian Pop.

- ModelicaML Tutorial - Virtual Verification of System Design against System Requirements. *2011 Modelica Conference*. Instructors: Wladimir Schamai and Adrian Pop.
- Automated Requirements Verification Model Composition in ModelicaML - Using ModelicaML Value Bindings and Model-Based Development Using Modelica MDT Eclipse Plugin with Run-Time Debugging. *2012 MODPROD*. Instructors: Wladimir Schamai and Adrian Pop.
- Model-Based Development Using ModelicaML Value Bindings for Model Composition and Requirements Traceability. *2013 MODPROD*. Instructor: Wladimir Schamai.

Discussions during the tutorial sessions as well as the feedback provided supported the validation of results presented in this dissertation.

The following conference papers were published during the dissertation period as well. However, they are peripheral to the addressed topic and are not included in the dissertation.

- Kessler, C. W., Schamai, W., & Fritzson, P. (2010). Platform-independent Modeling of Explicitly Parallel Programs. In *Architecture of Computing Systems (ARCS)* (pp. 1-11).
- Helle, P., & Schamai, W. (2010). Specification Model Based Testing in the Avionic Domain - Current Status and Future Directions. *Model-Based Testing MBT 2010*, 84.
- Paredis, C., Bernard, Y., Burkhart, R. M., de Koning, H. P., Friedenthal, S., Fritzson, P., ... & Schamai, W. (2010). An Overview of the SysML-Modelica Transformation Specification. In *2010 INCOSE International Symposium*.
- Christoffers J., Schamai W. (2010). Seamless Transition from Functional SysML Specification to Virtual Prototyping using Saber. *Saber eUpdate June 2010*, Munich.
- Myers, T., Schamai, W., & Fritzson, P. (2011). Comodeling Revisited: Execution of Behavior Trees in Modelica. In *EOOLT* (pp. 97-106).
- Asghar, S. A., Tariq, S., Torabzadeh-Tari, M., Fritzson, P., Pop, A., Sjölund, M., ... & Schamai, W. (2011). An Open-source Modelica Graphical Editor Integrated with Electronic Notebooks and Interactive Simulation. In *8th International Modelica Conference* (pp. 739-747).

1.6 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 offers an overview on the state of the art and describes the problem. Chapter 3 presents a new method for

design verification, independent of any particular language or tool. It also discusses how some steps of the method can be automated. Similarly, Chapter 4 theoretically presents the concepts for automation without a concrete implementation. Next, Chapter 5 presents a new language and tool/environment designed to support the new approach. Chapter 6 provides examples of applying the new approach using the language and tool. Finally, Chapter 7 presents the conclusions and suggestions for future work.

Chapter 2

Background

2.1 What is Design Verification?

Several definitions and explanations of the terms *verification* and *design verification* are available in international standards and practical guides, such as (INCOSE, 2006), (NASA, 2007), and (ISO/IECTR19760, 2003). These definitions address system, hardware, and software development:

- Verification is “confirmation, through the provision of objective evidence, that specified requirements have been fulfilled. NOTE Verification in a life cycle context is a set of activities that compares a product of the life cycle against the required characteristics for that product. This may include, but is not limited to, specified requirements, design description and the system itself.” (ISO/IEC15288, 2008, p. 7)
- “The purpose of verification is to ascertain that each level of the implementation meets its specified requirements. The verification process ensures that the system implementation satisfies the validated requirements. Verification consists of inspections, reviews, analyses, tests, and service experience applied in accordance with a verification plan.” (SAE, 1996)
- “The verification process provides assurance that the hardware item implementation meets the requirements. Verification consists of reviews, analyses and tests applied as defined in the verification plan. The verification process should include an assessment of the results.” (RTCA/DO-254, 2000)
- “The purpose of the software verification process is to detect and report errors that may have been introduced during the software development process.” (Federal Aviation Authority DO-178B, 1999)
- “Verification demonstrates, through a dedicated process, that the System meets the applicable requirements and is capable of sustaining its operational role during the project life cycle.” (ESA, 1996)

- Design verification: “The project shall perform the tasks of design verification for the purpose of assuring that
 - a) The requirements of the lowest level of the design architecture, including derived requirements, are traceable to the verified functional architecture.
 - b) The design architecture satisfies the validated requirements baseline.” (IEEE1220, 2005)

All the listed definitions, indicate that the purpose of verification is to determine whether the system design or the final system product complies with the specified requirements.

2.2 What is a Requirement?

According to (ISO 9000, 2005, p. 19), a requirement is a: “need or expectation that is stated, generally implied or obligatory...”

NOTE 1 “Generally implied” means that it is custom or common practice for the organization (3.3.1), its customers (3.3.5) and other interested parties (3.3.7), that the need or expectation under consideration is implied...

NOTE 3 A specified requirement is one that is stated, for example in a document (3.7.2).

NOTE 4 Requirements can be generated by different interested parties (3.3.7).

...

3.12.1

requirement

expression in the content of a document conveying criteria to be fulfilled if compliance with the document is to be claimed and from which no deviation is permitted”.

In industrial projects for developing complex physical systems, requirements are typically written in natural language and express a need to be satisfied or a constraint to be fulfilled by the system under development². Examples of natural-language requirement statements include:

² In (Loniewski, Insfran, & Abrahao, 2010) a systematic review of requirements engineering techniques in the context of software development is presented. It points out that there are approaches that use different kinds of models for expressing requirements. However, at the same time, it points out that most of the review approaches are in academic context and use theoretical examples, and that new approaches are still needed in industrial context.

- *“The ambulance control system shall be able to handle up to 100 simultaneous emergency calls.”*
- *“The ambulance driver shall not be placed in breach of national road regulations.”*
- *“The communications system shall sustain telephone contact with not less than 10 callers while in the absence of external power.”* (Hull, Jackson, & Dick, 2005)

A requirement may include references to further readings for explanations or rationale. In addition to the actual requirement statement, each requirement should have a unique identifier and title and may include further classification attributes as suggested by Hull, Jackson and Dick 2005, 78 and ReqIF 2011 in order to provide relevant information in a structured way. Practical guidance for writing *good* requirements is well established (see (Hull, Jackson, & Dick, 2005, pp. 73-86), (Firesmith, 2003), or (INCOSE, 2006, p. 76)) and is outside of scope of this dissertation.

Requirements can be classified into basic types (e.g. “Functional, performance, quality factor, environment, interface, constraint...” according to (Hull, Jackson, & Dick, 2005, p. 78). However, in practice, this is a challenging task because one requirement may address multiple aspects at the same time. Further, classification depends on the domain. An execution time requirement for a function in the domain of web service software may be classified as performance, while the same requirement from the perspective of a real-time embedded system may be considered functional, because computing in time is a basic function of an embedded real-time system.

It is common practice nowadays to include natural-language requirements in models as text and to link them to design or test artifacts (OMG SysML, 2012). Authoring and management of natural-language requirements is supported by commercial tools (INCOSE, 2013). A standard for exchanging textual requirements and their interrelations is defined in (OMG ReqIF, 2011).

2.3 Model-Based Systems Engineering (MBSE)

Model-Based Systems Engineering (MBSE) is defined in (INCOSE, 2007) as “the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.”

An overview of existing methodologies used in industry is given in (Estefan, 2007). Some of them use standardized languages, such as UML (OMG UML, 2011) or SysML (OMG SysML, 2012) for system modeling.

This dissertation follows the MBSE paradigm that proposes to use models as primary engineering artifacts instead of written documents or informal notations. The approach developed in this dissertation includes the new ModelicaML language (see Chapter 5), which extends existing MBSE languages, and supports the new method (see Chapter 3) for formally expressing and verifying requirements.

2.4 Verification Approaches

There are many different approaches and techniques that can be used to gather evidence that a model complies with the specified requirements. (Balci, 1995, p. 152) presents a classification of different techniques, ranging from informal techniques (i.e., those that are mainly based on human reasoning) to formal (i.e., those that are based on mathematical proof of correctness).

Human reasoning includes analyzing system requirements or design and using engineering judgment (based on the knowledge and experience of the engineers involved) in order to determine the completeness and correctness of the descriptions and to obtain evidence or to predict that the system will ultimately satisfy its requirements. We have limited capabilities to perform this mentally with complex systems (Stermann, 2002). It is natural that engineers use computer-aided techniques that can efficiently handle such complexity.

Computer-aided techniques that are relevant for verification can be classified into two main categories: static and dynamic analysis.

When using models as the main artifacts³, the static analysis approach examines the model definition. It neither translates the program into an executable form nor executes the model. In contrast, “dynamic analysis derives properties that hold for one or more executions by examination of the running program (usually through program instrumentation)” (Ball, 1999). In dynamic analysis, models are translated into executable form and executed in order to obtain the predicted system behavior and to be able to compare it to the specified properties.

Static and dynamic analysis are complementary in terms of completeness (undiscovered errors vs. infeasible paths), scope, and precision (executing the actual program vs. analyzing an abstracted version of the actual program) as pointed out by (Ball, 1999). The advantage of dynamic analysis is the ability to execute long paths on complex models and to discover relations in a larger scope, as well as the ability to easily relate inputs to outputs. However, “dynamic analysis cannot prove that a program satisfies a particular property, it can detect violations of properties as well as provide useful information to programmers about the behavior of their programs” (Ball, 1999).

³ This is in contrast to, for example, software code in a programming language.

Model checking is a well-known automated static analysis technique. Runtime verification, software testing, or system simulation are examples of dynamic analysis.

2.4.1 Model Checking

Model checking can be divided into numerical and statistical approaches. In the numerical approach the system model is checked against properties using symbolic and numeric computation; an exhaustive search is performed over the entire problem space. The answer is accurate and holds for any situation captured by the model that is being checked. Statistical model checking uses simulation rather than exhaustive search. Statistical model checking provides probabilistic guarantees of property correctness (Sen, Viswanathan, & Agha, 2004).

2.4.1.1 Numerical Model Checking

Model checking (Clarke, Grumberg, & Peled, 1999), (Baier & Katoen, 2008) is an automatic verification technique applicable to finite-state systems “for which all computation can exhaustively be enumerated” (Leucker & Schallhart, 2009). Using exhaustive search techniques, model checking proves that a specified property holds for every execution, or else provides counterexamples.

The main limitation of model checking is the state explosion problem (Baier & Katoen, 2008, pp. 15, 77 ff), which results from generating the entire state space of the system in order to be able to analyze all possible executions. This limitation and the prerequisite that the system has to be modeled in finite-state space make it difficult to apply model checking to the analysis of physical system behavior typically described by complex differential algebraic equation systems that include continuous variables and functions (Baier & Katoen, 2008, p. 15): i.e., systems that (theoretically) can have an infinite number of possible states.

2.4.1.1 Statistical Model Checking

Statistical model checking is a young research field. (Legay & Delahaye, 2010) and (Younes, 2005) provide an introduction to this approach, where it is assumed that the system model is executable and has no nondeterministic behavior, and where the probability distribution for system behavior is known. To check properties, this approach simulates the system for a finite number of runs using samples derived from probability distributions, “and use[s] hypothesis testing to infer whether the samples provide a statistical evidence for the satisfaction or violation of the specification” (Legay & Delahaye, 2010).

Statistical model checking can be applied to a “larger class of systems than numerical model checking algorithms including black-box systems and infinite state

systems,” and it can be parallelized, “which can help scale to large systems” (Legay & Delahaye, 2010).

Instead of exhaustive search, solving a probabilistic model checking problem means to decide whether a system satisfies a property with a probability greater or equal to a certain threshold (Legay & Delahaye, 2010). For example, say “we want to know whether the probability of an engine controller failing to provide optimal fuel/air ratio is smaller than 0.001; or whether the ignition succeeds within 1ms with probability at least 0.99” (Zuliani, Platzer, & Clarke, 2010). Statistical model checking algorithms suppose that the property can be checked on finite executions of the system (Legay & Delahaye, 2010). It relies “on simulation, which, especially for large, complex systems, is generally easier and faster than a full symbolic study of the system. This can be an important factor for industrial systems designed using efficient simulation tools” (Zuliani, Platzer, & Clarke, 2010).

The disadvantage compared to numerical model checking is that statistical model checking “only provides probabilistic guarantees about the correctness,” and “the sample size grows very large if the model checker’s answer is required to be highly accurate” (Legay & Delahaye, 2010). Moreover, it requires the models to be efficient in terms of computational effort, because the system may need to be simulated many times using different samples.

2.4.2 Runtime Verification

Runtime verification is a “lightweight verification technique complementing verification techniques such as model checking and testing” (Leucker & Schallhart, 2009) (Leucker & Schallhart, 2009); it is intended to overcome the limitations of model checking (Levy, Hassen, & Uribe, 2002) mentioned in Section 2.4.1, at the price of lower confidence in the verification results. Runtime verification is defined in (Leucker & Schallhart, 2009) as: “the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.”

Runtime verification originated from model checking. However, in contrast to model checking, which examines all possible executions of a system to conclude on the correctness of the specified properties, runtime verification examines only a single or a finite subset of executions. Moreover, compared to model checking—which requires the complete model that contains all possible executions—runtime verification is applicable to black-box systems for which certain system properties that are of interest can be observed. Runtime verification is similar to testing, with two main differences: 1) in testing, monitors (or oracles) are defined manually, whereas in runtime verification monitors are generated from some higher level specification (e.g., some variant of linear temporal logic expressions); 2) providing

appropriate system stimuli in order to test the system sufficiently is rarely considered to be the task of the runtime verification domain (Leucker & Schallhart, 2009).

Runtime verification uses *monitors* to determine whether or not properties are violated (Leucker & Schallhart, 2009), (Levy, Hassen, & Uribe, 2002). (Leucker & Schallhart, 2009) define a monitor as “a device that reads a finite trace and yields a certain verdict.” Monitors are used for online monitoring (monitoring of the current run) or offline monitoring (i.e., for analyzing recorded execution traces) (Leucker & Schallhart, 2009).

2.4.3 Testing

“Testing is the process of executing a program with the intent of finding errors” (Myers, Sandler, & Badgett, 2011, p. 11). Testing is a well-known discipline in the software and system engineering fields, and in recent decades many testing strategies have been developed, such as stress testing, fault injection, coverage-based testing, black-box, and white-box testing, or combinations of these.

However, “[i]n general, it is impractical, often impossible, to find all the errors in a program” (Myers, Sandler, & Badgett, 2011, p. 12). Testing cannot guarantee the absence of errors, but it can help to discover their presence. The economics of testing—the balance between the testing effort and project time or resource constraints—depends on the selected testing strategy and the way test cases are designed, as well as the experience of the testers (Myers, Sandler, & Badgett, 2011).

For a model-based approach there exists a standard—UML Testing Profile—which “defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems” (OMG UTP, 2012, p. 1). UTP allows modeling of test context, test configuration (the composite structure of test context), test data used in test procedures, the dynamic aspects of test procedures, and time-quantified definitions of test procedures (OMG UTP, 2012, p. 9).

A *test objective* defines in natural language what should be tested. “A test objective is a reason or purpose for designing and execution a test case. ... The underlying Dependency points from a test case or test context to anything that may represent such a reason or purpose. This includes (but is not restricted to) use cases, comments, or even elements from different profiles, like requirements from [SysML]” (OMG UTP, 2012, p. 25).

A *test case* defines the course of action that leads to the evaluation of the pass/fail criteria: “A test case specifies how a set of test components interact with an SUT to realize a test objective to return a verdict value” (OMG UTP, 2012, p. 22).

2.4.1 Model-Based Testing

Model-Based Testing (MBT) is not to be interpreted as using models when executing tests. The idea behind MBT is the automatic generation of test inputs and expected outputs from a model that represents the intended system behavior. The model must be correct with respect to the requirements imposed. Typically, the main goal of an automatic test case generator is to ensure some kind of coverage—such as code branch, function call, or statement execution coverage, or at the model level, the coverage of state transitions in state machines defined for a class. The derived test cases are then used to test the real application. Such an approach is referred to as *static* in (Artho, et al., 2005). The dynamic approach (Artho, et al., 2005) considers test-input generation as an optimization problem and automatically generates test inputs based on the results obtained in previous executions.

In contrast to runtime verification (see Section 2.4.1.1), model-based testing focuses on generating test cases automatically and not on modeling and checking individual properties.

2.4.1 Simulation

Simulation is defined in (Cellier & Kofman, 2006, p. 8) as a process that “concerns itself with performing experiments on the model to make predictions about how the real system would behave if these very same experiments were performed on it.” A similar definition is given in (IEEE Std 610.12-1990, 1999).

Clearly, simulation is based on models. In the context of this dissertation we will deal with a subclass of models: *mathematical models* (Cellier & Greifeneder, 1991) (Fritzson, 2004, p. 11). Mathematical models are expressed using, for example, algebraic or differential equations. Such models typically capture the dynamic behavior of systems (i.e., time-dependent behavior) and enable numerical experiments (Fritzson, 2004, p. 6). Mathematical models are well suited for capturing and simulating the behavior of complex physical systems (Cellier & Kofman, 2006).

In contrast to testing or runtime verification, simulation focuses on the prediction of real system behavior. Models used for that purpose should be as close as possible to the final system to be built. However, it is often the case that a detailed consideration of physical phenomena for complex systems is difficult to ensure, due to computational effort required for simulating models.

2.5 System Modeling and Simulation Languages

This section briefly introduces the languages that are used in this dissertation for prototyping support for the new method. An exhaustive survey of modeling and

simulation languages, comparison of languages, and motivations for reusing existing languages or developing a new one is left aside here for purposes of brevity. There are several potential languages which, when combined, could serve the purpose. To the best of our knowledge there is no single language that can already do so. The languages that are described below were selected based on a tradeoff in terms of model expressiveness, simulation capability, and domain applicability, and because they are standardized and open. The paradigm and formalisms they are based on, such as differential algebraic equations, state machines, object-orientation, etc., are well known and accepted by practitioners.

The languages are UML (OMG UML, 2011) and Modelica (Modelica, 2013). UML is the most common language for software modeling, and Modelica is popular for the modeling and simulation of physical systems. Both languages combined enable the modeling and simulation of Cyber-Physical Systems (CPS), which are “integrations of computation and physical processes” (Lee, 2008).

For further reading it is recommended to have a basic understanding of UML and Modelica.

2.5.1 UML-Based Languages for Descriptive Modeling⁴

UML (OMG UML, 2011) is a general-purpose graphical modeling language primarily used for modeling and communicating software designs.

UML is a graphical modeling language, not a methodology. UML provides a comprehensive set of diagrams and a meta-model for modeling of object-oriented software structure, behavior, and software deployment. The logical behavior of system components is captured in UML-based languages through a combination of activity diagrams, state machine diagrams, and/or interaction diagrams. UML neither prescribes when to use which diagram nor what steps to follow when modeling. However, there are several methodologies that use UML-based languages (Estefan, 2007) to structure information and improve communication between different disciplines involved.

UML does not fully specify the model execution semantics. Semantic variation points intentionally underspecify semantics: “The objective of a semantic variation point is to enable specialization of that part of UML for a particular situation or domain” (OMG UML, 2011, p. 23). This is different from modeling and simulation languages such as Modelica (Modelica, 2013), which specify the concrete syntax (textual notation) as well as the execution semantics.

⁴ The text of this section is an updated version of the text from (Schamai, Fritzson, Paredis, & Pop, 2009).

Further, UML is designed in such a way that it can be extended by so-called profiles. UML profile includes extensions of the UML meta-model concepts, as well as extensions or modifications of the concrete graphical syntax. The UML profile mechanism is used to adapt the language to a particular domain or purpose.

For systems modeling such a specialized language exists. SysML (OMG SysML, 2012) is a UML profile and a general-purpose systems modeling language that enables engineers to create and manage models of engineered systems using graphical notations. SysML reuses a subset of UML constructs and extends them by adding new modeling elements and two new diagram types. Through these extensions, SysML is capable of representing the specification, analysis, design, verification, and validation of any engineered system.

SysML includes parametrics to support the execution of constraint-based behavior such as continuous-time dynamics. However, the syntax and semantics of such behavioral descriptions in parametrics have been left unspecified to interoperate with other simulation and analysis modeling capabilities.

Textual requirements are still the main vehicle for communicating and agreeing on system specifications in a system development process. SysML provides mechanisms to include textual requirements into models.⁵ This facilitates the traceability of textual requirements to design and test artifacts. Requirements are still validated using inspection and engineering judgment. SysML does not incorporate the means to either formalize requirements or verify them using, for example, simulations.

SysML offers a notation for descriptive modeling. Like UML, SysML models are not executable by default because the execution semantics are not fully specified. Consequently, designs cannot be analyzed in terms of system dynamic behavior and compliance with specified requirements using simulation.

2.5.2 Modelica for Dynamic Behavior Modeling and Simulation⁶

Modelica is an object-oriented equation-based modeling language primarily aimed at physical systems (Modelica Association, 2012), (Fritzson, 2004). The model behavior is based on ordinary and differential algebraic equation (OAE and DAE) systems combined with discrete events, so-called hybrid DAEs. Such models are ideally suited for representing physical behavior and the exchange of energy,

⁵ The SysML requirement concept allows capturing the requirement ID and text.

⁶ The text of this section is an updated version of the text from (Schamai, Fritzson, Paredis, & Pop, 2009).

signals, or other continuous-time or discrete-time interactions between system components.

Modelica models are similar in structure to UML/SysML models, in the sense that Modelica models consist of compositions of sub-models connected by ports that represent energy flow (undirected) or signal flow (directed). The models are acausal, equation-based, and declarative. The Modelica language is defined and maintained by the Modelica Association, which publishes a formal specification but also provides an extensive Modelica Standard Library that includes a broad range of essential models covering domains ranging from analog and digital electrical systems, to mechanical motion and thermal systems, to block diagrams for control. Finally, it is worth noting that there are several efforts within the Modelica community to develop open-source solvers, such as in OpenModelica Project (Open Source Modelica Consortium, 2013).

2.6 Problem Description

The review of the state of the art presented in this chapter shows that there is no approach that is specialized for design verification against natural-language requirements.

MBSE methodologies focus more on description of designs and do not provide means for formal verification of system design against natural-language requirements. Further, many of those use modeling to structure information and not to perform verification.

Verification techniques have other special focus or are limited in terms of model complexity. For example, model checking does not scale to cope with continuous-time models. Runtime verification primarily focuses on software runtime properties checking and addresses challenges in generating efficient code from property monitors in order to minimize their impact while running them in parallel to the software being verified. Testing in general aims to find design errors by stimulating the system through appropriate test cases that include verdicts for determining the pass/fail criteria. The main intention of MBT is to automate the generation of test cases based on a special white-box system model. Simulation is primarily used for predicting the behavior of the real system that is represented by the model.

None of the listed approaches incorporates means for formalizing natural-language requirements for design verification. In conclusion, there is a need for a new approach which is tailored to a model-based verification of designs against requirements. The research questions stated in Section 1.3 have not been answered yet.

Chapter 3

Model-Based Design Verification Method

3.1 Motivation⁷

The ever-increasing complexity of products has a strong impact on time-to-market, cost, and quality. Products are becoming increasingly complex due to rapid technological innovations, especially with the increase in electronics and software—even inside traditionally mechanical products, generally referred to as cyber-physical systems.⁸ This is especially true for complex, high-value-added systems such as aircraft and automobiles, which are characterized by a heterogeneous combination of mechanical and electronic components. The industry confronts the competitive challenge of developing and integrating systems with sufficient maturity at entry into service. The development of model-based systems engineering (MBSE) approach (INCOSE, 2007) has progressively been used to address the limitations of traditional approaches relying on textual descriptions. Building on this general idea of MBSE, this dissertation aims to build a virtual verification environment for modeling cyber-physical systems, requirements, and scenarios, so that system designs can be simulated and verified against the requirements in the early system development stages.

This chapter presents a new method, called virtual Verification of Designs against Requirements, or vVDR. This method enables dynamic system behavior to

⁷ Some text in this section is an updated version of (Schamai, Helle, Fritzson, & Paredis, 2011).

⁸ The focus here is on an integrated modeling and simulation of physical and software components of complex systems. Network technology and communication aspects of cyber-physical systems are not addressed.

be verified against requirements using executable mathematical models.⁹ The method is described independent of the language and tool used to support it, as well as independent of particular types of analyses used for design verification. The discussions below analyze the challenges and derive requirements for the language and tool support.

3.2 Scope¹⁰

The method proposed here assumes that the requirements have been elicited¹¹ as requirement statements in natural language, according to common standards in terms of quality; i.e., for example, according to (Hull, Jackson, & Dick, 2005) the individual requirements should be unique, atomic, feasible, clear, precise, verifiable, etc. Methods to achieve this have been well defined and can be considered well established.

Further, an MBSE approach to system design—i.e., the development of a system design model based on requirements—is not within the scope of this dissertation¹². The method proposed here is neither a design nor a specification method. The main purpose of the proposed method is to enable design verification against natural-language requirements.

3.3 Using Models for Design Verification

Any model is an abstraction of reality. The confidence of any prediction of system behavior depends on the model's computational accuracy, level of detail, correctness of the modeled physical phenomena, etc. It certainly seems challenging to use inaccurate models to predict system behavior so as to determine whether a requirement is satisfied. It is, in particular, counterintuitive and challenging for people that make decisions based on models (Hazelrigg, 2012, p. 183) (Sterman, 2002, p. 525).

Because a model is an abstraction or simplification, any answer gained from verification using models is subject to intrinsic uncertainty and must be critically reviewed by experts (Fritzson, 2004, p. 10).

⁹ In the following, when we speak about models, we mean models that can be unambiguously represented on the computer, are based on mathematics, and often are executable.

¹⁰ Some text in this section is an updated version of (Schamai, Helle, Fritzson, & Paredis, 2011).

¹¹ A description of the various methods for eliciting—i.e., capturing—requirements can be found in (Loucopoulos & Karakostas, 1995) and (Gause & Weinberg, 1989).

¹² The interested reader can find a detailed overview of existing solutions for this approach in (Estefan, 2007) and (Helle, Strobel, Mitschke, Rivière, & Vincent, 2008).

This is particularly true for models representing complex physical systems. Such models require careful consideration of the approximated physical phenomena. Similar considerations apply to scenario models that represent a sequence of actions to be performed to stimulate a system in a certain way, and that may also include approximations (e.g., trigger time, or values that represent triggers from the physical system environment).

The purpose of the term *virtual* in the title *virtual Verification of Designs against Requirements* is to emphasize the fact that we are dealing with models, with all consequences that can be drawn from this fact.

The discussion above contributes to the Research Question 1 (see Section 1.3) by stressing the fact that any model is an approximation of the physical system modeled.

3.4 The Role of Scenarios

An experiment is a process of extracting information from a system by exercising its inputs. To accomplish this, the system must be controllable and observable. (Fritzson, 2004, p. 5)

The separation of model description and experimental description is essential in order to analyze the system in different experiments (Cellier & Greifeneder, 1991, p. 6). An experimental description includes configuration parameters and a scenario that presents the course of actions that serve as stimuli for the system model. When analyzing a physical system, the scenario essentially plays the role of the system user or its operational environment and provides external events or conditions for the system under analysis.

Scenarios are needed in both static (e.g. model checking) and dynamic (e.g. testing or simulation approach) verification techniques. Model checking will involve one non-deterministic part of the model that covers all possible external events or conditions that can be anticipated for the system. Testing or simulation will use many deterministic scenarios that cover the problem state sufficiently.

Scenarios can be as simple as turning the system on or off, or they may be complex scenarios that, for example, include branching conditions based on the system state and cover entire or partial operation procedures. Further, there may be normal operation scenarios that present the intended way of using the system, stress scenarios that will bring the system into critical states, misuse scenarios or scenarios stimulating dysfunctional system behavior—for example, during degraded system modes.

The discussion above contributes to the Research Question 1 (see Section 1.3) regarding required modeling artifacts. It indicates that scenarios should be one of the main concepts in the context a model-based design verification approach.

3.5 Analysis Approach for Design Verification

As pointed out in Section 2.4, there are two ways to manage the task of verifying designed system behavior against requirements. The purpose of the text hereafter is to explain why a simulation-based testing approach for design verification is used, in contrast to formal static analysis methods such as model checking.

3.5.1 Static Analysis: Model Checking

When using formal techniques, design and requirements are modeled using mathematical formalisms. Models are analyzed statically in order to determine whether assertions (e.g. violation of requirements) can be proven or refuted. Static analysis means that no models are executed in order to calculate the predicted behavior over time. In the case of design verification, such an analysis model would include the system design, requirements to be verified, and all possible conditions and events that the system may experience. Given this, formal methods such as model checking can prove¹³ that in any possible situation¹⁴ a particular requirement is satisfied or violated.

However, the applicability and scalability of the numerical model checking approach is limited due to the state explosion problem (Baier & Katoen, 2008, pp. 15, 77 ff). For an adequate representation of actual, complex physical system designs, models will have to involve differential algebraic equations (Fritzson, 2004), (Cellier & Kofman, 2006). Analysis of system behavior over time requires hybrid behavior simulation capabilities (i.e., combining continuous-time, discrete-time, and event-based system simulation). Static analysis and formal methods (e.g. model checking) is limited to discrete variables and are difficult to apply to such kinds of models (Levy, Hassen, & Uribe, 2002). Adequate modeling of complex systems “may easily exceed the amount of available computer memory,” and “[d]espite the development of several very effective methods to combat this problem ..., models of realistic systems may still be too large to fit in memory” (Baier & Katoen, 2008). Moreover, “the logics for which model checking algorithms exist are extensions of classical temporal logics, which are often not the most popular among engineers” (Legay & Delahaye, 2010). For that and other reasons, such formal techniques are not widely used in industrial projects (Woodcock, Larsen, Bicarregui, & Fitzgerald, 2009).

¹³ “It has a sound and mathematical underpinning; it is based on theory of graph algorithms, data structures, and logic” (Baier & Katoen, 2008, p. 15)

¹⁴ Note, the downside of this approach is that a particular scenario cannot be tested without changing the model.

(Leucker & Schallhart, 2009)) point out limitations of formal approaches such as theorem proving or model checking (while explaining when runtime verification should be used instead). Some arguments of (Leucker & Schallhart, 2009) (Leucker & Schallhart, 2009) can also be used in the context here to illustrate why static analysis methods such as model checking may not be the best choice for design verification.

For model checking, the verification verdict refers to a model that is neither the real system nor an adequate representation of the entire system. The models used for model checking typically only reflect certain aspects of interest because, applying these techniques directly to the real implementation is intractable (Leucker & Schallhart, 2009, p. 7). As mentioned in Section 2.4.1, models that represent the dynamics of physical systems are built based on continuous-time equations and are complex in terms of model size. Analyzing such models is also intractable without being trapped by the state explosion problem. It will be necessary to build a dedicated analysis model to be used for model checking which will abstract the system behavior and allow only certain aspects of interest to be analyzed. The abstraction process is always subject to the introduction of errors or the modeling of incorrect behavior due to misinterpretation or lack of deep domain knowledge. Even though this may be suitable for answering a particular analytical question, it will not be sufficient for design verification.

Furthermore, when using libraries of components that are not open,¹⁵ not all information is available statically and can only be obtained when executing (e.g. simulating) the system (Levy, Hassen, & Uribe, 2002). In numerical model checking (see Section 2.4.1.1), verdicts are defined in a way that comes to conclusions based on an analysis of the entire state space and all possible executions. They are not appropriate for online monitoring and are not reusable for real system testing or monitoring the system in operation (Leucker & Schallhart, 2009).

3.5.2 Dynamic Analysis: Simulation-Based Testing

In contrast, simulation focuses on predicting the behavior of the real system as precisely as possible. The main purpose of simulations is to be as close as possible to the real system with respect to its relevant aspects. This makes simulation attractive for design verification. Using appropriate modeling languages, such as Modelica, engineers can describe system behavior at an arbitrary level of accuracy

¹⁵ The use of third-party libraries with no implementation details available is common practice nowadays.

and simulate them. “Simulation is one of the most common techniques for using models to answer questions about systems” (Fritzson, 2004, p. 9).

Testing performs analysis of the final product. As pointed out in Section 2.4.3 and 2.4.1, the main challenge of testing is to stimulate the system in such a way that sufficient confidence can be gained that the system will behave as intended and that as many errors as possible will be discovered. In the same way testing can also be applied to models that adequately represent the system design.

Simulation and testing require deterministic scenarios (see Section 3.4). The scalability issue of model checking (identified in Section 3.5.1 as one of the main obstacles for design verification) is shifted, in the simulation and testing approach, to the issue of selecting the appropriate set of scenarios. Deterministic scenarios are specific projections of the overall problem space: i.e., a particular sequence of events, combination of parameter values, etc., that stimulate the system. Clearly, a subset of deterministic scenarios does not cover the entire problem space. Finding the right selection of appropriate scenarios to verify a system against particular requirements is challenging, because this decision directly impacts the coverage of the problem space and ultimately confidence in conclusions drawn. Moreover, this approach cannot determine if a requirement is satisfied in a design, because it does not test all possible operational conditions and scenarios but only a subset of them (see discussion in Section 3.6).

The limitation of scenario-based testing with respect to coverage of the problem space is a disadvantage compared to static analysis, such as model checking, which offers full coverage of the model.¹⁶ However, the advantage of simulation and testing is the scalability and accuracy of simulation models. Models that represent designs of complex physical systems are complex as well. Modeling and simulation of complex physical system (such as spacecraft, aircraft, cars, or their subsystems) will produce complex models that include differential algebraic equation systems to be solved for continuous-time simulations that predict system behavior. The complexity involved in simulating such kinds of models (Cellier & Kofman, 2006) in a reasonable time will, unfortunately, in most cases exclude techniques other than testing or simulation from practical use in industrial-sized projects.

A simulation-based testing approach has one additional advantage. Testing is an integral part of any system integration or development process. Scenarios that were used for simulations in the design phase can later be reused to test the real parts (e.g. in “hardware in the loop” testing when some parts are real and some are represented by models, or in final system integration and qualification testing) on

¹⁶ It offers full coverage of the model because model checking “verifies a system model, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques, such as testing, are needed to find fabrication faults (for hardware) or coding errors (for software)” (Baier & Katoen, 2008, p. 15).

real test rigs leveraging the work done upfront. In doing so, system design verification using models will reduce the work for product verification.

3.6 Satisfaction vs. Violation of a Requirement

Let us discuss the meaning of an answer to the question of whether a design complies with a requirement. Consider the following possible answers:

1. “Design d_1 satisfies ¹⁷/does not satisfy requirement r_1 .”,
2. “Design d_1 does not violate/violates requirement r_1 ”.

What is the difference? Essentially, *does not satisfy* is equivalent to saying *violated* and vice versa. However, saying that a *design does not violate requirement* does not mean that it *satisfies the requirement*. Why does this matter?

First, any conclusion drawn from verifications using models is based on the confidence of the supervising engineer who believes that requirements were tested sufficiently, i.e., using sufficiently accurate models.

Second, using simulations, any particular execution represents one trace within the entire problem state space. Such an approach does not guarantee that a requirement is tested in any possible situation. Using such an approach we can only conclude whether a requirement is satisfied/not satisfied, or violated/not violated while tested with some particular scenarios. So any of the answers from the above implies that a requirement was tested using a subset of scenarios that covers the problem domain sufficiently. Consequently, a more appropriate formulation of the answer from the above is:

1. “Design d_1 satisfies/does not satisfy the requirement r_1 while tested with scenarios $s_1...s_n$ ”,
2. “Design d_1 does not violate/violates the requirement r_1 while tested with scenarios $s_1...s_n$ ”.

Having determined a violation of requirement in one scenario means that the design violates the requirement.¹⁸ This statement holds for however many scenarios were used because there is at least one point evidence, namely the scenario that detected the violation. However, having found satisfaction of a requirement in one scenario does not mean that the design satisfies the requirement, because not all possible

¹⁷ The term *satisfies* is interchangeable with *fulfills*, *meets* or *conforms to* in this context.

¹⁸ Note that *design does not violate requirement* does not mean *satisfies*, which has a stronger insinuation.

scenarios have been tested and there is no evidence and guarantee of satisfaction (Hazelrigg, 2012, p. 131).

(Dijkstra, 1972) said, “[P]rogram testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.” Dijkstra was talking about computer programs. The statement still applies to simulation models, which are executable computer programs representing system designs. In using simulation for testing or verifying designs against requirements we strive to find design errors in order to prove their presence. In that context, we find an error each time a requirement is violated. However, finding no errors does not imply that the design satisfies the requirement. It only means that no errors (i.e., no violation of the requirement) have been found up to now using a subset of possible scenarios. This is because “it is impossible to test a program sufficiently to guarantee the absence of all errors” (Myers, Sandler, & Badgett, 2011, p. 20). A similar statement is made by (Ball, 1999), noting that “dynamic analysis cannot prove that a program satisfies a particular property, [but] it can detect violations of properties.”

Obviously, the terminology in version 2 of the answer above is more appropriate. A testing activity can only discover requirement violations and cannot guarantee requirement satisfaction.

The discussion above contributes to the Research Question 2 (see Section 1.3) regarding the representation of requirements using models. It points out that we should conclude on requirement violation and not on requirement satisfaction.

3.7 vVDR Method Description¹⁹

The cost of fixing problems increases as the lifecycle of the system under development progresses. Thus, there is strong motivation to detect defects early in the lifecycle from a business perspective. Design verification and testing needs to be applied as early as possible in the lifecycle to minimize the relative cost to repair any problems discovered. This means that testing should be integrated into the system design phase so that the system design can be verified against requirements early on.

To enable automatic verification of a design model against a given set of requirements, the requirements have to be processed by a computer. MBSE typically relies on building models that replace or complement the textual requirements. Links between the model elements and the natural-language requirements are usually kept at the requirements’ granularity level (OMG SysML, 2012), meaning that one or more model elements are linked to one requirement.

¹⁹ The text in this section is partially based on (Schamai, Helle, Fritzson, & Paredis, 2011)

This granularity is good enough for basic traceability and coverage analysis but fails when computer-based interpretation of a requirement's content is necessary.

Research on the automatic translation of natural-language requirements into behavioral models supports the automation of system and acceptance testing, but this has not been widely adopted in industrial practice (Mich, Mariangela, & Pierluigi, 2004). Formal mathematical methods may be used to express requirements, but their application requires a great deal of expertise, and hence they are not very common in industrial practice.

A recent survey came to the conclusion that “in spite of their successes, verification technology and formal methods have not seen widespread adoption as a routine part of systems development practice, except, arguably, in the development of critical systems in certain domains” (Woodcock, Larsen, Bicarregui, & Fitzgerald, 2009). The bottom line is that natural language is still the most common approach to express requirements in practice (Mich, Mariangela, & Pierluigi, 2004). One of the goals here is to provide a solution to the question of how to formalize requirements so that they can be processed and evaluated in simulations in order to detect errors or inconsistencies in a way that is easy to understand and apply.

The sections below provide answer to the Research Question 1 (see Section 1.3) about the characteristics of a method for a model-based design verification approach.

3.7.1 Overview

What kind of artifacts is required for design verification? As pointed out in Section 2.1, the main purpose of design verification is to collect evidence that a given design will satisfy the requirements imposed. Naturally, there will be system designs to be modeled. Furthermore, to determine whether a given system design violates an individual requirement, natural-language requirements will need to be formalized in order to be processible by computers and usable for simulations.

As pointed out in Section 3.4, it is crucial to separate the experimental description from the system model. To analyze designs under different conditions, deterministic scenarios will need to be modeled. This is the minimum set of modeling artifacts that is required for design verification. Let us list these modeling artifacts and briefly²⁰ define their purpose:

- **Requirement Model:** Each requirement is represented by a model that indicates requirement violation at any simulated time instant.

²⁰ A detailed definition is provided in subsequent sections.

- **System Design Model:** A particular design alternative or version is represented by a model to be used for verification against requirements.
- **Scenario Model:** Scenario defines the course of actions to be simulated. It emulates the user or external events that will be experienced by the system.
- **Verification Model:** Requirements, design alternatives, and scenarios can be combined in different ways. Such combinations are called *verification models*. They are used to perform design verification and to report on requirement violations²¹.

Figure 1 shows the basic steps of the new method, the roles involved, and the artifacts created. The nature of this approach is iterative. The sequential presentation that can be interpreted from Figure 1 should merely indicate that typically first the system development process requirements are elicited, then designs are produced, and finally tests are defined. However, in a concurrent engineering environment these activities may overlap and be rerun iteratively throughout development phases.

²¹ See discussion in Section 3.6.

Figure 1: vVDR overview

Figure 2 shows the flow of information. Requirement models are the output of the *formalize requirements* activity and are the input to the *formalize designs* activity. Design and requirement models are the input to the *formalize scenarios* activity that outputs scenario models. All models are the input to the *create verification models* activity, which outputs verification models to be used for analysis in the *execute and create report* step. The report generated and all these models are, finally, the input into the *analyze results* activity (not shown in Figure 2).

Figure 2: Information flow in vVDR

3.7.2 Roles

Each role at the left side of Figure 1 has associated skills and is responsible for creating the modeling artifacts listed on the right side of Figure 1. The definition of roles does not imply that each of them must be performed by a different person. Depending on the size and organization of the project, one person may perform one or more of the listed roles.

3.7.2.1 Requirements Analyst

The requirements analyst acts as the liaison between the business professionals and the customers, on the one hand, and the system design team on the other hand. The requirements analyst is responsible for ensuring the correctness and completeness of the input requirements that are handed down from a business analyst. The objective is to create a set of requirements which ensures that the product fulfills its original intent.

The analyst needs analytical skills to critically evaluate the information gathered from multiple sources, reconcile conflicts, and distinguish solution ideas from requirements. In our scope, the requirements analyst is responsible for translating

the textual requirements into a set of requirement violation monitors. This task includes selecting suitable requirements and formalizing requirements.

3.7.2.2 System Designer

A system designer develops the system design based on the requirements received from the requirements analyst. An engineer in this role creates the static architecture and defines the dynamic behavior of the system. The system designer needs technical and creative skills. In addition, a system designer is responsible for interpreting verification results and implementing changes in the design if necessary.

3.7.2.3 Tester

The tester is responsible for creating scenarios, simulating, generating verification results, and compiling verification reports; “in its broadest sense, testing is any activity that allows defects in the system to be detected or prevented, where a defect is a departure from requirements” (Hull, Jackson, & Dick, 2005). The main responsibility of the system tester therefore is to collect evidence that a particular system design will be compliant with system requirements.

One of the basic principles of software testing suggested in (Myers, Sandler, & Badgett, 2011, p. 17) is that “[a] programmer should avoid attempting to test his or her own program.”. The same applies to system testing. The tester and system designer should preferably not be the same person.

3.7.2.4 Difference to Traditional Role Responsibilities

There is one important difference in the role definitions compared to traditional roles. Traditionally, it is the responsibility of the tester to define test cases and pass/fail criteria (i.e., test verdict). When testing requirements, the pass/fail is used to determine either no violation or violation of the requirement, respectively. This assumes that the tester knows exactly how to interpret the requirements correctly.

This is different in vVDR. Here the requirement analyst is responsible for defining pass/fail criteria for each requirement (see Section 3.7.3). This is motivated by the fact that testers are not responsible for capturing or negotiating requirements. Requirement analysts are in charge of eliciting and writing down the requirements correctly. Consequently, the requirement analyst has more knowledge on how to interpret the requirement correctly, compared to a tester, making the requirement analyst the best person to define when a requirement is violated.²²

²² The violation is expressed independent of concrete designs or scenarios, as pointed out in Section 3.7.3.

3.7.3 Task: Formalize Requirements

There is a notion of “verification requirement” mentioned (but not defined) in (NASA, 2007), (INCOSE, 2006), and (ESA, 1996). Each requirement should be associated with a verification requirement that defines how it is intended to verify that requirement (ESA, 1996).

The approach for formalizing requirements presented hereafter addresses this need and provides an answer to the Research Question 2 (see Section 1.3). However, in contrast to an explicit definition of how a particular requirement should be verified, the purpose of the requirements formalization proposed in vVDR is as follows:

Each individual natural-language requirement is translated into a formal model that can detect requirement violations for any system design alternative or scenario used for verification. Such models are referred to as requirement violation monitors.

3.7.3.1 Requirement Violation Monitors

The idea for formalizing a natural-language requirement into a requirement violation monitor is similar to the monitor concept used in runtime verification (Leucker & Schallhart, 2009) (Levy, Hassen, & Uribe, 2002).²³ Like in runtime verification, the main purpose of the requirement violation monitor is to detect requirement violations without intervening into the analyzed system.²⁴ The detection is indicated by a status²⁵ attribute, which by definition is part of each monitor.

Let us first discuss what values the requirement violation monitor status should be enumerated with. Monitors “should adhere to the two maxims of *impartiality* and *anticipation*.” (Leucker & Schallhart, 2009). Both maxims require a monitor status “to have at least three different truth values: true, false, and inconclusive, but of course more than three truth values might give a more precise assessment of correctness” (Leucker & Schallhart, 2009). The reason is the following: when using or executing (e.g., simulating²⁶) a system, it is correct to conclude that a property does not hold as soon as a violation is detected. However, if no violation has been

²³ A brief introduction to runtime verification and monitor the concept was presented in section 2.4.1.1.

²⁴ (Leucker & Schallhart, 2009) mention application areas where monitors are not only used to monitor behavior but can also be used, for example, for fault detection, identification, and recovery (FDIR); i.e., monitors can intervene on the system. This topic is not addressed in this dissertation.

²⁵ Also referred to as verdict or oracle in the literature.

²⁶ This is opposed to an exhaustive search that analyzes all possible executions; see modeling checking in section 2.4.1.

detected so far, it is not correct to conclude that the property holds, because it may happen that the property was not evaluated at all, or there may be violations at some point later in this particular execution or in others.

In a simulation-based testing approach that uses a finite number of executions based on deterministic scenarios, this means that the requirement violation monitor status should be enumerated with at least the following values²⁷:

- *not evaluated* (default value, provided there is a precondition for the evaluation), which indicates that the requirement was not yet evaluated,
- *violated*, which indicates a violation of the requirement (implying that the requirement was evaluated),
- *not violated*, which indicates that no violation was found (still implying that the requirement was evaluated).

The violation monitor status shall be provided at any simulated time instant and can change between *not evaluated*, *not violated*, and *violated* in any possible way. This way violation monitors can be used for online monitoring (Leucker & Schallhart, 2009). The status *not evaluated* means that the precondition is not met and that the requirement cannot be evaluated under the given conditions.

(Leucker & Schallhart, 2009) propose *four-valued semantics* for a monitor, meaning that it “either (1) satisfies the monitored property, (2) violates the property, (3) will presumably violate the property, or (4) will presumably conform to the property in the future, once the system has stabilized.” This approach seeks to handle situations where properties cannot be evaluated because in the current execution the system has not yet gotten into a state which is required for the evaluation. This approach is not adopted in vVDR. In vVDR this issue is instead resolved by ensuring the use of appropriate scenarios which stimulate the system in such a way that any requirement of interest will be evaluated (see Section 3.7.5).

Let us specify a requirement violation monitor as follows:

Definition 1. Requirement Violation Monitor: A requirement violation monitor is a model that, at any simulated time instant, indicates whether the requirement it represents is evaluated, violated, or not violated.

3.7.3.2 From Natural-Language Statement to Violation Monitor

The task of formalizing requirements into violation monitors—i.e., the expression of the violation condition(s)—can be accomplished in many different ways using different formalisms. For example, in runtime verification, monitors are expressed

²⁷ This enumeration is referred to as “three-valued semantics” in (Leucker & Schallhart, 2009) with the literals “inconclusive”, “false” and “true” respectively.

in some higher level specifications (e.g. LTL, RV-LTL, TLTL, MITL, etc.) which are then used to generate efficient code for the actual monitors (Leucker & Schallhart, 2009).²⁸ In testing to verify software written in a particular programming language, monitors may be expressed in the very same programming language. Using UML or Modelica models, monitors may be expressed using state machines, equations, or functions. In addition, templates or specialized library functions may be helpful in reducing manual effort and modeling errors.

The choice for using a particular formalism will depend on the modeler's preferences, the skills of the audience (reader of the model or code), project policies, or the list of formalisms supported by the analytical tools that will be used. However, independently of the choice of formalism, the task of converting the natural-language statement into formal language will require correct interpretation and understanding of the requirement statement and the ability to translate that meaning into a model that expresses it identically.

Let us investigate if there is a general systematic way to derive monitors from natural-language requirement statements. Consider the following method:

1. *Read the requirement statement*
2. *Identify properties that can be quantified either by explicit numbers or by logical conditions*
3. *Identify preconditions (if any) that must be satisfied before the requirement can be evaluated*
4. *Express when the requirement is violated and when not*

Consider the following natural-language requirement (Hull, Jackson, & Dick, 2005) imposed on a communications system that must be able to handle a certain number of calls:

Req. 001 *“The communications system shall sustain telephone contact with not less than 10 callers in the absence of external power.”*

What are quantifiable properties that are mentioned in this statement? We can immediately identify that this statement specifies the minimum number of 10 callers to be sustained. Further, there is an expression “*while in the absence of external power,*” which can be quantified as a logical expression, i.e., with either *true* or

²⁸ Generation of efficient code from monitors is one of the main technical challenges in runtime verification (Leucker & Schallhart, 2009).

false, saying that the external power is absent or not, respectively. Let us write down the identified properties using Modelica syntax²⁹:

```
constant Integer minNumberOfCallersToSustain = 10;
input Boolean isExternalPowerAbsent;
```

The property `minNumberOfCallersToSustain` cannot be a real number (i.e., a continuous variable or a fraction), but only positive natural numbers. Consequently, its type must be `Integer`. Moreover, in order to ensure that only positive numbers are allowed, an assertion is added as follows:

```
assert(minNumberOfCallersToSustain >=0, "Number of callers
      must be a positive Integer number.",
      level=AssertionLevel.error
    );
```

The property `isExternalPowerAbsent` is a precondition for the evaluation, because as long as there is external power this requirement does not specify how the system should behave. As long as there is external power—i.e., as long as `isExternalPowerAbsent==false`—the violation monitor shall report *not evaluated*. This property is also an input to be provided by other model, i.e., either by the system design or scenario model.

Further, let us consider the part “*sustain telephone contact with not less than 10 callers.*” Clearly, we are concerned with the number of accepted or sustained callers in the system. However, what does *sustain* mean in this requirement?

- Is this requirement violated when there are no callers on the line and external power is absent?³⁰
- Or does it imply that there should be more than 10 callers on the line before the evaluation of this requirement can start?

It is difficult for a person to interpret this requirement correctly if they did not participate to the elicitation process that identified it. However, the person who is in contact with the stakeholder (e.g. consumer or certification authority) either knows exactly how to interpret it or can answer this question by contacting the stakeholder. This person acts in the role of requirement analyst in vVDR and is responsible for resolving such issues of ambiguity, incompleteness, or incorrectness.

Let us assume that the correct interpretation is that there should be more than 10 callers on the line before this requirement can be evaluated. For example, if there are 100 on the line and external power is cut off this requirement specifies that the

²⁹ The term *property* is adopted from UML. In UML, properties are class attributes. It should not be confused with *properties* in the context of model checking or runtime verification, which instead corresponds to parts or entire monitors.

³⁰ This interpretation is unlikely to be correct.

system shall “*sustain telephone contact with not less than 10 callers.*” A more explicit version of the statement would be:

Req. 001: *With more than 10 callers on the line, the communications system shall sustain telephone contact with not less than 10 callers while in the absence of external power.*³¹

Let us formalize the number of callers in line and the number of callers with whom telephone contact is sustained as follows:

```
input Integer numberOfCallersInLine;
input Integer numberOfSustainedContacts;
```

These properties are also inputs: i.e., the actual values are to be provided by the system design or scenario model for the evaluation. Further, the values can only be positive natural numbers, so we add to the assertion above to ensure the validity of this monitor model.

```
assert( minNumberOfCallersToSustain >=0
        and numberOfCallersInLine>=0
        and numberOfSustainedContacts>=0,
        "Number of callers must be a positive natural number.",
        level=AssertionLevel.error
    );
```

Now we are ready to express when this requirement is evaluated, violated, or not violated. For the monitor status we explicitly introduce the attribute *status*, which can be set to³²:

- 0 means not evaluated,
- 1 means not violated,
- 2 means violated

The resulting code for the requirement violation monitor model is:

```
model SustainedCalls "Req. ID 001: With more than 10 callers on
the line, the communications system shall sustain telephone
contact with not less than 10 callers while in the absence of
external power."

    constant Integer minNumberOfCallersToSustain = 10;
    input Boolean isExternalPowerAbsent;
```

³¹ Note that the requirement still does not specify which of the 100 callers should stay on the line, but only that the number of the sustained calls shall be not less than 10. This may raise a further question “for which callers on the line should contact be sustained?” which is to be resolved by the requirement analyst. For the time being, we will assume that this requirement is not concerned about this issue.

³² See Section 3.7.3.1.

```

input Integer numberOfCallersInLine;
input Integer numberOfSustainedContacts;

Integer status (start=0, fixed=true) "Monitor status";

equation

if isExternalPowerAbsent and numberOfCallersInLine >
minNumberOfCallersToSustain then
  if numberOfSustainedContacts < minNumberOfCallersToSustain
  then
    status = 2 "violated";
  else
    status = 1 "not violated";
  end if;
else
  status = 0 "not evaluated";
end if;

assert( minNumberOfCallersToSustain >=0
  and numberOfCallersInLine>=0
  and numberOfSustainedContacts>=0,
  "Number of callers must be a positive natural number. ",
  level=AssertionLevel.error
);
end SustainedCalls;

```

Note that neither a particular system design nor concrete scenarios were discussed and needed in the process of formalizing this requirement. We merely formalized the content of the requirement statement and resolved interpretation issues. The resulting monitor can be used to verify any system design alternative using any appropriate scenario. Moreover, the same requirement can be tested in multiple scenarios in order to increase confidence that the system being designed will comply with the requirement³³.

Furthermore, note that models that represent requirement statements—i.e., those that formally represent what is stated in natural language—are, in contrast to models that represent physical systems, neither simplifications nor abstractions. They are not built based on known physical laws which are simplified or approximated by the model. They are based on agreements between parties involved or human logical reasoning. A model in this setting is an alternative representation of the natural-language requirement.

For a simulation, all that is left to do is to connect the monitor to the system and then design and scenario models in order to provide the data required for the

³³ To determine whether a design does not comply (i.e., violates a requirement), a finding of one scenario is sufficient.

evaluation. In the requirement above, the following properties will need to be bound to the design or requirement model properties:

```
input Boolean isExternalPowerAbsent;
input Integer numberOfCallersInLine;
input Integer numberOfSustainedContacts;
```

Binding means that during analysis at any simulated time instant these properties of the requirement violation monitor will have exactly the same value as the properties³⁴ they are mapped to within the system design or scenario model. The concrete binding of the properties will depend on a concrete system design alternative or scenario, as well as the structure of the models representing them.

Consider another example of a requirement imposed on a system that will maintain a certain level of liquid in a tank by actuating the outflow valve depending on the input flow.

Req. 002: *The level of liquid in the tank shall not exceed 80% of the tank height.*

The quantifiable properties are the current level of liquid in the tank, the tank height, and the maximum value for the level of liquid.

```
input Real levelInTank;
constant Integer maxLevelHeightPerc = 0.8;
input Real tankHeight;
```

The definition of the violation condition is straightforward: This requirement is violated if the level in tank is greater than `maxLevelHeightPerc * tankHeight`. There is no precondition; this requirement is evaluated at any simulated time. The following violation monitor model results:

```
model ReqLevelOfLiquidInTank "Req. ID 002: The level of liquid
in tank shall not exceed 80% of the tank height."
```

```
input Real levelInTank;
constant Integer maxLevelHeightPerc = 0.8;
input Real tankHeight;
```

```
Integer status (start=0, fixed=true) "Monitor status";
```

equation

```
if levelInTank > (maxLevelHeightPerc * tankHeight) then
    status = 2 "violated";
else
    status = 1 "not violated";
end if;
```

³⁴ If there will be no 1:1 mapping of requirement input properties to design model properties, the binding will refer to an expression to be computed during analysis.

```

    assert(levelInTank >=0
           and maxLevelHeightPerc >=0
           and tankHeight >=0,
           "Level and height values must be positive.",
           level=AssertionLevel.error
    );
end ReqLevelOfLiquidInTank;

```

Again, no information about the actual system design, scenarios that will be used for testing, or implementations of models are required to formalize this requirement. The values for `levelInTank` and `tankHeight` will need to be provided by the system model to verify a particular design alternative.

3.7.3.3 Testing of Violation Monitor Models

It is essential to test the requirement violation monitor models independent of the designs they will be used to verify in order to ensure the correctness and validity of the monitors being modeled.

Consider a simple test for the **Req. 001** from Section 3.7.3.2. The following model includes the **Req. 001** violation monitor and code to stimulate it. The simulation results are shown in Figure 3. Note that in the beginning the requirement violation monitor status is *not evaluated*. Then, after second 1³⁵ the external power is cut off and, depending on the number of sustained calls, the status changes between *not violated* and *violated*. After second 4 external power becomes available again. Thus, the violation monitor status is *not evaluated*, meaning that the precondition for evaluation is not met and that this requirement cannot be evaluated under such conditions.

```

model Test_1_Req1
  SustainedCalls req1_1 (
    numberOfCallersInLine = numberOfCallersInLine,
    numberOfSustainedContacts = numberOfSustainedContacts,
    isExternalPowerAbsent = isExternalPowerAbsent);

  Integer numberOfCallersInLine = 20;
  Integer numberOfSustainedContacts;
  Boolean isExternalPowerAbsent;

algorithm
  if time > 4 then
    isExternalPowerAbsent := false;
  elseif time > 3 then
    numberOfSustainedContacts := 5;
  elseif time > 2 then

```

³⁵ For simplicity, let us consider the simulated time units to be seconds.

```

    numberOfSustainedContacts := 10;
    elseif time > 1 then
        isExternalPowerAbsent := true;
    else
        numberOfSustainedContacts := numberOfCallersInLine;
    end if;
end Test_1_Req1;

```

Figure 3: Testing a requirement violation monitor^{36 37}

3.7.3.4 Added Value

The formalization of requirements supports the identification and resolution of ambiguities, incompleteness, and incorrectness of natural-language statements. Any issues identified in this step must be resolved with the requirement stakeholders, which may lead to an update of the agreed-on textual requirement statements.

³⁶ Recall that the meaning of the monitor status 0 means *not evaluated*, 1 means *not violated*, and 2 means *violated*.

³⁷ The simulation plots are created using the Dymola tool (Dassault Systèmes Dymola, 2013).

This approach makes requirement testable: i.e., it translates natural-language requirements into a form that can be processed by computers and enables analysis and automation. The proposed formalization approach does not require any specific knowledge of concrete system designs or scenarios. Formalization is done solely based on the content of the natural-language requirement and can be done before concrete designs or scenarios are in place. Later, requirement violation monitors can be used to verify different designs and different scenarios.

This activity contributes to the system design modeling by clarifying what data must be provided by system design alternative models to enable their model-based verification against particular requirements. It supports system designers in creating models that can be verified using the vVDR method.

A particular requirement can be verified using the vVDR method if there exists a design model which can provide all input information that is necessary for the evaluation of the requirement violation monitor.

Consider the following requirement:

“The armrest of the seat shall not have sharp edges.”

Can this requirement be evaluated using models, and how can we know if this requirement can be verified using the vVDR method? The answer is straightforward: We first formalize this requirement into a violation monitor in order to obtain the input information required for evaluation. Then we find out if there exists, or will be, a system design model that can provide this information with adequate precision. The only quantifiable property mentioned in this requirement and the violation condition is:

```
input Boolean isSeatArmRestSharp;
...
status = if isSeatArmRestSharp then 2 else 1 "1 means not
violated, 2 means violated";
```

Clearly, this requirement will have to be evaluated for each armrest of each seat. The information on whether a particular armrest is sharp will need to be provided by the system design model. If there is a system design model that can provide this information—i.e., a model that includes a detailed geometrical shape of armrests and the definition what *sharp* means in that context—then we can use the vVDR method to verify the system design against this requirement. If there is no such model, then the vVDR method will not help.³⁸

³⁸ However, it may still help in determining whether requirements are complete and unambiguous, as shown in the first example in section 3.7.3.2.

3.7.3.5 Requirements Preselection

Before formalization, requirements may be preselected. From the set of agreed-on specified requirements, from the very start the requirements analyst may want to preselect only requirements that are good candidates to be verified using the vVDR method. This step may require consultation with the system designer. For example, consider the following requirements taken from (Schamai, Helle, Fritzson, & Paredis, 2011):

- *“The ATP system shall be located on board the train.”*
- *“The ATP system shall consist of a central controller and five boundary subsystems that manage the sensors, speedometer, brakes, alarm and a reset mechanism.”*
- *“The sensors shall be attached to the side of the train and shall read information from approaching trackside signals: i.e., they detect what the signal is signaling to the train driver.”*

Clearly, there is no need for a dynamic analysis of system behavior to verify a particular design alternative against these requirements. Design inspection would be sufficient.

By contrast, the following requirement is a good candidate for the creation and use of violation monitors in detailed system dynamics analyses:

- *“If at any time the controller calculates a ‘caution’ signal, it shall, within 0.5 seconds, enable the alarm in the driver cabin.”*

3.7.3.6 vVDR Language Support

To ease the adoption of the vVDR method by engineers, it is crucial that we simplify or automate complex and/or repetitive tasks. Sections like this one summarizes requirements³⁹ for a language or tool that will be implemented to support the vVDR method.⁴⁰

The listed requirements are derived from the discussion in sections above. Requirements should state what is required and not how to solve it (Hull, Jackson, & Dick, 2005). However, this should not be confused with the fact that any requirement is a solution of a higher problem.

³⁹ These are requirements that are imposed in the language and tool that will support the vVDR method. They should not be confused with examples of system requirements discussed in other sections.

⁴⁰ Note that terms such as *vVDR language*, *vVDR tool* and *vVDR artifact* (referring to a model, model element, or generated document) are referred to simply as *language*, *tool*, *artifact*, etc.

ID ⁴¹	Text
LR 1	The language shall include the concept <i>requirement</i> .
LR 2	The <i>requirement</i> concept shall include the properties <i>id</i> and <i>text</i> for capturing the respective data from the natural-language requirement it represents.
LR 3	For each individual <i>requirement</i> it shall be possible to define a <i>violation monitor</i> that indicates the requirement evaluation status at any simulated time instant during analysis.
LR 4	<p>To indicate evaluation status, a violation monitor model shall have the property <i>status</i>, which can be enumerated with the following values at a minimum:</p> <ul style="list-style-type: none"> - <i>not evaluated</i> (default value, provided there is a precondition for the evaluation), to indicate that the requirement was not yet evaluated, - <i>violated</i>, to indicate a violation of the requirement (implying that the requirement was evaluated), - <i>not violated</i>, to indicate that no violation was found (also implying that the requirement was evaluated).

3.7.4 Task: Formalize Design

System design and/or modeling methodologies, languages or tools are not within the primary scope of this dissertation. The system designer's task in this method step is to create a design model for each design alternative using an appropriate language, which allows an accurate description of the system behavior following physical laws. "System designs are system models. When modelers describe some part of reality as a 'real' system, it means that their system model 'adequately' represents the reality" (Estefan, 2007, p. 20).

While developing the models, the system designer will consider the requirement violation monitors and the data they need for evaluation. This information will help the system designer to decide which requirement should be verified using the design models created, and to develop the design model accordingly.

Models created in this step represent designs: i.e., normal behavior, degraded mode behavior, failure case behavior, etc. It is crucial to represent the physical system design at an adequate level of detail in order predict system behavior as

⁴¹ LR stands for language requirement.

closely as possible to the real system and to increase the confidence in conclusions drawn from model-based verification activities.

From the vVDR perspective, there is no need to explicitly mark models to be system design models. Any model can be considered part of the system design under verification. It may be the overall system, its subsystem, or the leaf components.

Design models include anything that represents the design ideas but should not include anything else (e.g. artifacts for instrumenting the model for any particular analysis purpose). To be useful for model-based verification, a system design model should expose all components from which data will be required to evaluate requirements.

Further, the final model to be used for verification should contain the system design model *and* the system environment. System environment will include models that represent the behavior⁴² of the physical environment, other systems that will interact with the system under development, and possible models for human actors. Such a composite structure is called a *test configuration* in UTP (OMG UTP, 2012, p. 60). Further, the model shall enable stimulating the design via scenarios (see Section 3.7.5), which will orchestrate a particular analysis run.

3.7.5 Task: Formalize Scenarios

Creating test cases is an innovative process that relies on testers' knowledge of the system, experience with similar systems, and ability to be unbiased with regards to the expected results. For this reason, typically, testers are not the same persons as the persons who design the system to be tested (Myers, Sandler, & Badgett, 2011, p. 17).

Traditionally, a test case consists of some behavior (e.g., a sequential, conditional course of actions) and some configuration setting (e.g. values for parameters that can be varied). A test case also includes the verdict—i.e., the pass/fail criteria—that is used to reach a conclusion on whether the system under test complies with the requirement the test case is testing (OMG UTP, 2012, p. 22).

Typically, a tester defines the verdict by interpreting the requirement being tested in a given test case. This approach is subject to misinterpretation and is unneeded in the vVDR approach. In the vVDR method, requirement violation monitors take over the task of concluding whether a requirement has been violated. These monitors were created by a requirement analyst, the person who knows best when a particular requirement can be said to be violated or not. Moreover, including the verdict for exactly one requirement prevents the same test case from being

⁴² Or parts of behavior that are relevant to the operation of the system under development.

appropriate for testing multiple requirements. Consider the requirement imposed on the aircraft fuel system:

“When refueling the aircraft, the fuel system shall ensure that the fuel is distributed uniformly within all tanks.”

We can think of a number of possible parameterized or completely different test cases for testing this requirement. For example, we could define several test cases, each starting with a different level of fuel in tanks and stopping when certain level of fuel is reached. Some test cases may also take into account fuel properties and environmental conditions, as well as operation procedures to be followed. Each of the test cases would include a test verdict with pass/fail criteria for the requirement above.

Some of the above-mentioned test cases could be used for testing other requirements as well: for example, a requirement such as:

“It shall be possible to fill an empty fuel system within 10 min.”

However, the verdict is not applicable anymore because it evaluates only the first requirement. Removing a verdict from test cases will enable their reuse for verification against several requirements.

Furthermore, a test case should be as free as possible from fixed values for system configuration parameters that are specific to particular designs. Clearly, it will not be possible to define test cases completely independent from particular designs. However, a tester should strive to define test cases that can be reused to the greatest extent possible without the need for modifications. This way, the same test case can be used to verify different design alternatives against a number of requirements.

Let us refer to the part of a test case which only specifies the behavior (i.e., the course of actions for setting of variables or producing events in a causal fashion) as *verification scenario*⁴³. A scenario can be arbitrarily complex, including branching or conditional activations of certain scenario parts depending on the state of the tested system, and it will have outputs that are to be used for stimulating the system.

Definition 2. Verification Scenario: A verification scenario is a model that specifies a course of actions to simulate the system under test in such a way that one or more requirement violation monitors can be evaluated.

The purpose of creating and formalizing scenarios in the context of design verification is to detect violation of requirements. Clearly, a tester will have to study

⁴³ In OMG UTP (2012, 4) the term *test behavior* is used to refer to the “[d]ynamic aspects of a test case or parts of the test components involved in the test case.”

requirements in order to make sure that the created scenarios will cover the requirement tests sufficiently. However, requirements are not the only inputs to this process. Naturally, some basic knowledge about the system, its potential users, the physical environment, and the system it will operate in, is important to define appropriate sets and kinds of scenarios.

3.7.5.1 Relations between Scenarios and Requirements

The tester should strive to create scenarios in such a way that one scenario can be used to evaluate multiple requirements. This will reduce the number of tests and lend more confidence in verification results because one requirement will be tested in different situations (i.e., scenarios). In our setting, this means that one scenario will be able to bring several requirement violation monitors out of the *not evaluated* state. Consequently, it will be necessary to explicitly define what scenarios can be used to verify which requirements, and vice versa—which scenarios should not be used to verify what requirements. For that purpose let us introduce two types of unidirectional relations pointing from a scenario to a requirement:

- *use to verify* (indicates that the scenario is appropriate for testing the requirement)
- *do not use to verify* (indicates that the scenario is not appropriate for testing the requirement)

3.7.5.2 Manually vs. Automatically Generated Scenarios

Manual test case generation becomes an increasingly tedious task for real-life system specifications with hundreds or thousands of requirements. Model-based testing provides methods for automated test case generation (see Section 2.4.3).

vVDR does not prescribe any particular approach. Scenarios⁴⁴ can be generated automatically or created manually. Using Model-Based Testing it is possible to generate scenarios that cover the system input domain exhaustively. The downside of this approach is that it is difficult for the generation algorithm to know that these are *good* scenarios that cover requirements sufficiently. As already pointed out above, creating scenarios is a highly creative and innovative process, and in many situations the involvement of testing experts is indispensable despite any kind of possible automation. For example:

- Some scenarios, such as intended system usage scenarios, must be tested explicitly. It is difficult or pointless to use an automated approach to generate such scenarios.

⁴⁴ As mentioned in above, only the part of the test case that specifies the course of actions and its associated variables are of interest for scenarios; verdicts are unneeded.

- Reduction of the number of scenarios becomes crucial for tests that cannot be executed in an automated fashion⁴⁵ or for tests which require long execution times. To identify the fewest possible good and appropriate scenarios in an automated fashion may be difficult.

3.7.5.3 Testing of Scenario Models

It is essential to test the scenario models to ensure the correctness and validity of requirement violation monitors.

3.7.5.4 Added Value

The added value of this task is that scenario models become reusable in the sense that:

- One scenario can be used to evaluate multiple requirements,
- To some extent, one scenario can be reused to verify several different design alternatives.

Treating scenarios as self-contained reusable models will enable automation and reduction of model maintenance effort.

3.7.5.5 vVDR Language Support

The following table captures requirements derived from the discussion in the above sections.

ID	Text
LR 5	The language shall include the concept <i>verification scenario</i> .
LR 6	The language shall allow the definition of relations between scenarios and requirements to indicate which scenarios <i>can</i> be used for evaluating which requirements.
LR 7	The language shall allow the definition of relations between scenarios and requirements to indicate which scenarios <i>should not</i> be used for evaluating which requirements.

3.7.6 Additional Models

Let us refer to any kind of model which is neither a requirement violation monitor, nor a scenario, nor a system design model, as an *additional computation model* or

⁴⁵ For example, human interaction may be required, or, for safety reasons, some tests may need to be executed manually on real physical system test rigs.

simply *additional model*. Additional models may be useful or even necessary for different kinds of computation (e.g., data conversion) not included in requirement, design, or scenario models but needed by those.

For example, assume we are modeling a particular satellite design. This design alternative does not contain any component that continuously calculates the overall center of gravity (CoG) of the satellite while the satellite acquires an orbit. CoG depends on the satellite hardware shape, mass, and position, which do not change over time. However, the level of propellant in the satellite changes over time when the propellant is consumed to perform orbit acquisition or maneuvers. Consuming propellant changes the overall CoG because the mass of propellant in tanks decreases. Further, assume there are requirements that need the current CoG as input. Since this data cannot be provided by the design model, an additional computation model would be created. This additional model will, in turn, require some data from the design model or scenario, such as:

- Propellant starting mass,
- Number of thrusters and propellant mass flow rate per thruster,
- Mass and positions and CoG of hardware components,
- Etc.

Based on these inputs, the additional computation model will then output the current overall CoG of the satellite at any simulated time instant. Each requirement that requires this information can refer to this additional model in order to ensure that this additional model will also become a part of the generated verification model⁴⁶. If this calculation is needed for any consideration of a particular design, then one relation between the design and calculation model will be sufficient.

Another example of using explicit dependencies between models is the combination of violation monitors. “In practice, monitors can share and exchange other information” (Levy, Hassen, & Uribe, 2002, p. 6). “For instance, the correct behavior of M1 may depend on assumptions that are checked by M2” (Levy, Hassen, & Uribe, 2002, p. 7). “Given monitors M1 and M2, monitor M3 relates their activity by including among its observable variables some of the auxiliary variables of M1 and M2. This allows checking properties that are stronger than those checked by M1 and M2 alone” (Levy, Hassen, & Uribe, 2002, p. 7).

In the context of design verification, this means that the violation detection of a requirement r_1 may also depend on the violation status of requirements r_2 and r_3 . Consequently, r_1 would explicitly refer to r_2 and r_3 by requiring a relation in order to ensure that anytime r_1 is instantiated, r_2 and r_3 will be instantiated too. This way

⁴⁶ Verification models are introduced in Section 3.7.7.

requirement violation monitors for r_2 and r_3 will become additional models from the perspective of r_1 .

Yet another example when additional models may be useful is when different system configurations—i.e., sets of concrete values for system parameters—are needed for different scenarios. Assume scenario s_1 requires configuration c_1 and scenario s_2 requires configuration c_2 ; then the configurations would be captured by models c_1 and c_2 and the scenario models s_1 and s_2 would reference them. Note that if one configuration is used by *all* scenarios then there is a modeling alternative. Instead of using additional models, mediators (configuration mediators, see Section 4.2.3.1) can be used to define configurations. A particular group of configuration mediators can then be loaded to generate verification models in a verification session and will provide the design model with a particular set of configuration parameter values. This is a more efficient approach, because in this way no additional models will be included and simulated as part of the verification models generated.

In order to express that a particular additional model is needed by some other model, we shall create the following explicit⁴⁷ relations:

- *required for* (pointing from the additional model to the model that requires data from it)
- *or requires* (the opposite direction).

Let us define such dependencies between models.

Definition 3. Instantiation Dependency: An instantiation dependency is a relation between two models which indicates that whenever the dependency source model is instantiated, the dependency target model will be instantiated as well, and both models are bound to each other.

If the same additional model is needed by multiple other models (e.g., the additional model is referenced by the design model as well as by multiple requirement violation monitors), the additional model should be instantiated only once in order to avoid redundant calculation, to keep the verification model concise and keep the computational effort at a minimum.

The following table captures requirements derived from the discussion in the above sections.

ID	Text
LR 8	The language shall support the definition of <i>instantiation</i>

⁴⁷ *Explicit* means that the relations should be stored by referencing both elements explicitly.

	<i>dependencies</i> between models to indicate that if the relation <i>source</i> model is instantiated the relation <i>target</i> shall be instantiated too. The relation <i>target</i> model is referred to as an <i>additional model</i> .
LR 9	If the same <i>additional model</i> is referenced by several models in a combination, the additional model shall be instantiated only once.

3.7.7 Task: Create Verification Models⁴⁸

So far we discussed the basic set of artifacts required for design verification, namely requirement violation monitors, system design models, and scenario models.

Each of these models is needed in order to create a *verification model*. In a scenario-based approach, a verification model will comprise one system design alternative that is to be verified against a set of requirements by running one verification scenario as illustrated in Figure 4. The tester is responsible for this task in vVDR, which will probably raise some questions:

- Given a design model that is to be verified against requirements, how can we know which requirements can be evaluated: i.e., which requirements are already addressed in the given design?
- Which scenarios can be used for stimulating the given design?
- How can we know which scenarios can be used to evaluate which requirements? We discussed explicit relations between requirements and scenarios in Section 3.7.5.1. However, how can we know if these relations cover all appropriate combinations of a scenario and a set of requirements? Can we discover these relations by any means?

⁴⁸ This section uses an updated version of sections from (Schamai, Fritzson, Paredis, & Helle, 2012).

Figure 4: Concept of verification models

Further, when combined into a verification model, some of the components will require data from others and may in turn provide data for other components. For example, each requirement violation monitor needs data from the design or scenario models. The verification scenario provides stimuli for the design model and might require feedback in order to complete or branch parts of the scenario depending on the state of the tested design.

Even though models will potentially depend on other models when combined in a verification model, they may be created without any a priori knowledge of such relations. For example, the formalized requirement violation monitors will most likely be created before the system design alternative models are in place. In turn, design models will most likely be created before or in parallel with the verification scenarios.

To bind all models correctly is not a trivial task. This task requires knowledge about each model. Obviously, for one person—for example, a tester—who did not create all the models, this will be rather difficult to achieve. There must be a better way. Chapter 4 discusses a proposal to improve this situation. For now, let us define what a verification model is and how it can be classified.

Definition 4. Verification Model: A verification model is a model which represents a combination of one system design which is to be verified against a set of requirements by running one scenario.

Definition 5. Design Verification Run: A design verification run is any type of dynamic analysis (e.g., simulation) for the purpose of evaluation of included monitors by predicting the system behavior over a period of simulated time while executing a particular scenario.

Definition 6. Optimal Verification Model: A verification model is said to be *optimal* if the selected scenario stimulates the design in such a way that all included requirement violation monitors are evaluated⁴⁹ in a design verification run.

Definition 7. Suboptimal Verification Model: A verification model is said to be *suboptimal* if the scenario stimulates the design in such a way that not all included requirement violation monitors are evaluated in a design verification run.

Clearly, we should strive to create optimal verification models in order to reduce computational effort and to evaluate requirements using only appropriate scenarios. However, independent of whether a verification model is optimal or suboptimal, models must be valid with regards to the intended dynamic system analysis (e.g. simulation).

3.7.7.1 Added Value

The added value of the verification model concept is that the main artifacts (i.e., requirement violation monitor, system design, and scenario models) are free of any instrumentation required for a particular verification session. Moreover, the verification models enable reproduction of the verification results. Also, verification models are the only artifacts that explicitly provide the information on which requirements were really evaluated in simulations using which scenarios.

3.7.7.2 vVDR Language Support

The following table captures requirements derived from the discussion in the above sections.

ID	Text
LR 10	The language shall include the concept <i>verification model</i> .
LR 11	A <i>verification model</i> shall consist of a combination of one system design model (any model representing the design of the entire system or a subsystem), one verification scenario model and a set of requirement violation monitor models.

⁴⁹ *Evaluated* means that during analysis the requirement violation monitor changed its state from *not evaluated* to *violated* or *not violated*.

3.7.8 Task: Analyze Verification Models and Create Report

After having created the verification models, the system tester can run design verification analysis, collect results, and compile a verification session report that includes relevant information presented in a suitable manner.

Definition 8. Design Verification Session: A design verification session is an activity that performs a number of design verification runs and produces a verification session report.

Definition 9. Design Verification Session Report: A design verification session report is a document which includes the following information at a minimum:

- Time and date when the verification session was performed
- The list of violated requirements with references to verification model details.
- The list of not-violated requirements with references to verification model details.
- The list of not-evaluated requirements with references to verification models details.
- The list of invalid/incorrect verification models that could not be analyzed.
- Each verification model detail generated includes reference to the system model, verification scenario, as well as the information on how those models are connected to each other.

3.7.9 Task: Analyze Verification Results

The verification session report is the basis for discussions among the roles who are involved and may lead to iterations and modification of artifacts based on the newly generated knowledge.

The reported requirement violations may result from errors in any model used: i.e., errors in the violation monitor models, scenarios, or system design that was verified. This is why all roles should participate in the analysis of verification results.

However, the goal of this method is to determine whether the system design complies with the requirements. The system designer is responsible for the correctness of the produced designs, and for that reason, the system designer should initiate and lead this activity.

Furthermore, later on the reports generated can be reused as a reference for comparing results from product verification, i.e., the testing of real physical system items on test benches or in operation.

3.8 What Can be Automated?

Let us investigate which steps or tasks of the vVDR method can be automated. The task of formalizing requirements, i.e., the transition from textual statement to a formalized violation monitor, is unlikely to be a good candidate for automation because it requires correct interpretation and understanding of the natural-language requirement (possibly including all implicit information that is not captured in the statement). The correct interpretation of requirements should be ensured by the same person who is involved in eliciting and negotiating requirements, and who knows the captured and implicit information best.

Further, creating and formalizing designs is also difficult to automate, because this task requires creative human engineering capabilities, a great deal of experience, and a deep understanding of the domains involved, the underlying assumptions and simplifications, etc.

Similar arguments apply to the process of creating scenarios for testing a system. A good deal of attention is paid nowadays to automatic test case or test input generation approaches. Despite the possible degree of automation, to some extent the work of creating scenarios will still be a manual task performed by testing experts, as discussed in Section 3.7.5.2.

However, the creation of the verification models—i.e., finding combinations of designs, scenarios, and requirements—is straightforward. We wish to combine a system design alternative with one scenario (that can stimulate the design alternative) and with a set of requirements (which can be tested using the selected scenario). Having created verification models, automated approaches can be used to run analyses, do post-processing of analysis results, and present the verification results appropriately.

The discussion above contributes to the Research Question 3 (see Section 1.3) regarding the automation of the method steps.

3.9 Conclusion

This chapter discussed what it means to use models for design verification, identified the main modeling concepts, and presented a method for model-based verification of designs against requirements. The proposed method explains how requirements and scenarios should be formalized and what characteristics system design models should have.

By relying on system design simulation models with appropriate levels of detail and accuracy (with respect to the physical phenomena involved) and hybrid⁵⁰ simulations technologies, this approach enables verification of cyber-physical systems (CPS, see (Lee, 2008)). vVDR can be applied based only on models or on real parts, in, for example, hardware in the loop simulations (HIL). In doing so, the application of vVDR can also decrease the effort involved in testing the final product.

Any model that represents the design of a complex physical system is an approximation of the real system to be built. Further, any model may address only a part of the system (e.g. a subsystem) or only a specific aspect (e.g. its dynamic behavior, costs, installation, etc.). Consequently, each model contains uncertainty that results from approximations, abstractions, and simplifications of modeled parts or phenomena and the incompleteness of the context and perimeter considered (see discussion in Section 3.3).

When using vVDR for design verification, the quality of and the confidence in verification results primarily depend on the accuracy of the system design models used. The more precise and complete the system design model is, the more confidence we can have in the verification results produced.

The goal of simulations is to predict the behavior of the real system. In order to do so, models need to be as close as possible to the real system and as accurate as necessary in order to be computationally affordable. The goal of predicting system behavior as accurately as possible matches the task of verifying system design against requirements. For that reason, the vVDR approach is primarily built upon simulation of physical system designs as justified in Section 3.5.2.

Simulation models for complex physical systems typically require continuous-time computation of differential algebraic equations and may be computationally expensive. The most robust and applicable approach for coping with that complexity in the context of design verification is testing (see motivation in Section 3.5.2). Testing is also an integral part of any system development process. The work done when using testing for design verification in the early development phases can be reused to verify products in later development phases. However, any verification that is based on a *subset of all possible* scenarios cannot guarantee satisfaction of properties but only detect their violations (see discussion in Section 3.6).

In vVDR, each requirement is formalized into a model that determines the requirement violation. The answer to the sub-question of Research Question 1 (see Section 1.3) of which requirements can be verified using the proposed vVDR in a particular project depends on the system design models that are in place or are planned to be created. The vVDR method and the prototype implementation in

⁵⁰ The term *hybrid* is used to refer to time-continuous simulation that can simulate non-linear question systems as well as handle events and discontinuities.

ModelicaML (see Chapter 5) focus on enabling verification of different design alternatives against a set of individual requirements in different scenarios. It is designed to automate (see Chapter 4) the composition of verification models and to facilitate a flexible way to reuse formalized requirements models, scenarios, and system design alternative models in various combinations.

The same scenario model can be used to verify different system design alternatives against a number of requirements. In turn, verifying requirements in different scenarios increases the confidence in the verification results. However, the vVDR approach does not integrate individual requirements and hence does not provide the means to determine inconsistencies between requirements. Inconsistencies between requirements can only be discovered when requirement violations are detected during simulations.

By contrast, the Behavior Engineering (BE) methodology (Dromey, 2003) provides the means to ensure consistency within sets of requirements. Any inconsistency or incompleteness detected in the specification is resolved and corrected so that each of the individual requirements does not conflict with any other requirement in the specification.

The presented vVDR approach complements BE and can be integrated with BE (Myers, Schamai, & Fritzson, 2011), for example, by using the notation of BE for expressing requirements and by using the created Model Behavior Tree as the source for the generating vVDR requirements violation monitors and scenarios (Marc-Florian, Ina, & Alain, 2011). In order to do so, the BE method will have to be enhanced so as to create requirement violation monitors that comply with the definitions given in Section 3.7.3. Scenarios generated from the Model Behavior Tree would be used to stimulate a given system design model in order to determine requirement violations.

Chapter 4

Framework for Automation

4.1 Introduction⁵¹

As pointed out in Section 3.7.6, the task of combining and binding models to create verification models is not a trivial one. The challenge is that we wish to find combinations of system design alternatives, scenarios, and requirements and to create valid analysis models (i.e., verification models) for each combination (see Figure 4).

Let us first formulate and break down the problem before we elaborate possible solutions. We can state the question as follows.

Is it possible to automatically find combinations that are composed of one system design model, one verification scenario, a set of requirement violation monitor models, and any other additional models required, and to automatically generate verification models that represent such combinations and contain correct binding code for components?

Breaking down the problem leads to smaller questions:

- *What is necessary to determine if a model requires other models and what data it requires from them?* For example, how do we identify properties of requirement violation monitors that require data from system design or scenario models when combined in a verification model?
- *What is necessary to determine which models can provide data required by other models?* For example, how do we determine if a verification scenario provides the mandatory stimuli to the design model?

⁵¹ This section uses an updated version of sections from (Schamai, Fritzson, Paredis, & Helle, 2012).

- *What is necessary to determine appropriate combinations of models?* For example, how do we determine which verification scenario can be used to verify which requirements by providing the necessary stimuli to the design model?
- *What is necessary to determine which requirements are implemented in the design and can be evaluated?* Moreover, if a requirement is imposed on a component class that is instantiated multiple times in the design, how should we deal with requirements for each of the instantiations?

In turn, more general questions include: *It is possible to achieve automated model composition? Which information needs to be captured a priori in order to enable such automation?*

The discussions in sections below provide answers to the Research Question 3 (see Section 1.3) by presenting how some method steps can be automated.

4.2 Bindings Concept

Section 3.7 introduces the basic modeling artifacts, such as requirements, designs, and scenarios, and the way they should be formalized. The way the formalization is proposed implies that some models will require data from other models when they are used together. Such dependencies will only appear when models are put together. Following the vVDR method there will be models that are created independently, but will become dependent when used together and therefore need to be related to each other. How should we represent such dependencies?

Let us first discuss the subtle part of this question. The model that requires some data does not have to know the exact model that can provide the data. For example, a particular requirement violation monitor will require data from a system design model.

From the perspective of the violation monitor it is not clear which particular design model will be selected for verification and will provide the required data during analysis. Actually, it does not matter at all because any design model to be used for verification will have to provide the required data. Further, such dependencies will be specific to the model structure and can be viewed rather as instrumentation for verification purposes than as part of the models.

Dynamic analysis techniques typically involve instrumentation (Balci, 1995), (Ball, 1999): i.e., enhancement of models for the purpose of collecting information while the models are executed. Clearly, populating models (e.g. requirement violation monitors, design models, or scenarios) with elements that exist only for the purpose of instrumentation is not a good idea.

Consequently, the question is twofold:

1. How can we facilitate the definition of data dependency relations in a loosely coupled way: i.e., without enforcing that both references are known?
2. How should we leave the models that depend on each other free of instrumentation? There are two reasons for this: first, it will facilitate reuse and will avoid cluttering up models with information that is only useful in certain situations. Second, it might not be possible to change the models, for example, if they are from a closed or read-only or from a library.

The new *bindings* concept addresses these questions.

4.2.1 Model Instantiation

Before introducing the bindings concept let us first clarify what a model instantiation is in our context. A model instantiation represents the instance hierarchy.

It is a tree that starts with the root node representing the model being instantiated. Each child node represents a component of the parent model. The recursive tree construction process stops at leaf nodes.

Leaf nodes represent components of primitive types which do not have any further internal structure. Figure 5 shows an example of Modelica models⁵² and the correspondent instance hierarchy.

Model qualified name is the path (structured name) of a model element—e.g. a class or an attribute of a class—that identifies the element within the structure used to organize the model (e.g., by means of packages or nested classes). *Instance path*⁵³ identifies a component within an instantiated model.

Each tree node should contain all relevant information about the element (e.g. the component model qualified name, component instance path etc., see Figure 5).

Instance hierarchy is the basic structure used for inferring bindings as explained in the following sections.

⁵² In Modelica the primitive types, such as Real, Integer, String, and Boolean, still have one level of internal structure of predefined properties, see (Modelica, 2013).

⁵³ The dot-notation for referencing instances is typical for object-oriented languages and may appear differently in other languages.

Models

model V	package PA	package PB	package PC	package PD
...	model A	model B	model C	model D
A a1;	...	...	Real cVar1;	Real dVar1;
C c1;	B b1;	Real bVar1;	Real cVar2;	Real dVar2;
D d1;	B b2;	end B;	end C;	end D;
end V;	end A;	end PB;	end PC;	end PD;
	end PA;			

Instance Hierarchy

Figure 5: Example of instance hierarchy

4.2.2 Basic Concepts

We introduce a notion of *clients* and *providers*. Clients require certain data, providers can provide the required data. However, clients and providers do not know each other a priori.

Further, there may be multiple clients which require the same information which can be provided by one provider. In turn, data from several providers may be needed by one client. This results in a many-to-many relation between clients and providers.

Let us now also introduce the *mediator*⁵⁴ concept for relating a number of clients to a number of providers as illustrated in Figure 6 (answer to question 1 in Section 4.2).

Relations between clients and mediator and mediator and providers are stored in mediators so that elements that represent client or provider are not modified

⁵⁴ The idea of mediators is similar to mediator pattern from the software development domain (Gamma, Helm, & Johnson, 1994), which promotes the idea of a loose coupling of objects to enable their communication without knowing each other explicitly.

(answer to question 2 in Section 4.2). Moreover, the mediator concept allows an independent specification of clients and providers.

Figure 6: Basic concept of client, mediator and provider

Figure 7 illustrates the usage of the bindings concept in the context of vVDR. It shows that some properties of requirement violation monitors, design, and scenario models are clients and other are providers. Clients and providers are referenced by mediators.

Figure 7: Bindings concept in vVDR

Definition 10. Client: A client is any component of a model that requires data from other components.

Definition 11. Provider: A provider is any component of a model that provides data for other components.

Definition 12. Mediator: A mediator relates a number of clients to a number of providers. It groups all clients that require the same data and all providers that provide that data.

Definition 13. Mediator Container: A mediator container contains mediators. The main purpose is to group (i.e., to organize) mediators for specific concerns (e.g., requirement inputs, system mandatory stimuli, failure injection, etc.).

Definition 14. Client Reference: A client reference is a relationship between one mediator and one client. The relationship includes the *model qualified name* of the client component.

Definition 15. Provider Reference: a relation between one mediator and one provider. The relation includes the *model qualified name* of the provider component.

For example, given the following components from the example in Figure 5 assumed to be clients that require the same data:

- *PC.C.cVar1*
- *PD.D.dVar2*

The provider component that can provide the data required by the clients above is:

- *PB.B.bVar1*

Note, at this point it is not clear how the clients and the providers should be bound because until now there is no model that includes instances of all of them. Consequently, we can only use the model qualified names to identify them.

Mediator m1 relates the clients and providers above by means of their model qualified names:

- *m1 clients: PC.C.cVar1, PD.D.dVar2*
- *m1 providers: PB.B.bVar1*

The main purpose of a mediator is to allow a determination of whether a binding⁵⁵ expression can be inferred for a given client based on the referenced providers. A binding is defined as follows:

Definition 16. Binding: A binding has the form “client instance reference = binding expression.” A binding is a causal relation which specifies that, at any simulated time, the value for the referenced client instance shall be the same as the

⁵⁵ Semantically, the term *binding* is similar to the term *binding equation* in Modelica: “binding equation: Either a declaration equation or an element modification for the value of the variable. A component with a binding equation has its value bound to some expression” (see Section 8.1.), (Modelica, 2013, p. 241). Originally the term *value binding* was used by (Schamai, Fritzson, Paredis, & Helle, 2012). In this text we will use *binding* for brevity.

value computed by the right-hand expression. The right-hand, i.e. the binding expression, is inferred using the corresponding mediator that references the client at hand and all available providers. Optionally, client, provider or mediator operations may contain scripts that will need to be interpreted when inferring the binding expression.

For example, an inferred binding for $cl.cVar1$ (the instance of $PC.C.cVar1$ in the instantiation V) may look as follows: “ $cl.cVar1 = al.bl.pVar1$ ”.

Any component of a model can be a client. It is also possible for a component to be client and provider at the same time, unless it is referring to itself or two components point to each other in such a way that would result in an algebraic loop.

Furthermore, some of the clients may be *mandatory*: i.e., they will have to be bound in any case. Other may be optional. Optional clients must have default values to account for cases when they are not bound to any provider. For example, any client of a requirement violation monitor must be a mandatory client because each of them is needed for the evaluation. In contrast, some of system design model clients are mandatory and some may be optional because not each scenario will have to provide all potential stimuli.

Definition 17. Mandatory Client: A *mandatory* client indicates that this model component must be bound to some provider during analysis.

4.2.3 Binding Operations

The relation between clients and providers can convey further information called *client*, *provider* and *mediator operations*. A client or provider reference can contain an *operation* (see Section 4.2.2). The purpose of client or provider operations is to make the definition of bindings flexible and, more important, to enable referencing particular client or provider subcomponents. A mediator can also have an operation for reducing lists of providers as explained below.

In the following sections the concepts are defined theoretically. Chapter 5 and Chapter 6 explain the concepts based on concrete examples, including concrete syntax for operations.

4.2.3.1 Mediator Operation

As mentioned above the main purpose of the bindings concept is to automatically infer the binding for a client. Consider the following possible cases, resulting from the fact that one client is related to several providers by using a mediator, when inferring bindings for a particular client ci_k in an instantiation I . A mediator that references ci_k can have the following number of referenced providers which exist in I :

- 1 provider
- n providers
- 0 providers

1) If there is one provider, inferring the binding is straightforward. All we need to do is process the binding operations (if specified) and generate the binding expression.

2) If there are n providers there are two possibilities:

2a) It is intentional, and all the referenced providers are necessary to provide the required data. In the first case the mediator binding operation should provide means for reducing the list of providers to an expression that returns only one value. For example, similar to the array reduction functions in Modelica (Modelica, 2013), such as for finding the minimum or the maximum value, sum values, build a product, or to use the logic operators such as AND, OR, or XOR.

For example, assume the client needs the information about the overall system weight being a sum of the weight of each individual component. Instead of summing up the component weights in the system design model and offering one single provider, it may be a good idea to reference the weight property of each component and let the mediator generate the sum expression for this binding⁵⁶.

2b) It is *not* intentional and is instead because of the given model structure. For example, assume a client for a requirement violation monitor requires the current value of liquid in a tank. If the provider is a property of a tank model and there are several tanks in the design, then there will be multiple providers to choose from. In this case user interaction will be necessary to manually decide which provider should be used for a particular client. A possible way to store the manual decisions is discussed in Section 4.2.5

For the case in 2b), consider the example in Figure 5. Assume that the following mediator is defined:

- *m1 clients: PC.C.cVar1, PD.D.dVar2*
- *m1 providers: PB.B.bVar1*

When inferring the binding for *c1.cVar1* (the instance of *PC.C.cVar1* in the instantiation *V*) there will be 2 providers that exist in this instantiation:

- *a1.b1.pVar1*

⁵⁶ The concrete syntax will depend on the action language used.

- *a1.b2.pVar1*

The reason is the resulting model structure. The provider owning class *PB.B* is instantiated several times in class *PA.A*.

3) Having zero providers means that:

- there are no providers in this combination. Then no binding can be inferred for this client.
- or, the mediator should provide constant values to the referenced clients. For this case the mediator operation should provide syntax for expressions to capture constant data.

Recall that a mediator always represents the need of the client. Consequently, any inferred binding must be compatible with the types of the subscribed clients. Since providers and clients do not know each other explicitly, it is the task of the mediator to indicate the type compatible with all referenced clients.

Definition 18. Mediator Operation: A mediator operation is an expression that contains macros for reducing a list of providers to an expression used for calculating only one value, or an expression that captures constant data or model function calls.

From the example in Figure 5, assume that a mediator *m* has the following mediator operation:

`SUM ( : )`

`SUM ( : )`⁵⁷ is an example of an array reduction function macro which will be expanded to an expression that calculates the sum of all provided values.

4.2.3.2 Provider Operation

The provider operation can be used when the actual provider is a subcomponent. For example, imagine a mediator *all valves are fully closed*. Which valves exist in the system and how to determine if all of them are closed can be expressed by specifying the component that contains all valves (e.g. the valve subsystem) to be the provider, and by using the each of the relevant valve instances to express what is meant by *all valves are closed*.

Another example for using provider binding operations is value-type conversion, which may be necessary in order to provide data compatible to the type of the client.

⁵⁷ “:” indicates that this function operates on a list of provided items, i.e., on a list of providers.

Definition 19. Provider Operation: A provider operation is an expression that during analysis will return a single value. The expression may contain references to provider subcomponents. Any reference to the provider itself or its subcomponents starts with a placeholder representing the future provider *instance path*. This path will be replaced with the concrete path when the provider model is instantiated within another model.

From the example in Figure 5, assume that the provider is referenced by the mediator *m*:

- *m1 providers*: PB.B

A provider operation associated with this provider reference may be:

```
providerPath.bVar1 > 0
```

providerPath is an example of a placeholder for the future instance path. Assume the provider is of type *Real*. Then the specified expression converts it to a *Boolean* type of value.

4.2.3.3 Client Operation

Similar to the provider binding operation, client operations are typically used when a subcomponent should be referenced.

Another usage example for client binding operations is when the client is subscribed to an existing mediator that does not provide a compatible type. Then the binding operation can be used for value-type conversions, as in provider binding operations.

Definition 20. Client Operation: A client operation is a list of relations of the form “left side = right side”. Each specifies binding for one client. The left side is the reference to either the client itself or one of its subcomponents. Any reference of the client itself or its subcomponent starts with a placeholder representing the future client path. This path will be replaced with the concrete path when the client model is instantiated within another model. The right side must contain a dedicated placeholder which indicates the place where the inferred binding expression should be written, and may contain further expressions involving constant data or function calls.

From the example in Figure 5, assume that a mediator *m* references client PD.D.dVar1. A client operation attached to this client reference may appear as follows:

```
clientPath.dVar1 = getBinding();
clientPath.dVar2 = getBinding();
```


`clientPath` is an example of the placeholder for the future instance path. `getBinding()` is an example of the indication where the inferred binding expression should be written.

The order within the list of relations implies the priority. If there is a binding for the same client, then the top binding⁵⁸ in the list will be used and a warning will be indicated to the user.

4.2.3.4 Overwriting of Bindings

Using client operations it is possible to specify several bindings for the same client. Assume the following situation:

- *m1* clients:
 - `PD.D.dVar2`,
 - `PC.D{clientPath.dVar2 = getBinding();}`

The mediator *m1* references the client `v.d1.dVar2` (instance of `PD.D.dVar2` in *v*) two times. The first client reference points directly to the `PD.D.dVar2` element. The second client reference points to the same client using the operation `clientPath.dVar2=getBinding();`. In such a case the top-level binding specification (i.e., `PC.D{clientPath.dVar2=getBinding();}`) overwrites the one further down in the instance tree. This mechanism is similar to the Modelica modification (Modelica, 2013, p. 70) and has the advantage that existing bindings can be overwritten at upper levels without a need for changing lower-level models.

4.2.4 Validity of Binding Operations

Before considering an algorithm or functions for inferring binding expressions (presented in Section 4.1 and 4.7), let us first define valid binding operations. Note, in the following some concrete syntax is used for explanations. It is the syntax specified in Appendix B. The concrete syntax is to be adopted to the particular action language used to implement the binding concept.

First we recap (Section 4.2.3) what binding operations are:

- **Client Operation:** A set of relations owned by a client reference (see Section 4.2.1). The right side is an expression that contains the client binding function (e.g. `getBinding()`), which indicates the position at which the inferred binding expression is to be placed. The left side is the reference to the client itself or to one of its subcomponents. The reference

⁵⁸ The one that most likely is the newest.

starts with a placeholder (e.g. `clientPath`) that will be replaced with a concrete path in a verification model (see Section 4.2.3.3).

- **Provider Operation:** Expression owned by the provider reference (see Section 4.2.3.3). It is the right-hand expression of the binding. The expression can involve references to the provider itself or its subcomponents. Each reference starts with a placeholder (e.g. `providerPath`) that will be replaced with a concrete path in a verification model (see Section 4.2.3.2).

Mediator operation is an expression owned by a mediator. It can only be an expression involving reduction function macros, such as `SUM(:)`, `AVG(:)`, `PROD(:)`, etc., that use a collection of providers to calculate a single value. Such macros will need to be expanded to produce model code. For example, using the Modelica language for binding operations the `SUM(:)` macro will be expanded into “... the scalar sum of all the elements of array expression: `A[1,...,1]+A[2,...,1]+...+A[end,...,1]+A[end,...,end]`” (Modelica, 2013, p. 113), where the array `A` is the collection of providers referenced by the mediator.

Mediator operations can take one of the following forms:

- **Constant mediator operation:** a mediator operation that contains no reduction function macros, but only an expression involving numbers or model function calls.
- **Multi-provider mediator operation:** a mediator operation that contains an expression involving reduction function macros.

A client, mediator, or provider is said to be valid either if it contains no operation or if it contains an operation that complies with the descriptions above.

Given an instantiation tree V that contains valid clients and providers, we define the following sets:

- **C_S:** Set of clients in the given instantiation,
- **P_S:** Set of providers in the given instantiation,
- **M_S:** Set of mediators that are referenced by clients and providers from C_S and P_S respectively.

For each valid client c_i from C_S a valid binding expression can be inferred if there is exactly 1 valid mediator from M_S and one of the following cases applies:

- The mediator contains no operation and there is exactly 1 provider from P_S
- The mediator contains a constant mediator operation and there are 0 providers from P_S.

- The mediator contains a multi-provider mediator operation and there are n providers from P_S .

If there are multiple providers, and no mediator operation that can handle that, user interaction and manual decision will be required. The case, if there is one mediator containing a constant mediator operation and there are multiple providers should result in an error message. Further, the case if there are multiple mediators for one client is also an error.

Definition 21 Valid Binding: A binding is said to be valid if a binding expression can be inferred for client ci_k and the resulting type of the right-hand binding expression is compatible with the left-hand-side expression.

4.2.5 Preferred Bindings

On page 66 we discussed the case in which there are n providers and no mediator operation that includes reduction function macros. In such cases a manual decision is required to know which of the providers should be used for binding. Taken decisions can be stored in so-called preferred bindings in order to increase efficiency.

Definition 22. Preferred Binding: A preferred binding captures the following information: *client model qualified name*($cQName$), *client instance path*($cPath$), and *provider model qualified name*($pQName$), *provider instance path*($pPath$). A preferred binding means that whenever for the client that has $cQName$ and $cPath$ there are multiple providers the one with $pQName$ and $pPath$ should be used.

4.3 Applications for Bindings

The formalization proposed by vVDR and the bindings concept have a number of applications. The following subsections describe a few of them. The first three applications are prototyped in this dissertation and shown in the examples in Chapter 6.

4.3.1 Automated Composition of Verification Models

The main motivation for the bindings concept is pointed out in Section 3.8. We want to enable automated generation of verification models. This challenge is twofold:

- First, we need to find combinations of one design, one scenario, and one set of requirements. Thereby, we want to ensure that only those scenarios should be considered which can stimulate the system. Further, we want only to consider requirements that are already addressed in the given design, and for a particular scenario, only those requirements for which this scenario is appropriate for verifying the particular requirement.
- Second, we want to generate valid binding expressions for each of the clients in the verification models.

Figure 8 illustrates how bindings can be leveraged to create verification models. It starts with the one design alternative selected by the user.⁵⁹ Then it iterates over all scenarios. For each of the scenarios it checks whether all mandatory clients of the design and this particular scenario model can be satisfied. If yes, the scenario is selected because we can assume that it will stimulate the design under consideration and it will receive all necessary feedback from the design. If not all mandatory clients can be satisfied, then this scenario is discarded.

Satisfying a client means that it is possible, based on the given set of mediator definitions, to infer a correct binding expression for the client from the providers available in the combination under consideration.

Further, having selected a scenario, the algorithm iterates over all requirements that are referenced by this particular scenario with a relation indicating that this scenario can be used to verify designs against this particular requirement. For each requirement, again the algorithm checks whether all mandatory requirement violation monitor clients can be satisfied. If yes, the requirement is added to the combination. If no, then we assume that this requirement is not completely or not correctly addressed in the given design and cannot be evaluated yet.

When the loop terminates with a selected scenario and one or more requirements we have found a combination that can be translated into a verification model by instantiating the design, selected scenario, and requirements and generating bindings for each of the clients.

When additional models are used, as explained in Section 3.7.6, any step in Figure 8 that checks whether clients are satisfied must include the consideration of all additional models that are needed by either design, scenario, or requirement models.

⁵⁹ In vVDR the tester will be performing this task.

Figure 8: Illustration of the verification models generation algorithm

4.3.2 Discovery of Relations between Scenarios and Requirements

Assume there is a large number of requirements, and, as system development progresses, new scenarios are constantly being added. For a tester, it may take a great deal of effort to define explicit relations for all requirements when adding a new scenario. Moreover, in large projects there will be several testers who will focus on testing specific aspects or only subsystems. None of them would be concerned with all requirements, but only a subset of them.

It also may happen that a relation between requirements and scenarios is not obvious to testers, so that some requirements would not be referenced by a particular scenario even though this scenario could be good for testing them. Consequently, it is a good idea to enable automated discovery of relations in an exhaustive fashion and to store the new knowledge. A tester can store new relations discovered and repeat the procedure when needed: for example, when new requirements or scenarios are added, or when requirements, scenarios, or system design models are modified.

Ultimately, the generation of models based on explicit relations between scenarios and requirements will be used for verification sessions. In doing so, the generation of verification models is deterministic and the tester has rigorous control over which scenarios will be combined in verification models. However, the relations discovery feature may be used in advance to ensure consistency of these

relations and to ensure that requirements will be evaluated in as many scenarios as possible in order to increase the confidence in the verification results.

4.3.3 Traceability of Requirements, Designs and Scenarios

The proposed formalization approach, the bindings concept presented, and the algorithm for finding combinations (see Section 4.1) can be used for providing traceability between requirements, designs, and scenarios. For example, having selected one particular system design model, we can:

- List all requirements that are implemented by the design, and, based on defined relations between scenarios and requirements, list for each requirement all scenarios that can be used to verify the selected design against this particular requirement.
- List all scenarios that can stimulate the selected design. And, for each scenario, list all requirements that can be evaluated using this particular scenario.
- List all requirements that are not implemented, i.e., not addressed completely or correctly, in the selected design. If some requirement violation monitor clients cannot be satisfied by the selected design, then the requirement is not completely addressed in this design and cannot be evaluated.
- List all scenarios that cannot be used to stimulate the selected design. If the mandatory design clients are not satisfied by the scenario providers, then the scenario does not provide all stimuli required for the selected design.

Note however, the algorithm presented in Figure 8 is a static type of analysis which cannot provide complete answers and should be considered as a step preceding the dynamic analysis. Having concluded that a particular scenario can stimulate the selected design (i.e., that all mandatory clients of the design model will be satisfied by providers in the scenario model), the course of action specified in the scenario may be defined in such a way that none of the providers used will change its value during analysis and thus will not stimulate the system properly. Such situations can only be securely discovered way using dynamic analysis (e.g. simulation).

Whether a selected scenario will stimulate the system so that all selected requirement violation monitors will be evaluated can also be answered only by the dynamic analysis. For that, a dynamic analysis (e.g. simulation) of the verification model and the resulting status of the violation monitors will provide the ultimate answer. Consequently, verification models, which will be generated using this algorithm, may be optimal or suboptimal (see definition in Section 3.7.6). Chapter 6 discusses when suboptimal verification models are useful.

4.3.4 Detection of Redundant or Conflicting Requirements

The way requirements are formalized, combined with the bindings concept, can also enable the detection of inconsistencies in sets of requirements. The following examples are simplified to only show how the vVDR approach and the bindings concept can be leveraged. Detection of inconsistencies within requirement sets is far more complicated and difficult to achieve.

4.3.4.1 Redundant Requirements

Examples of redundant requirements are statements that address the same requirement by using different formulations:

- *The level of liquid in a tank shall not exceed 80% of the tank height.*
- *The maximal level of liquid in a tank shall be less than 80% of the tank height at any time.*

The two requirements above state essentially the same thing. When formalized into violation monitors, these requirements would most likely have the same providers used to infer their bindings. Moreover, the analysis trace (violation monitor status over time) will most likely look that same. Consequently, it would be possible to identify such requirements by comparing their bindings (in particular the providers used) and their traces when simulated with the same scenario.

4.3.4.2 Conflicting Requirements

Multiple requirements that address the same thing but specify different values are likely to conflict with each other. Consider the following examples:

1. *The level of liquid in a tank shall not exceed 80% of the tank height.*
2. *The level of liquid in a tank shall not exceed 90% of the tank height.*
3. *The maximal level of liquid in a tank shall be less than 1 m at any time.*

Clearly, the first two requirements are in conflict. Since they are very likely to have the same providers, we could select an appropriate scenario (e.g. by looking for a scenario that references both requirements with the relation *useToVerify*; see Section 3.7.5.1), create a verification model, and simulate it. If the trace is not the same then we have potentially detected a conflict.

If it happens to be that the value of 80% of the tank height is greater than 1 m, then the first and the third requirements will be in conflict. However, in this case it would be difficult to identify them automatically because their provider sets would not be identical. Heuristics may help: for example, if multiple requirements have at least one common verification scenario and the provider set overlaps to some extent, we may consider creating verification models and analyzing the traces in order to draw conclusion on potential conflicts.

4.3.5 Impact Analysis

Yet another application for bindings could be in support of impact analysis in the context of vVDR. For example, it could help to answer the following questions:

- If the system design model changes, which requirements are affected? Support in answering this question would involve an analysis of all providers contained in parts of the system design model that were modified. Determining which clients of which requirements are bound to the identified providers will give hints as to which requirements should be re-verified to see if the change affects their violation status.
- If scenarios have changed, which requirements may be affected? Support in answering would analyze relations between scenarios and requirements and return a list of requirements that should be re-verified.
- If a requirement changes, we might be interested in determining a subset of scenarios to be used for a new verification session.

4.4 vVDR Enhancements

To capture information about clients and providers, the vVDR method has to be extended as follows:

- Each model that requires data from other models should express this need by creating a new mediator or by subscribing to an existing one.
- Each mediator must have defined⁶⁰ providers so that the correct binding code for its clients can be determined.

In a collaborative work environment, a requirement analyst, who formalizes requirements, will express the need that a particular requirement violation monitor needs data for evaluation by creating a new mediator or by subscribing to an existing one. The system designer, who formalizes designs, will respond to this need by defining one or more providers for each mediator. In turn, a design model will require stimuli from the verification scenario, and the system designer will create new or reuse existing mediators to express this need. That will be answered by the tester, who will create verification scenarios and define the appropriate providers for each concerned mediator.

If clients and providers are defined, then binding code can be automatically derived for each client and automated generation of verification models will be

⁶⁰ This is true unless this mediator holds constant data. This is a special case for using mediators, for example to define different parameter sets. Such parameter sets can then be loaded and used for a particular verification session.

possible. The following tasks of the vVDR method (see Section 3.7) should now include the specification of clients and providers (marked in bold below):

- Formalize Requirements
 - **Specify clients and providers**
- Formalize Design
 - **Specify clients and providers**
- Formalize Scenarios
 - **Specify clients and providers**

Each of these steps contributes to bindings specification (Figure 9).

Figure 9: vVDR steps contributing to the specification of bindings

Moreover, the overall vVDR picture in Figure 10 now indicates which steps can be automated (compare to Figure 1). The algorithm for composing verification models (mentioned in Section 4.3.1) is presented in detail in Section 4.6 and Section 4.7. After verification models are created they can be translated into executable form and simulated one after another. The simulation results produced are then post-processed and conclusions on requirement violations etc., see Section 3.7.8, are presented in a verification session report.

Figure 10: Enhanced vVDR overview

4.5 vVDR Language Support

The following table captures the requirements derived from the discussion in the above sections.

ID	Text
LR 12	The language shall include the concept <i>client</i> (i.e., any component that requires data during analysis).
LR 13	A client can be mandatory or optional.
LR 14	The language shall include the concept <i>mediator</i> .
LR 15	The language shall include the concept <i>provider</i> (i.e., any component that provides required data during analysis).
LR 16	The language shall allow each mediator to reference a set of clients and providers.
LR 17	The language shall enable the specification of an optional operation for each mediator.

LR 18	The language shall enable the specification of an optional operation for each relation between a client and a mediator.
LR 19	The language shall enable the specification of an optional operation for each relation between a provider and a mediator.

4.6 Algorithm for Generating Verification Models

We will now enhance the algorithm sketched in Figure 8 based on the discussion in Section 3.7.6 and elaborate it in pseudocode⁶¹.

Notes regarding collections in pseudocode:

- A list is an ordered collection of elements. In the algorithm below all lists order their items (e.g. items representing models), based on pre-defined criteria (e.g. alphabetically based on the model name). A list is indicated by the postfix “**_L**”.
- A set denotes an unordered collection of items that contains no duplicates. A set is indicated by the postfix “**_S**”.
- A map denotes an unordered collection of linked items that contains no duplicates. A map is indicated by the postfix “**_M**”.

Algorithm 4-1: Generate Verification Models

Input: Element representing system design model dm
Locations to search for scenarios LS_S
Locations to search for requirements LR_S
Locations to search for mediators LM_S
Locations to search for additional models relations LA_S

Output: List of Verification Models VM_L

begin

```

01  cd = collectData(LS_S, LR_S, LM_S, LA_S) // Collect all relevant data
02  S_L = getScenarios(cd) // List of found scenarios
03  R_L = getRequirements(cd) // List of requirements found
04  M_S = getMediators(cd) // Set of mediators found
05  A_S = getAdditionalModels(cd) // Set of additional models
06  I_M = empty //Map of models and their instantiation trees

```

⁶¹ A sample implementation of this pseudocode can be found in the SVN repository (within the *org.openmodelica.modelicaml.helper.generators.example* package). See ModelicaML website (Schamai, 2013) for SVN repository access instructions.

```

07 for each  $s_k$  in  $S\_L$  do
08    $B\_M = \text{empty}$  //Map of instantiation tree nodes (clients) and inferred bindings

09   log("combining  $s_k$  with  $dm$ ")
10    $V = \text{newInstantiation}()$  // Instantiation root node

11*1  $I\_M, V = \text{instantiate}(\{dm\})$  // Create instance hierarchy and add to map
12    $\{dmi\} = \text{getInstantiation}(\{dm\}, I\_M)$  // Get instantiation from the map

13    $AD\_S = \text{getAdditionalModelsFor}(dm, A\_S)$  // Find additional models
14    $I\_M, V = \text{instantiate}(AD\_S)$  // Set of instantiations
15    $\{adi_0 \dots adi_n\} = \text{getInstantiation}(AD\_S, I\_M)$  // Set of instantiations

16    $I\_M, V = \text{instantiate}(\{s_k\})$ 
17    $\{si_k\} = \text{getInstantiation}(\{s_k\}, I\_M)$ 

18    $AS\_S = \text{getAdditionalModelsFor}(s_k, A\_S)$  // Additional models for  $s_k$ 
19    $I\_M, V = \text{instantiate}(AS\_S)$ 
20*2  $\{asi_0 \dots asi_n\} = \text{getInstantiation}(AS\_S, I\_M)$ 

21*3  $B\_M = \text{inferBindings}(\text{getClients}(\{V\}), M\_S, B\_M)$ 

22*4 if  $\text{allMandatoryClientsSatisfied}(V, B\_M)$ 
    and  $\text{usesAtLeastOneProviderFrom}(\text{mandatoryClientsFrom}(dmi), si_k, B\_M)$  then

    // Find requirements for scenario  $s_k$ 
23*5  $RS\_L = \text{getRequirementsFor}(s_k, R\_L, \text{selectionOptions})$ 

    // Analyze requirements
24    $R_{\text{comb\_}S} = \text{empty}$  // Empty set of requirements
25   for each  $r_k$  in  $RS\_L$  do

    // Instantiate all models needed by this requirement in addition
26*6  $AR\_S = \text{getAdditionalModelsFor}(r_k, A\_S)$ 
27    $I\_M, V = \text{instantiate}(AR\_S)$ 
28    $\{ari_0 \dots ari_n\} = \text{getInstantiation}(AR\_S, I\_M)$ 

    // Instantiate requirement in order to determine if binding can be derived for its clients
29    $I\_M, V = \text{instantiate}(\{r_k\})$ 
30    $\{ri_k\} = \text{getInstantiation}(\{r_k\}, I\_M)$ 

31    $B\_M = \text{inferBindings}(\text{getClients}(\{ri_k\}), M\_S, B\_M)$ 

32*7 if  $\text{allMandatoryClientsSatisfied}(ri_k, B\_M)$  then
33   log("rk is included in the combination of  $dm$  and  $sk$ .")

```

```

34      addToCollection( $R_{comb\_S}$ ,  $r_k$ ) // Selected requirements

// Determine if this requirement should be instantiated multiple times
35*8      addInstNumber = getAdditionalInstantiationsNumber( $r_k$ , B_M)
36      if addInstNumber > 0 then
37          I_M, V = instantiateN( $r_k$ , addInstNumber)
// Get all requirement instantiations
38          { $rim_0 \dots rim_k$ } = getInstantiation( $r_k$ )
39          B_M = inferBindings(getClients({ $rim_0 \dots rim_k$ }), M_S, B_M)
40*9      end if

41      else
42          removeChildren(V, {  $r_k$  }) // Remove requirement
43          log(" $r_k$  is discarded because not all mandatory clients can be satisfied.")
44      end if
45  end for

// Analyze all additional models collected in this combination
46*10 AMI_S = { $adi_0 \dots adi_n$ } + { $asi_0 \dots asi_n$ } + { $ari_0 \dots ari_n$ }
47  for each  $ai_k$  in AMI_S and in do
48      if not usesAtLeastOneProviderFrom(mandatoryClientsFrom(V),  $ai_k$ , B_M)
49          then
// Prune the additional model if no providers are used by other models.
49          removeChildren(V, {  $ai_k$  })
50          log(getModelFromInstantiation( $ai_k$ ).getQualifiedName() + "is pruned
because none of its providers is used in this combination.")

51      else
52          log(" $a_k$  included as addition model.")
53      end if
54*11 end for

// Create verification model. Optionally, check whether some requirements are included
at all. If there are no requirements, we can discard this combination
55*12 vm = createVerificationModel(V, B_M)

56      addToCollection(V_M_L, vm) // Add vm to the list to be returned
57      log("Combination of dm and sk. and  $R_{comb\_S}$  was successful.")

58  else
59      log("combination  $s_k$  and dm is discarded because  $s_k$ 
will not stimulate all inputs
of dm. The following clients ( $client_0 \dots client_n$ ) are not satisfied.")

60  end if
61 end for

```

end

The input for the algorithm is the selected system design model to be verified against requirements using scenarios, and sets of locations to search for requirements, scenarios, mediators, etc. For example, using languages such as UML or Modelica, such locations would be the qualified names of packages or classes to be considered. This way, the user has control over the sets of requirements, scenarios, mediators, or any models additionally required, which will be considered for this particular verification session. The output of the algorithm is a list of the verification models generated, which can then be used for the analysis (e.g. simulation).

The algorithm starts by pre-collecting data that will be used afterwards multiple times. This pre-collection avoids the expensive search for models over and over again. The specific collections of requirements, scenarios, etc. are created from the pre-collected data.

The construction of combinations of the selected system design model, one scenario that can stimulate the system model, and all requirements that are implemented and can be evaluated using this particular scenario, starts by iterating over the set of all scenarios found.

The function `newInstantiation()` creates a new instantiation tree root node. An instantiation tree is the explicit hierarchical instance structure of a model (see Section 4.2.1).

The system design model, scenario, and all models additionally required are instantiated and added to the instantiation root node `V` (lines marked with *1-*2).

So far, the combination contains the system design model, the verification scenario, and all additional models. In the line marked with *3 the algorithm infers bindings for all clients in the instantiation tree, and in the line marked with *4 it checks whether the all mandatory clients are satisfied. It then makes sure that some providers from the scenario are used by the design model, i.e., ensuring that this scenario is likely to stimulate the system design model. If both evaluate to true, the scenario is *selected* for this combination and the algorithm goes on with considering requirements.

In the line marked with *5 the algorithm obtains requirements for the selected scenario. The `selectionOptions` arguments specify the criteria for selecting requirements. For example, `selectionOptions` may specify that only requirements should be considered for which there is an explicit (`useToVerify` or `doNotUseToVerify`) relation to the selected scenario (see Section 3.7.5.1). Another strategy may consider any of the requirements found.⁶² In the lines marked

⁶² A useful application of such strategy is described in section 6.2.5.2.

with *6-*7, for each requirement the algorithm adds all additional models referenced by the requirement, instantiates the requirement, and checks whether all mandatory clients⁶³ are satisfied. It then checks whether this requirement should be instantiated multiple times (lines marked with *8-*9).

Having selected the requirements, the algorithm prunes all additional models from which no client is used (e.g., the lines marked with *10-*11) in this combination in order to keep the verification model concise. Finally, the verification models are created (the line marked with *12).

At any step the in condition that evaluates to false, the algorithm logs comments on the model composition process. The logged information is crucial for understanding why scenarios or requirements were discarded.

The presented algorithm includes several functions for inferring bindings and for determining whether clients are satisfied or providers are used.

Function 4-1: `instantiate({m0 ... mk}, I_M, V)` “Instantiate model”

Input: List of models {m₀ ... m_k}
 Map of models and their instantiation trees I_M
 Instantiation V
Output: Updated map of models and their instantiation trees I_M

```

begin
01 for each m in {m0 ... mk} do
02   mi=getExistingInstantiation (m, I_M)
03   if not instantiationExists(mi) then
04     mi = instantiate(m)
05     addToMap(I_M, mi) // Add to instantiation map
06     addChildren(V, {m}) // Attach instantiation to the root node of V
07   end if
08 end for
09 return I_M, V
end

```

Function 4-2: `inferBindings({ci0 ... cik}, M_S, B_M)` “Infer bindings for components”

Input: List of clients to infer bindings for {ci₀ ... ci_k}
 Set of mediators M_S
 Map of instantiation tree nodes (clients) and their bindings B_M
Output: Map of instantiation tree nodes (clients) and their bindings B_M

⁶³ For a requirement violation monitor all clients are mandatory.

```

begin
01 for each  $c$  in  $\{c_{i_0} \dots c_{i_k}\}$  do
02   if not bindingExistsFor( $c$ ,  $B\_M$ ) then
03     binding = inferBinding( $c$ ,  $M\_S$ ) // Binding object
04     if binding.isBindingPossible then
05       addToMap( $B\_M$ ,  $c$ , binding) // Add to bindings map
06     end if
07   end if
08 end for
09 return  $B\_M$ 
end

```

Section 4.7 discusses how the *inferBinding()* function can be implemented. The function for determining whether all mandatory clients are satisfied is presented in pseudocode below.

Function 4-3: allMandatoryClientsSatisfied(V , B_M) “Check if all mandatory clients are satisfied”

Input: Reference to the instantiation tree root node V
 Map of instantiation nodes (clients) and their inferred bindings M_S
Output: true or false

```

begin
01  $C\_S$  = getAllMandatoryClients( $V$ ) // Set of mandatory clients in instantiation tree
02 if isEmpty( $C\_S$ ) then
03   return false // If there are no clients, return false
04 end if
05 for each  $c_k$  in  $C\_S$  do
06   if not bindingExistsFor( $c_k$ ,  $B\_M$ ) then
07     return false
08   end if
09 end for
10 return true
end

```

The function presented below determines whether at least one of the providers from a component is used in the inferred binding for clients of another component.

Function 4-4: `usesAtLeastOneProviderFrom({ci0 ... cik}, cip, B_M)` “Check if at least one provider is from the instantiation subtree”

Input: Clients {ci₀ ... ci_k}
 Instantiated component (instantiation tree node) with providers ci_p
 Map of instantiation nodes (clients) and their inferred bindings M_S
Output: true or false

```

begin
01 C_S = {ci0 ... cik} // Set of mandatory clients in instantiation tree
02 for each ck in C_S do
03   binding = getBindingFor(ck, B_M) // Binding object
04   if isDefined(binding) and binding.isBindingPossible then
05     P_S = binding.getAllPossibleProviders() // Providers used for ck binding
06     if not clientIsFrom(ck, cip) and atLeastOneProviderIsFrom(cip, P_S) then
07       return true
08     end if
09   end if
10 end for
11 return false
end

```

The function for determining the number of required instantiations is shown below.

For a given instantiated requirement the algorithm iterates over all mandatory clients in order to obtain the maximal number of providers. If the corresponding mediator does not include reduction functions in its operation (see explanations in Section 4.2.4) then the maximal number of providers indicates the number of components for which this requirement is imposed.

Consequently, this requirement should be instantiated for each of the components. In cases where requirements are instantiated multiple times, the user will have to decide manually which of the requirement instantiations should be bound to which component within the instantiated system design and may store the manual decisions using preferred bindings (see Section 4.2.5).

Function 4-5: `getAdditionalInstantiationsNumber(ci, B_M)` “Compute the require number of requirement instantiations.”

Input: Instantiation tree node ci
 Map of instantiation nodes (clients) and their inferred bindings M_S
Output: Integer additionalRequiredInstantiationsNumber

```

begin
01 C_S= getAllMandatoryClients(ci) // Set of mandatory clients
02 additionalRequiredInstantiationsNumber = 0

```

```

03 for each  $c_k$  in C_S do
04   binding = getBindingFor( $c_k$ , B_M) // Binding object
05   mediator = binding.getMediator() // Mediator used for inferring bindings

06   if not mediatorOpContainsReductionFunctions(mediator) then
// Number of providers for manual selection
07     numberOfProviders = binding.getProvidersForSelection().size()

08     if numberOfProviders > additionalRequiredInstantiationsNumber then
09       additionalRequiredInstantiationsNumber = numberOfProviders - 1
// "- 1" because we are looking for the additional number of instantiations
10     end if

11   end if
12 end for

13 return additionalRequiredInstantiationsNumber
end

```

The system design model, requirements, and additional models will need to be instantiated in several runs of the outer loop. For reasons of efficiency,⁶⁴ it is crucial to keep and reuse the instantiation tree objects in iterations of the outer loop.

4.7 Algorithm for Inferring Bindings

Functions that determine whether clients are satisfied or whether providers are used are crucial for the algorithm used to generate verification models presented in Section 4.1. The core feature used in such functions is the ability to infer binding expressions for clients. In order to enable this, the following is required:

- instantiation of the model (as explained in Section 4.2.1), for example, of a verification model
- means for identifying clients,
- means for identifying mediators,
- means for identifying providers.

⁶⁴ Requirements and scenarios are typically simpler models compared to system design models. Instantiation of system design models may become computationally expensive. A lot can be gained by reusing the instantiations created.

Furthermore, as explained in Section 4.2.3, clients, mediators, and providers may include operations in order to make the binding definitions flexible and, more importantly, to enable referencing particular client or provider subcomponents. Thus, in addition to the functions mentioned above, functions for processing binding operations will be required. The algorithm for inferring bindings, presented in pseudocode below, includes all the functions mentioned.

Function 4-6: *inferBinding*(ci, M_S): Infer binding for a client

Input: Client ci (node in instantiation I) for which binding is to be inferred
Set of mediators M_S that should be used

Output: bindingExpression (binding expression, i.e., the right-side expression),
isBindingPossible (is possible to infer binding including manual decisions?)
ci_a (actual client used, i.e., ci or one of its parents in I)
m_a actual mediator used
{p₀ ... p_n} set of providers referenced by m_c and contained in I

begin

// Set all to undefined or false.

01 isBindingPossible = **false**

02 bindingExpression, ci_a, m = **null** *// Set all to null (meaning they are undefined)*

03 dataCollection = **empty** *// Empty list*

// Get the instantiation (starting from root node) that contains ci.

04 I = *getInstantiation*(ci)

// Find the actual client (either ci itself or an upper level node in I). Note, that upper level client operations may overwrite bindings from lower levels.

05 {n₀ ... n_n=ci} = *getParentsTopDown*(ci) *// Get parents top-down including ci*

06 **for** n_k in {n₀ ... n_n} **do**

07 {m₀ ... m_n} = *getMediators*(n_k, M_S) *// Ordered list of mediators that reference n_k*

// For each n_k find those mediators with client references containing specifications for ci.

08 **for** m_k in {m_{k0} ... m_{kn}} **do**

09 nco_k = *getClientOperation*(m_k, n_k) *// Get client operation for n_k*

// If the left-side expression (placeholder replaced with instancePathOf(n_k)) is equal to instancePathOf(ci) -> then select. If there are there multiple matches, select only the topmost entry because this overwrites lower entries. If we reached ci then select ci to be the actual client.

10 **if** *specifiesBindingFor*(nco_k, instancePathOf(ci)) **or** n_k == ci **then**

// Check whether the client operation and client reference are valid. A client operation must contain the placeholder (e.g. providerPath) to be replaced by a concrete client instance path in I.

// If n_k is ci the client operation is discarded because it can only specify lower-level clients, which are not of interest at this point.

```
11    assert(isValidClientOperation(ncok)) -> abort otherwise "INVALID:
      Client operation ncok, attached to  $n_k$  specifying ci is not valid."
```

// Ensure that there is only one reference from mediator to client.

```
12    assert(getClientReferences(mk, nk) == 1) -> abort otherwise
      "INVALID: mk references client nk several times.)
```

// Continue if asserts are true.

```
13    cia = nk // Actual client found.
```

// Get providers (and their operations) that are referenced by m_k and contained in I.

```
14    {p0 ... pn}, {op0 ... opn} = getProviders(mk, I)
15    Omk = getMediatorOperation(mk)
```

// Mediator operation can be empty or can contain reduction function macros.

```
16    assert(isValidMediatorOperation(Omk, sizeof({p0 ... pn}))) -> abort otherwise
      "INVALID: Mediator operation Omk, of mk is not valid
      for inferring binding for ci."
```

// Continue if asserts are true.

// Add a row to dataCollection.

```
17    dataCollection.add(ci, cia, ocia, mk, omk, {{p0 ... pn}, {op0 ... opn}})
```

// It will be possible to infer binding assuming manual selection.

```
18    isBindingPossible = true
19    end if
20    end for // End of iteration over mediators.
21 end for // Top-down iteration looking for cia.
```

// Check that there is only one entry, i.e., only one valid mediator.

```
22 assert(sizeof(dataCollection) == 1) -> abort otherwise
    or ask for manual decision "Select a
      mediator from {...} for inferring binding for ci."
```

// Check whether the mediator operation can handle multiple providers or if providers should be selected manually.

```
23 if sizeof({p0 ... pn}) > 1 and
    not (isMultiProviderMediatorOperation(Oma)
      or preferredBindingExists(ci, {p0 ... pn})) then
```

```

24 abort or ask for manual decision "Select provider
    from {p0 ... pn} for inferring binding for ci."
25 optionally store the selection in preferred bindings
    by using instancePathOf(ci), modelQualifiedNamesOf(ci)
    and instancePathOf(pk), modelQualifiedNamesOf(pk)
26 end if

// Continue if assertions are true.
// If there is only one mediator, or it was selected manually -> the first row contains all
// relevant data.
27 cia, ocia, ma, oma, {{p0 ... pn},{op0 ... opn}} <- the first row in dataCollection

// A provider operation may be empty or must contain placeholder (e.g. providerPath)
// for future provider instance path otherwise.
28 assert(areAllValidProviderOperations({{p0 ... pn},{op0 ... opn}}) -> abort otherwise
    "Selected providers {...} have invalid operations {...}"

// If the actual client and the mediator are found and it is possible to infer binding.
29 assert(isDefined(cia) and isDefined(ma) and isBindingPossible) -> abort otherwise
    "No binding could be inferred for ci."

// Continue if assertions are true.
// At this point we have found the actual client, mediator, providers, and all operations.
// Now we can generate the binding expression.

// In all provider operations, replace the placeholder with the concrete provider
// instance path in I.
30 {otp0 ... otpn} = translateProviderOperations({p0 ... pn})

// Expand reduction function macros in the mediator operation.
31 otm = translateMediatorOperation(mk, {otp0 ... otpn})

// Replace the placeholder with the client instance path in I; replace the placeholder
// with the inferred binding expression.
32 bindingExpression = translateClientOperation(cia, ocia, otm)

33 return {bindingExpression, isBindingPossible, cia, ma, {p0 ... pn}}
end

```

Any **abort** in the algorithm above means that the function returns the current status of the outputs. Functions such as `isValidClientOperation(...)` test the validity of the operation as described in Section 4.2.4. The `preferredBindingExists(...)` function determines whether, in the given set of providers, there is one provider that should be used for the given client.

Note that this algorithm returns providers that are referenced by the selected mediator. However, these providers may contain operations involving subcomponents. For an analysis that requires the actual providers (e.g., those described in Section 4.3.4 or 4.3.5) it will be necessary to parse each provider operation in order to extract the actual providers.

4.8 Conclusion on Verification Results

The way requirement violation monitors are formalized in vVDR allows them to be used for online monitoring in which the requirement violation monitor status indicates, at any simulated time instant, the violated status and can change its state throughout the entire execution.

Consequently, the very last value for requirement violation monitor status is not the one to conclude on. When the analysis (e.g. simulation) is completed, traces of the violation monitors status have to be analyzed in order to be able to conclude whether requirements were *evaluated* or *not evaluated*, and *violated* or *not violated*. For one design verification run, in which the same requirement may be evaluated in multiple scenarios, conclusion on violation of requirements can be drawn automatically by implementing checks based on the following definitions.

Definition 23. Evaluated Requirement (partial evidence): Requirement r is said to be *evaluated* for design d using scenario s if in the corresponding analyzed verification model the r violation monitor status changed from the *not evaluated* state to any other state for at least one simulated time instant.

Definition 24. Evaluated Requirement (complete evidence): Requirement r is said to be *evaluated* for design d if in all analyzed verification models the r violation monitor status changed from *not evaluated* state to any another state for at least one simulated time instant.

Definition 25. Violated Requirement (partial evidence): Requirement r is said to be *violated* by design d using scenario s if in the corresponding analyzed verification model the r violation monitor status was *violated* for at least one simulated time instant.

Definition 26. Violated Requirement (complete evidence): Requirement r is said to be *violated* by design d if there exists at least one evidence for violation⁶⁵,

⁶⁵ Adopted from (Leucker & Schallhart, 2009), where similar explanation was provided for safety properties.

i.e., if in at least one of the analyzed verification models the r violation monitor status was *violated* for at least one simulated time instant.

Definition 27. Not Violated Requirement (partial evidence): Requirement r is said to be *not violated* in design d using scenario s if in the corresponding analyzed verification model the r violation monitor status changed from *not evaluated* state for at least one simulated time instant and was not in *violated* state for any of the simulated time instants.

Definition 28. Not Violated Requirement (complete evidence): Requirement r is said to be *not violated* in design d if in all analyzed verification models the r violation monitor status changed from *not evaluated* state for at least one simulated time instant and was not in *violated* state for none of the simulated time instant.

The confidence in *complete* evidence depends on the set of scenarios used, as well as on which requirements were combined with which scenarios. The term *complete* is not to be confused with evidence provided by formal verification techniques, such as model checking. Evidence provided by model checking includes the answer for all requirements and any scenario captured.

In the approach presented in this dissertation, there is a possibility that not all requirements will be evaluated in each scenario. The term *complete evidence* means that the conclusion is drawn not based on one scenario, but based on all scenarios in which a requirement was evaluated.

4.9 Verification Session Report Generation

New knowledge is generated when particular conclusions are drawn from the verification session results. The way this new knowledge is organized, stored, and made accessible is crucial to allow it to be reused and presented for different purposes. Semantic Web (W3C, 2013) aims at organizing data using standardized data formats, such as (W3C, 2013), in order to enable effective reuse of, and reasoning about, captured knowledge. Many API implementation and reasoners are available (W3C, 2013).

In the ModelicaML prototype developed in this dissertation, the results of verification sessions are stored in a XML (W3C XML, 2013) file after each verification session (i.e., batch mode execution of several simulation models and results post-processing). In addition, the XML file is translated into an HTML report file using (W3C, 2013).

Figure 11 shows an example of such an HTML report. In order to enable different predefined views (e.g. “Violated Requirements”, partial evidence; see

Figure 11) some data is stored redundantly in the generated XML and HTML report files.

The screenshot displays a 'Verification Session Report' with the following sections:

- Verification Report**: Creation date: 2012.11.22 11:30:36.
 - Violated Requirements (3 of 0)
 - Not Violated Requirements (3 of 0)
 - Not Evaluated Requirements (1 of 0)
 - Not Implemented Requirements (0 of 0)
 - Used Scenarios (8 of 0)
 - Not Used Scenarios (0 of 0)
 - New Positive Relations (39)
 - New Negative Relations (9)
 - All Verification Models (8)
 - Simulated Models (8 of 8)
 - Not Simulated Models (0 of 8)
- Violated Requirements**:
 - Requirements that were evaluated and violated.
 - Metric: 3 of 0
 - 001: Volume of a tank**
 - 003: Max level of liquid in tank**
 - 004: Settling time and bounds after a change of input flow**

TwoTanksExample:Requirements:Settling time and bounds after a change of input flow
Text: After each change of the tank input flow, the controller shall, within 20 sec., ensure that the level of liquid in tank is level with a tolerance of +/- 0.15 m.
- Simulated in the following verification models:**
 - VeM: aVeM: Change of input flow
- System Model**: sm_systemenvironment
 - Client: twoTanksSystem.controller1.ref = vs_change_of_input_flow.refForController1
 - Client: twoTanksSystem.controller2.ref = vs_change_of_input_flow.refForController2
 - Client: twoTanksSystem.source.flowLevel = vs_change_of_input_flow.inputFlowLevel
 - Client: twoTanksSystem.tank1.levelOfLiquid.start =
 - Client: twoTanksSystem.tank2.levelOfLiquid.start =
- Scenario**: vs_change_of_input_flow
 - TwoTanksExample:Scenarios:Change of input flow
 - (1) Requirement (004): req_004_0_settling_time_and_bounds_after_a_change_of_input_flow**

TwoTanksExample:Requirements:Settling time and bounds after a change of input flow
 Client: inputLevelChanged = abs(sm_systemenvironment.twoTanksSystem.source.flowLevel - delay(sm_systemenvironment.twoTanksSystem.source.flowLevel, 0.1)) > 0.001
 Client: levelInTank = sm_systemenvironment.twoTanksSystem.tank1.levelOfLiquid
 Client: refLevel = sm_systemenvironment.twoTanksSystem.controller1.ref
 - (1) Requirement (004): req_004_1_settling_time_and_bounds_after_a_change_of_input_flow**

TwoTanksExample:Requirements:Settling time and bounds after a change of input flow
 Client: inputLevelChanged = abs(sm_systemenvironment.twoTanksSystem.source.flowLevel - delay(sm_systemenvironment.twoTanksSystem.source.flowLevel, 0.1)) > 0.001

Figure 11: Example of verification session report generated

This approach is advantageous because with the preprocessed and preformatted data, neither a reasoning nor a rendering engine is required to create or present the report. An HTML file can be viewed in any web browser. On the other hand the generated files have rather large file-size, a disadvantage. Moreover, the ModelicaML XML schema for verification results is proprietary and no off-the-shelf reasoning or rendering engines can be invoked on the data straightaway. Finally, the new knowledge generated in verification sessions is not shared, i.e., it is not made accessible in repositories for other engineers.

The following approach will improve the situation. It is based on semantic web technologies (W3C, 2013) and proposes an ontology for capturing verification results in an extensible and standardized fashion. The basic idea is the following:

- Information about the verification models and analysis results generated should be stored in a new ontology in the OWL format.

- No redundant data should be stored if it can be inferred by reasoning. The ontology should include the necessary logic to infer knowledge when needed instead of storing it redundantly.
- It should be possible to query the ontology inferred in order to enable reusing of knowledge.

Classes from the proposed vVDR verification ontology (see Appendix C for the definition of ontology) are shown in Figure 12 using the *protégé* tool (Stanford Center for Biomedical Informatics Research, 2013). The classes *Model* and *Evaluation* reflect the fact that models are analyzed (e.g. simulated) and conclusions drawn. Object properties such as *consideredDesign* or *consideredRequirement* relate these concepts.

Figure 12: vVDR verification results ontology

Some object properties are defined to be the inverse of other properties. For example, Figure 13 shows that the object property *evaluated* is an inverse of the property *consideredRequirement*. Such *inverse of* axioms enable the knowledge to be inferred by an OWL reasoner (W3C, 2013) instead of storing data redundantly. Data properties are shown in Figure 14.

Figure 13: vVDR verification results ontology: Inverse object properties

Figure 14: vVDR verification results ontology: Data properties

After each verification session, OWL files (that import the ontology definition as presented in Appendix C) can be generated automatically.

The OWL files generated can then be used by a report engine⁶⁶ that will access data and include invocation of reasoners for inferring data (such as *inverse of* relations; see an example in Figure 15 and Figure 16) and rendering engine for presenting the report. An example of SPARQL queries (W3C, 2013) for different views of the inferred ontology are listed in Appendix C.

⁶⁶ Such a report engine is not prototyped in this thesis.

Figure 15: vVDR ontology with individuals

Figure 16: vVDR ontology with inferred data

For example, a query to retrieve violated requirements and corresponding designs and scenarios, and a result⁶⁷ from such a query, may look like this:

```
SELECT ?reqName ?designName ?scenarioName

WHERE {
  {
    #individuals
    ?requirement rdf:type vvdr:Requirement.
    ?requirement vvdr:evaluated ?evaluation.
    ?evaluation rdf:type vvdr:ViolatedRequirement.
    ?evaluation vvdr:consideredDesign ?design.
    ?evaluation vvdr:consideredScenario ?scenario.
  }

  #names
  ?requirement vvdr:name ?reqName.
  ?design vvdr:name ?designName.
  ?scenario vvdr:name ?scenarioName.
}
```

⁶⁷ Twinkle (Dodds, 2013) was used for testing SPARQL queries.

reqName	designName	scenarioName
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_5 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Draining tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_4 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_1 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_6 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_3 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Volume of a tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_2 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_1 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_6 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_3 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_4 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_2 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_3 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_5 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_1 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_2 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_4 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_5 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_4 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_4 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_3 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_6 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_5 ⁶⁸ http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank ⁶⁸ http://www.w3.org/2001/XMLSchema#string	SystemEnvironment ⁶⁸ http://www.w3.org/2001/XMLSchema#string	Change of input flow_5 ⁶⁸ http://www.w3.org/2001/XMLSchema#string

Figure 17: SPARQL query result for "Violated Requirements" view

The advantages of the proposed approach are the following:

- The file size of the report generated can be much smaller compared to the old report (XML and HTML report files) because no data will be stored redundantly. The required data can now be inferred and queried when needed.
- Relations between scenarios and requirements (that indicate which scenarios are appropriate to verify which requirements) are currently defined manually⁶⁸ by the tester. Optionally, the tester may use the relation discovery feature as proposed in Section 4.3.2, which requires execution of an analysis (e.g. simulations) and may be time consuming. With the approach presented above this knowledge can be shared between engineers and can aid in the discovery of relations without having to perform simulations.
- Querying of knowledge about relations between scenarios and requirements may be part of the algorithm presented in Section 4.1.

Knowledge management is outside the scope of this dissertation. Consequently, no effort was spent on implementing a new verification report engine to replace the existing ModelicaML HTML report generator. A new report engine would use the OWL files generated, invoke a reasoned, and process query results to present particular views on verification results.

⁶⁸ This task may be supported by a relation discovery feature as proposed in Section 4.3.2.

4.10 Conclusion

This chapter presents ideas concerning the automation of the model composition and verification results evaluation (first identified in Section 3.8). It presents the new bindings concept to define a loose coupling of models. This new concept can be used to automate the model composition (see algorithms in Section 4.1 and 4.7), determine traceability between artifacts, discover relations between scenarios and requirements, etc. (see Section 4.3).

The proposed method for requirement formalization supports the automation of drawing conclusions based on generated analysis results (see Section 4.8). The new knowledge that is generated when concluding the generated results can be shared using semantic web technologies, as proposed in Section 4.9.

Chapter 5

Language and Tool⁶⁹

5.1 Introduction

This chapter presents new language called ModelicaML – a UML profile for Modelica. The main purpose of ModelicaML is to support the vVDR method (see Chapter 3) and to enable reading or understanding, and maintaining simulation models by using standardized graphical notations and executable semantics.

The main portion of the ModelicaML concrete syntax is defined as graphical notation that facilitates different views (composition, inheritance, behavior) on system models. It is based on a subset of the OMG Unified Modeling Language (OMG UML, 2011). Since the ModelicaML profile is an extension of the UML meta-model, it can be used for modeling both with standard UML and with UML-based languages such as SysML⁷⁰. Parts of the model are entered as text (i.e., Modelica equations or algorithmic code, modification and declaration expressions).

UML provides the modeler with powerful descriptive constructs at the expense of loosely defined semantics that are marked as “semantic variation points” in the UML specification. The intention of ModelicaML is to provide the modeler with powerful executable constructs and precise execution semantics that are based on the Modelica language.

Therefore, ModelicaML is based on a limited part of the UML meta-model and extends it using the UML profiling mechanism (see Appendix A for the complete ModelicaML profile description), offering new constructs to allow capturing of Modelica concepts that are not provided by UML and to support all concepts required for the vVDR method. However, like UML or SysML, ModelicaML is a

⁶⁹ The text in this chapter incorporates and updates text from (Schamai, Fritzson, Paredis, & Pop, 2009).

⁷⁰ SysML itself is also a UML profile. All stereotypes that extend UML meta-classes are also applicable to the corresponding SysML elements.

graphical notation. ModelicaML models are eventually translated into Modelica code. Hence, the execution semantics are defined by the Modelica language and ultimately by a Modelica compiler that will translate the generated Modelica code into an executable form.

In the course of this dissertation ModelicaML was also used to prototype ideas and contribute to the SyM standard (OMG SyM, 2012), (Paredis, et al., 2010) that defines a bidirectional transformation between SysML and Modelica.

The sections below contributes to the Research Question 4 (see Section 1.3) about required characterizes of a language that enables modeling and simulation of complex physical systems and the verification of system designs against requirements.

5.2 Related Work

In previous work, researchers have already identified the need to integrate UML/SysML and Modelica, and have partially implemented such an integration. In (Pop, Akhvlediani, & Fritzson, 2007) the basic mapping of the structural constructs from Modelica to SysML is identified.

The SysML Parametrics concept (essentially the same as equations) is used for the integration of continuous-time behavior into SysML models (Johnson, Kerzhner, Paredis, & Burkhart, 2012). A concept to use SysML to integrate models of continuous-time dynamic system behavior with SysML information models representing systems engineering problems is introduced in (Johnson, Kerzhner, Paredis, & Burkhart, 2012).

Rules for the graph-based bidirectional transformation of subsets of SysML and Modelica models are provided in (Johnson, Kerzhner, Paredis, & Burkhart, 2012) as well. Finally, the recent OMG standard SyM (OMG SyM, 2012) defines a bidirectional translation of SysML and Modelica models.

5.3 UML-to-Modelica Mapping

The class concept is the basic structural unit in Modelica. Classes provide the structure for objects and contain equations, which ultimately serve as the basis for the executable simulation code.

The most general kind of class is a *class* or *model*. Specialized categories of classes—such as record, type, block, package, function and connector—have most of the properties of a model but with some restrictions or enhancements. In UML the class is the main structural unit which a behavior can have. It is natural to map the Modelica class concept to the UML class concept.

The complete definition of the ModelicaML profile is provided in Appendix A. Section 5.5 presents the set of UML diagrams reused in ModelicaML and Appendix A.13 proposes enhancements of the UML graphical notation. Section 5.6 discussed the handling of semantic issues when translating UML state machines into Modelica code.

5.4 vVDR Support

ModelicaML supports the vVDR method by introducing vVDR concepts such as requirement, verification scenario, mediator, etc. Furthermore, for the definition of bindings an extension of the Modelica language is proposed (see Appendix B for its grammar). The implementation of the algorithms proposed in Section 4.1 and 4.7 uses the following definitions to collect required data from models:

- *isUMLClass*(*e*): returns true if the element *e* is a UML::Class; false otherwise.
- *hasStereotype*(*e*, *s*): returns true if the element *e* has the stereotype *s* applied; false otherwise.
- **Mediator Container**: UML::Class with stereotype «MediatorContainer» applied:
 - $isMediatorContainer(e) \leftarrow isUMLClass(e) \wedge hasStereotype(e, "MediatorContainer")$
- *ownerIsMediatorContainer*(*e*): returns true if the element *e* is a UML::Class and is a *Mediator Container*; false otherwise.
- **Mediator**: UML::Property with stereotype «ValueMediator» applied and owned by a *Mediator Container*:
 - $isMediator(e) \leftarrow ownerIsMediatorContainer(e) \wedge hasStereotype(e, "Mediator")$
- **Client Dependency**: UML::Dependency with stereotype «ProvidesValueFor», *Mediator* as source and UML::Property as target.
- **Provider Dependency**: UML::Dependency with stereotype «ObtainsValueFrom», *Mediator* as source and UML::Property as target.
- **Client**: UML::Property that is target of a *Client Dependency*.
- **Mandatory Client**: UML::Property that is target of a *Client Dependency* with the attribute *isMandatory* set to true.
- **Provider**: a UML::Property that is target of a *Provider Dependency*.

5.5 Graphical Modeling with ModelicaML

It is the belief of the author that in the medium-term, information such as requirements or designs of complex systems will be captured using models instead of manually written natural-language documents. However, nowadays documents are the main means for communicating and signing contracts, and they will probably still play an important role in system development in the near future. For this reason, much attention should be paid to aspect of displaying model data in diagrams or tables so as to be readable on screens and on printed paper.

When building models all model data is stored in a repository (files or databases). Diagrams are only a view of a selected part of the model data (which is typically a huge quantity of structured information). The use of graphical notation to visualize model data correctly and appropriately by means of diagrams creates trade-offs between several factors:

- Let the size of a diagram grow (e.g. beyond the limits of the screen or paper format) or split the information into multiple diagrams? Splitting information into multiple diagrams will allow only a partial view on information in each diagram and may make it difficult to understand the interrelations between elements.
- Show all information or filter the information display on a diagram? Overloaded diagrams complicate the perception of information. A filtered view of information is clearly more appropriate as long as the reader is aware that it is a partial view and can adjust the view filter.
- Simple and abstracted notation or detailed domain-specific icons? UML-like notations (i.e., boxes with squared or rounded corners, lines or arrows, etc.) are simple but not expressive. The graphical representation of the element is often not enough to know what it actually represents. In contrast, domain-specific icons are very expressive and tell the reader much more without having to know the name, type, and more context-sensitive information. However, such notations are best suited for domain experts.
- Automated or free diagram layout? (This includes the decision as to how elements appear on diagrams.) Modeling becomes time consuming when the user has to spend time arranging elements in diagrams. Approaches and implementations for automated arrangement of elements on diagrams have been available for at least 15 years (Seemann, 1997) and have made considerable progress (Fuhrmann & von Hanxleden, 2010). Automated layouts and element naming or coloring can greatly contribute to leveraging the advantages of using unified notations for communication between people. On the other hand, diagrams that model graphs—such as UML composite, state machine, or activity diagram—are better arranged manually.

UML addresses the challenges listed above. It allows the options above to be balanced against one another in order to achieve appropriate visualization of information. For example, it is possible to create multiple diagrams and to show only selected content (e.g., only the inheritance or containment of classes, only class decomposition, etc., or any filtered view of these). It is possible to add graphical icons for classes to be displayed on diagrams. There are several types of diagrams in UML, each focusing on a particular kind of content (e.g. class diagram for class definition information, composite diagram for architectural view, behavior diagrams such as state machine or activity diagram for class behavior).

In Modelica it has until recently only been possible to create one diagram for a class, namely the connection diagram. This diagram shows components and their interconnection. It corresponds to a UML composite diagram. In Modelica, content cannot be split into multiple diagrams. A second kind of diagram, the state machine diagram, has become available in Modelica since version 3.3.

From a language perspective, the integration of the UML graphical notation for representing models and the simulation power of the Modelica language has a strong motivation. From the modeling tool perspective, the goal of minimizing modeling effort is a strong motivation. The ModelicaML prototype contributes to this, for example, by providing action code editing support with syntax highlighting and code completion, as well as model validation features. Moreover, it is based on the (The Eclipse Foundation, 2013) framework, which provides features that can be reused and a mechanism for implementing one's own extensions.

The subset of UML diagrams that is used for modeling with ModelicaML is listed below (see also Figure 18):

- The *class diagram* (based on the UML class diagram) shows relevant information regarding the definition of the class concept and inheritance and import relations to other classes. In ModelicaML it is used to represent model definition information and requirements.
- The *connection diagram* (based on the UML composite structure diagram) shows class components and their interconnection. In ModelicaML this diagram is used to represent the internal structure of system components.
- The *state machine diagram* (based on the UML state machine diagram) shows states and transitions of a class behavior. In ModelicaML this diagram is used to model state-based behavior of components, requirement violation monitors and scenarios.
- The *activity diagram* (based on the UML activity diagram) shows conditional equations or algorithm statements. In ModelicaML this diagram is used to model conditional equations or statements.

Figure 18: ModelicaML diagrams

5.6 Model Transformation Implementation

An important goal of the ModelicaML prototype implementation is to enable simulation of UML-based models. This is achieved by translating models defined in UML to Modelica. The following sections explain how the bidirectional transformation between UML and Modelica is implemented in ModelicaML.

5.6.1 ModelicaML to Modelica

Figure 19 gives an overview of the technologies that were used for ModelicaML-to-Modelica code generation. A template-based language (The Eclipse Foundation - Acceleo Project, 2013) is used for the translation, following (OMG MOFM2T, 2013). In addition, Eclipse EMF (The Eclipse Foundation - EMF Project, 2013) is used for implementing the ModelicaML validator that checks model consistency rules.

The ModelicaML-to-Modelica transformation allows engineers to create models graphically using UML notation and to translate them into Modelica code for simulation.

Figure 19: ModelicaML prototype technology

5.6.2 Modelica to ModelicaML

In order to enable the use of existing Modelica models in ModelicaML, an import and synchronization module was implemented in ModelicaML. After having been initially imported, the Modelica code can be kept synchronized with the ModelicaML representation each time the Modelica models change.

5.7 Resolution of UML State Machines Semantics Issues⁷¹

Graphical modeling, as promoted by the OMG (OMG, 2013), promises to be more effective and efficient regarding editing, human-reader perception of models, and maintaining models compared to traditional textual representations.

The usage of a unified standardized graphical notation—for example, one based on the UML standard—in systems modeling and simulation applications facilitates the common understanding of models for all parties involved in the development of systems (i.e., systems engineers, designers, testers; software developers, and customers or other stakeholders).

⁷¹ The text in this section incorporates and updates text from (Schamai, Fritzson, & Paredis, 2013).

By integrating UML and Modelica, UML-based modeling becomes applicable to the physical-system modeling domain as well, and UML models become executable.

This covers simulation of hardware and software, with integrated continuous-time and event-based discrete-time behavior. In order to achieve this goal ModelicaML is designed to have a precise and clear execution semantics and resolves semantic variation points or ambiguities of the UML specification.

The statechart (Harel, *A Visual Formalism for Complex Systems*, 1987), (Harel & Kugler, 2004) and UML (OMG UML, 2011) state machine formalisms have been used extensively during the last few decades and have proved to be a powerful tool for describing state-based behavior of components. For example:

- State machines are used to specify the required system component behavior by describing its states or modes and transitions between them. In this case state machines represent the required functional behavior of a component, in contrast to the predicted real physical behavior of a system component, which is typically described using differential algebraic equations (DAEs).
- State machines are also used to create simulation models that represent an abstracted or simplified behavior of a physical component with discrete-time or continuous-time behavior. The level of the abstraction or simplification depends on the analysis needs and requirements for simulation performance.
- In software development, state machines are used for analysis (e.g. by simulation), as well as for the implementation of reactive software and the generation of code that is deployed.

Hence, the state machine formalism seems to be appropriate for use in describing both the reactive (i.e., event-based) behavior of software components and the states and modes of physical system components with continuous-time behavior.

However, from a system-simulation perspective, the behavior described in the UML state machine is typically translated into discrete-time or reactive (event-based) simulation models and is thereby limited by this modeling formalism.

In contrast, Modelica enables mathematical modeling and simulation of hybrid (integrated continuous-time, discrete-time and event-based dynamic) behavior and also enables adequate modeling and simulation of physical phenomena. Modelica version 3.3 also includes synchronous language elements with clocked semantics “in order to define sampled data systems in a safe way” (Modelica, 2013, p. 181).

The implementation of UML state machines in ModelicaML is an attempt to adopt this formalism for system modeling and simulation by:

- Providing a clear and unambiguous semantics definition for state machines that can be translated into executable form. This implies that all ambiguities,

semantic variations points, as well as missing semantics definitions in the UML specification need to be resolved.

- Integration of state machines with a language that enables modeling and simulation of continuous-time dynamic behavior of components in order to enable the integrated simulation of physical components and software. This addresses the need for modeling and simulating cyber-physical systems that are composed of hardware and software. For ModelicaML this means that state machines are translated into Modelica code.

This chapter addresses the fundamental need for formal modeling of system behavior by combining state machines and continuous-time differential-algebraic equations.

By transforming UML state machines into Modelica, both the discrete-time behavior of software components and the continuous-time behavior of physical components can be modeled and simulated efficiently within a single modeling framework: Modelica.

Design decisions made are not specific to the translation of UML state machines into Modelica code. The questions addressed in this chapter will most likely be raised by anyone who intends to generate executable code from UML state machines.

5.7.1 Simple Example

Assume one would like to simulate the behavior defined by the state machines depicted in Figure 20 using Modelica. This state machine defines part of the behavior of the class `SimpleStateMachine`. In UML, this class is referred to as the *context* of `StateMachine_0`. The class `SimpleStateMachine` has two attributes: `x` of type `ModelicaInteger` and `t` of type `ModelicaReal`. The value of `t` is calculated continuously based on the equation $\text{der}(t) = \text{time}^{72}$ (see code in Figure 21). The initial value of `x` is 0.

The UML graphical notation for state machines consists of rectangles with rounded corners representing states, and edges representing transitions between states. Transitions can only be executed if appropriate triggers occur (e.g. “all” trigger which denotes the acceptance of any event that occurs) and if the associated guard condition (e.g., $[t > 1 \text{ and } x < 3]$) evaluates to true. In addition, transitions can have effects (e.g., $/x := 1$).

⁷² The function `der(x)` in Modelica represents the first time-derivative of the variable `x`.

The UML semantics define that a state machine can only be in one of the (simple) states in a region⁷³ at the same point in time. The filled circle and its outgoing transition mean that, by default, (i.e., when the execution is started) the state machine is in its initial state, *State_0*. The expected execution behavior is as follows:

- When the execution is started the state machine is in its initial state, *State_0*.
- As soon as the guard condition, $[t > 1 \text{ and } x < 3]$, is true, *State_0* is exited, the transition effect, $x := 1$, is executed and *State_1* is entered. The state machine is again in a stable configuration that is referred to as an active configuration.
- As soon as the guard condition, $t > 1.5 \text{ and } x > 0$, becomes true, *State_1* is exited, the effect, $x := 2$, is executed and *State_2* is entered.
- As soon as the condition $x > 1$ becomes true, *State_2* is exited, the transition effect, $x := 3$, is executed and *State_0* is entered.

Figure 21 shows Modelica code that performs the behavior described above.

Figure 20: Example of a simple state machine owned by a class

⁷³ If a composite state or multiple regions are defined for a state machine, then the state machine is in multiple states at the same time.


```

model SimpleStateMachine
  Boolean State_0 "State_0 representation";
  Boolean State_1 "State_1 representation";
  Boolean State_2 "State_2 representation";
  Integer x "discrete variable";
  Real t "continuous variable";
  equation //Code for continuous integration of t
    der(t)=time;

  algorithm //Code for StateMachine_0
  when initial() then
    State_0:=true "Activation of the initial state";
  end when;
  //Transition from State_0 to State_1
  if pre(State_0) and t > 1 and x < 3 then
    State_0:=false "Deactivation of state";
    x:=1 "Transistion effect";
    State_1:=true "Activation of state";
  //Transition from State_1 to State_2
  elseif pre(State_1) and t > 1.5 and x > 0 then
    State_1:=false "Deactivation of state";
    x:=2 "Transistion effect";
    State_2:=true "Activation of state";
  //Transition from State_2 to State_0
  elseif pre(State_2) and x > 1 then
    State_2:=false "Deactivation of state";
    x:=3 "Transistion effect";
    State_0:=true "Activation of state";
  end if;
end SimpleStateMachine;

```

Figure 21: Modelica code for the simple state machine

5.7.2 State Machines in ModelicaML

State machines are used in ModelicaML to model parts of class behavior. A behavioral class (i.e., Modelica `class`, `model`, or `block`) can have multiple state machines along with other behavior definitions (e.g. equation or algorithm sections). This is different from a typical UML application where usually only one state machine is used to represent the *classifierBehavior*.

In ModelicaML it is possible to define multiple state machines for one class. All state machines of a class are executed in parallel. This possibility allows the modeler to structure mutually exclusive behavior by separating it into individual state machines. This is equivalent to using regions in one state machine for modeling orthogonal behavior.

UML state machines are typically used to model the reactive (event-based) behavior of objects. Usually an event queue is implemented that collects all previously generated events, which are then dispatched one after another (the order is not fully specified by UML) to the state machine and may cause state machine reactions.

Hence, a UML state machine reacts (is evaluated) only when events are taken from the event queue and are dispatched to the state machine. ModelicaML uses Modelica as the execution (action) language. In contrast to the typical

implementations of UML state machines mentioned above, the Modelica code for a ModelicaML state machine is evaluated continuously, namely, after each continuous-time integration step if the state machine contains guard-generating events based on the change of variable values.

The concept of event iterations is defined in Modelica (Modelica, 2013, p. 30) as follows: “A new event is triggered if at least for one variable v “ $\text{pre}(v) \triangleleft v$ ” after the active model equations are evaluated at an event instant. In this case the model is at once reevaluated. This evaluation sequence is called “event iteration”. The integration is restarted, if for all v used in pre-operators the following condition holds: “ $\text{pre}(v) == v$ ”.”

The definition of $\text{pre}(v)$ is the following: “Returns the “left limit” $y(t^-)$ of variable $y(t)$ at a time instant t . At an event instant, $y(t^-)$ is the value of y after the last event iteration at time instant t ” (Modelica, 2013, p. 29). Furthermore, the following definition is essential for understanding the execution semantics of Modelica: “Modelica is based on the synchronous data flow principle and the single assignment rule, which are defined in the following way:

- All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants.
- At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently (equations are not active if the corresponding if-branch, when-clause or block in which the equation is present is not active).
- Computation and communication at an event instant does not take time. [If computation or communication time has to be simulated, this property has to be explicitly modeled].
- The total number of equations is identical to the total number of unknown variables (= single assignment rule).” (Modelica, 2013, p. 88)

Moreover, Modelica provides predefined event-related operations, such as `edge()` or `change()`, etc., which allow the expression of conditions on the occurrence of an event. These predefined operations can be used in transition guard code. This way it is possible to use only guards with expressions and Modelica event-related operations for enabling transitions, with no need for explicit triggers.

In ModelicaML the following design decision was made: If no trigger is defined for a transition, then the transition reacts to any event (e.g. an event that was generated because the variable that is referenced in the transition guard changed its value). The enabling of the transition then only depends on its guard expression at this time instant. An empty guard implies the evaluation to “true” by default. These assumptions are common and are implemented in UML tools, for example, such as

Rhapsody (IBM, 2013), which generate executable code from state machines. The reader should keep this in mind when reading the state machines diagrams below.

5.7.2.1 Transformation of State Machines to Modelica Code

There are multiple ways to enable the execution of state machines using Modelica. The first question is whether a Modelica state machine library should be used or whether a code generator should be implemented.

A library would contain basic state machine concepts such as state machine, state, transition, region, etc., which would be used by modeler and parameterized as needed. The advantage of using a library is that the execution semantics can be understood simply by inspecting the library classes. In order to understand the execution semantics of generated code one also needs to understand the code generation rules.

The second fundamental question concerns the representation of state machine behavior: it can be expressed by using Modelica algorithmic code or by using equations. Note that the statements inside an algorithm section in Modelica are executed exactly in the sequence they are defined.

This is different from the equation sections, which are declarative, so that the order of equations is independent from the order of their evaluation. For example, the Modelica StateGraph library (Otter, Malmheden, Elmqvist, Mattsson, & Johnsson, 2009) uses equations to express behavior that is similar to a subset of the UML state machine formalism.

In ModelicaML the behavior of a state machine is translated into algorithmic Modelica code that is generated into one algorithm section of the containing class⁷⁴. The rationale for the decision to implement a specific code generator instead of implementing a library and to use algorithm statements instead of equations is the following:

- A library has components with predefined elements, such as interfaces, configurable parameters, etc. This approach is difficult to apply to states, which potentially always have different number of state-outgoing transitions with an individual transition guard specification. The code generation approach is more flexible and appropriate for that.
- The behavior expressed by a state machine is always causal: i.e., the sequence of deactivation and activation of states and the execution of associated actions always define the input/output relation. There is no need to use the acausal equation-based modeling capability of Modelica. By using an algorithm (with predefined causality) no sorting of equations is required.

⁷⁴ A Modelica class can have multiple algorithm sections.

- For the implementation of inter-level transitions, i.e., transitions which cross states' hierarchy borders, the deactivation and activation of states and the execution sequence of associated actions (exit/entry action of states or state transitions effects) has to be performed in an explicitly defined order. This is hard to achieve when using equations that are sorted based on their data dependencies.

It is interesting to note that, because of the disadvantages of the library approach, some of the authors of the previous equation-based state machine libraries (Malmheden, Elmqvist, Mattsson, Henriksson, & Otter, 2008) eventually took the initiative to introduce built-in clocked state machines (Elmqvist, Gaucher, Mattsson, & Dupont, 2012) into the Modelica language from version Modelica 3.3 (Modelica, 2013). The implementation of these built-in state machines is based on code generation inside the Modelica compiler.

5.7.2.2 Interrelation with Other Class Behavior Definitions

For one class, multiple parts of its behavior can be defined by using state machines or UML *OpaqueBehavior* which contain Modelica code inside.

For example, a class can have multiple defined state machines, multiple algorithm sections (containing pure Modelica code or resulting from code generation from a conditional algorithm that is modeled using a subset of the UML *Activity Diagram* notation) or multiple equation sections (also pure Modelica code or code generated from conditional equations modeled using a subset of the UML *Activity Diagram* notation).

All sections and state machines are mutually exclusive: i.e., they are disjoint and cannot set the same variables (see the *single assignment rule* of Modelica in (Modelica, 2013, p. 88). However, all sections can read any class variable value, as well as the active configuration of state machines. This allows activation and deactivation or algorithmic code or equations⁷⁵ based on the status of state machines defined for a class.

5.7.2.3 Combining Continuous-Time, Event-Based or Discrete-Time Behavior

The fact that a ModelicaML state machine is translated into algorithmic Modelica code implies that all actions (transition effects or entry/do/exit actions of states) can only be defined using algorithmic code. Hence, it is not possible to insert equations into transition effects or entry/do/exit actions of states directly.

⁷⁵ Equations are primarily used for continuous-time behavior modeling.

However, it is possible to activate or deactivate a set of equations based on the active state machine configuration, as illustrated in Figure 24. Also note that a Modelica algorithm section as an aggregate construct behaves like a declarative single-assignment equation that assigns the variables defined by the algorithm section. This enables algorithm sections to be freely mixed with equation sections in Modelica models.

Consider the state machine defined for a tank. Depending on the level of liquid in the tank (represented by the variable h) the state of the tank can be *empty*, *partially filled*, or *overflowing*.

Figure 22: State Machine of the tank

The next state machine specifies the behavior of the controller. It shows that only if the controller is in the state *on* will it monitor or control the level of liquid in the tank depending on the sensor values received. Here, the state machine reacts to the status of a continuous-time variable, the value delivered by the sensor. Another example is the Modelica code in Figure 21 that shows that the state transitions read the continuous-time variable t that is calculated from an equation.

Figure 23: State Machine of the controller

Since state machines are translated into algorithmic Modelica code, any variable that is set within state machine actions must be *discrete-time* (i.e., any Real type

variable must be set to have discrete-time variability) (see Modelica 2013, 125, “Execution of an algorithm in a model”).

Other parts of the class behavior can be defined as being dependent on an explicit state or mode of the component. In this case, certain equations are active or not active depending on the active state machine configuration. For example, the following shows how conditional equations include the dependency to a state machine state.

Figure 24: Example of state-dependent equations

5.7.2.4 Event Processing (Run-To-Completion Semantics Applicability)

UML (OMG UML, 2011, p. 574) defines the run-to-completion semantics for processing events. When an event is dispatched to a state machine, the state machine must process all actions associated with the reaction to this event before reacting to further events (which might possibly be generated by the transitions taken). This definition implies that, even if events occur simultaneously, they are still processed sequentially.

In practice, this is solved by implementing an event queue (also referred to as “event pool”) that ultimately prevents events from being processed in parallel. However, this can lead to ambiguous execution semantics, as is pointed out in Section 5.7.3.2.

The problem is avoided in ModelicaML. If events occur simultaneously (at the same simulated time instant or event iteration) in ModelicaML state machines, they are processed (i.e., consumed) in parallel in the next evaluation of the state machine.

However, the definition of the run-to-completion semantics is still applicable to Modelica and, thus, to ModelicaML state machines, in the sense that when an event has occurred a state machine first finishes its reactions to this event before processing events that are generated during these reactions.

5.7.2.5 Subset of UML2 State Machine Concepts Supported in ModelicaML

UML defines two types of state machines: behavior state machines and protocol state machines. Behavior state machines are used to model parts of class behavior, and ModelicaML state machines are derived from UML behavior state machines.

Compared to behavior state machines, protocol state machines are limited in terms of expressiveness and are tailored to the need to express protocols or to define the lifecycle of objects. Since this is not the main intended area of application for ModelicaML, the protocol state machines are not taken into account. Consequently, none of the chapters of the UML specification that address the protocol state machines are considered.

5.7.2.6 Support of UML State Machine Graphical Notation

The graphical notation for ModelicaML state machines is based on the notation for UML behavior state machines. All graphical modeling constructs are supported except *SendSignalActions* or other actions (OMG UML 2011, 586). Instead the modeler can capture required behavior in the behavior bodies (e.g., entry/do/exit behavior of states).

5.7.2.7 Supported UML State Machine Concepts

ModelicaML state machines are based on a comprehensive subset of the UML state machine concepts that are considered by the author as being needed and typically used for control logic modeling. The UML state machine concepts⁷⁶ which are not supported in ModelicaML or which are supported in a different way compared to UML are the following:

- **15.3.2 FinalState (from BehaviorStateMachines):** Supported in ModelicaML. However, no explicit completion events are generated if regions of a composite state have reached the FinalStates (OMG UML, 2011, p. 583); see also comments for “Completion transitions and completion events” below). Furthermore, no termination of the context object (i.e., the class owning the state machine) is implied if all regions of a composite state reach their FinalStates (see also comments for terminate state below).
- **15.3.6 CallEvent (from Communications):**
 - *CallEvent* implies a call of a class operation. This is not applicable to Modelica, since Modelica does not support the class-method

⁷⁶ The chapter numbers correspond to the UML specification chapter numbers.

concept that can be found in object-oriented languages such as Java or C++.

- **15.3.8 Pseudostate** (from BehaviorStateMachines), **15.3.9 PseudostateKind** (from BehaviorStateMachines):
 - *DeepHistory* is not supported in ModelicaML. *History* can be used at different state hierarchy levels in order to express the same behavior as *DeepHistory*.
 - *Terminate*: Reaching a terminate state in ModelicaML does not imply termination of the context object (i.e., the class owning the state machine). Reaching a terminate state implies the deactivation of the state machine.
- **15.3.11 State** (from BehaviorStateMachines, ProtocolStateMachines):
 - *Deferred events* and *State redefinition* are not supported in ModelicaML. *Deferred events* are not supported because no event-queue is used. *State redefinition* is subject for future considerations.
- **15.3.12 StateMachine** (from BehaviorStateMachines):
 - Priority scheme for conflicting transitions that are at different hierarchy levels is different from UML. UML defines “[t]he priorities of conflicting transitions” as “based on their relative position in the state hierarchy. By definition, a transition originating from a sub state has higher priority than a conflicting transition originating from any of its containing states” (OMG UML, 2011, p. 576). The ModelicaML priority scheme for conflicting transitions is different (see Section 5.7.3.6).
 - *StateMachine extension* is not supported in ModelicaML.
- **15.3.14 Transition** (from BehaviorStateMachines):
 - *Completion transitions* and *completion events* as defined in UML (see p. 574) are not supported in ModelicaML. The reason is that ModelicaML uses different priority scheme for conflicting transitions (that are at different hierarchy levels) in state machines (as explained in Section 5.7.3.6). The same behavior can be expressed in ModelicaML using state dependency or joins.
 - *Deferred triggers* and *Transition redefinition* are not supported in ModelicaML. *Deferred triggers* are not supported because no event-queue is used. *Transition redefinition* is subject for future considerations.

5.7.3 State Machines Execution Semantics Issues Discussion

5.7.3.1 Issues with Instantaneous States: Deadlocks (Infinite Looping)

In Modelica the global variable `time` represents the simulated real time. Because computations at event iterations do not consume simulated time, states can be entered and exited at the same simulated point in time. When instantaneous states are allowed it is possible to model deadlocks that lead to an infinite loop at the same simulated point in time. Consider Figure 25 (left): the behavior will loop infinitely without advancing the simulated time, and a simulation tool will stop the simulation and report an issue.

If such a behavior is intended and the state machine should loop continuously and execute actions, the modeler can break the infinite looping at a time instant by adding a time delay⁷⁷ to one of the transitions involved (see the right state machine in Figure 25). In doing so the simulation time is advanced and the tool will continue simulating.

Large models can contain deadlocks that are not as obvious as in the simple example depicted in Figure 25. Infinite looping is often not modeled on purpose and is hard to prevent and to detect. This issue is still subject to future research in ModelicaML.

Figure 25: Deadlocks (infinite looping) example

5.7.3.2 Issues with Concurrency When Using Event Queues

Consider the state machine in Figure 26 modeled using (IBM, 2013). The events *ev1* and *ev2* are generated simultaneously at the same time instant when entering the initial states of both regions. However, it is not obvious to the modeler in which order the generated events are dispatched to the state machine.

⁷⁷ `AFTER(expression)` is a ModelicaML macro. It is expanded to the guard condition `state_local_timer > expression`.

When the simulation is started Rhapsody shows that the events are generated and put into the event queue in the following order: *ev1*, *ev2*. When restarting the simulation the order of events in the queue will always be the same. Obviously, there is a mechanism that determines the order of region executions based on the model data.

Next, these events are dispatched to the state machine one after the other. First the event *ev1* is dispatched and the transition to *state_1* is executed, then the event *ev2* is dispatched and the transition from *state_1* to *state_2* is executed, as shown in Figure 27.

According to this behavior the occurrence of *ev2* is delayed. However, with the state machine in *state_1* no *ev2* occurs. The event *ev2* occurs when the state machine is in *state_0*.

This behavior seems to be similar to the concept of deferred events described in the UML specification (OMG UML, 2011, p. 563): “An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event pool while another non-deferred event is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.”.

Figure 26: Events queue issue 1

Figure 27: Events queue issue 1 simulation⁷⁸

Consider Figure 28. It shows a slightly modified state machine. The event *ev2* is generated inside the left region and the *ev1* is generated inside the right region. The order of events in the queue is now reversed: *ev2*, *ev1*.

Figure 29 shows the simulation result. In contrast to the assumption above, the event *ev2* is not deferred. It is dispatched to the state machine and discarded after the transition to *state_1* is taken. The state machine finally stays in *state_1*, which is a different behavior than in Figure 27.

Along with the fact that the modeler cannot control the execution order of the parallel regions (this issue is addressed in Section 5.7.3.3) and, thus, the order in which events are generated, the main issue here is that it leads to behavior that is unpredictable and cannot be expected from the modeler's perspective.

Even experienced modelers were not able to immediately anticipate the behavior described in the example above before simulating it in Rhapsody and were surprised by the behavior pointed out in this example. In typical implementations of UML state machines a simulation, queuing of events is an instrumentation of the simulation environment that the modeler needs to know and take into account even if it is not a part of the behavior the modeler expresses.

⁷⁸ The state machine ends up in *state_2* in both regions.

Figure 28: Events queue issue 2

Figure 29: Events queue issue 2 simulation⁷⁹

The example above shows that the state machine semantics defined in UML prevents simultaneous events from being processed simultaneously. However, the queuing of events was not part of the behavior that the modeler has expressed.⁸⁰

If events occurred simultaneously, the state machine should react to both events at the same time. Moreover, a parallel handling of events is a prerequisite for using multicore simulations or for modeling multicore embedded systems. In this case, all

⁷⁹ The state machine ends up in *state_1* in both regions.

⁸⁰ If there is a need for queuing of events then it should be expressed and modeled explicitly.

independent state machines must react immediately in parallel to all simultaneous events.

Figure 30: Same model in ModelicaML⁸¹

Figure 30 shows the same model in ModelicaML. In contrast to the examples above, regardless of whether the event `ev2` is generated in the right region or in the left region, the execution behavior is the same: the state machine always ends up in `state_1` in both regions.

This is because the events that occur simultaneously are processed (i.e., dispatched and consumed) in parallel during the next state machine evaluation cycle. When the transitions from the states `state_0` to `state_1` in both regions are executed, event `ev2` is also consumed and the state machine stays in both states `state_1` because with the state machines in states `state_1` in both regions no event `ev2` is generated.

5.7.3.3 Issue with Concurrent Execution in Regions

In ModelicaML, there are two ways to model behavior that is orthogonal or will be executed in parallel. The modeler can use separate state machines for one class. In this case the state machines must be mutually exclusive in terms of setting the same variables for the class. Modelica ensures that the state machines are executed in parallel because each state machine is translated into individual algorithm sections which in Modelica can be treated in parallel (simultaneously).

⁸¹ The state machine ends up in `state_1` in both regions.

If there are dependencies between variables then the modeler should use one state machine and multiple regions. Regions represent orthogonal behavior, i.e., the state machine can be in multiple states in different regions at the same time. When using this approach, it is recommended that the behavior inside different regions should also be mutually exclusive, i.e., independent of each other.

However, in order to enable the usage of fork and join constructs in order to trigger state activations or deactivations in different regions, the code needs to be in the same Modelica algorithm section. In order to let the modeler know and control the order in which the behavior of regions is executed, in ModelicaML each region is automatically given a priority relative to its neighboring regions (a lower priority number implies higher execution order priority; for example, code in the region with priority 1 is executed prior to code in the region with priority 2).

The priority definition is necessary when there are actions in multiple parallel regions that will set the same variables, as illustrated in Figure 31.⁸² Moreover, it facilitates debugging, because the modeler knows exactly the execution order of regions and can infer the order in which behavior code in each region is executed.

Consider Figure 31: since *Region_0* is given a higher execution priority, it will be executed prior to *Region_1*, which has lower execution priority. The result is that *openValve* is set to *false*. Note that if *State_0* in *Region_0* was a composite state, then its internal behavior would also be executed before the behavior of *State_0* in *Region_1*.

Figure 31: Definition of priority for parallel regions

⁸² This example is artificial and is meant for illustration purposes only. Modeling such behavior is not a productive exercise.

Priorities are set by default by the ModelicaML modeling tool. The modeler can change priorities and, in doing so, define the execution sequence explicitly to ensure the intended behavior.

Furthermore, a region priority definition is also used to resolve the issue with exiting and entering composite states, as well as inter-level transitions as discussed in Section 5.7.3.7 and 5.7.3.8.

5.7.3.4 Issues with Conflicting Transitions

When a state has multiple outgoing transitions and trigger and guard conditions overlap—i.e., they can be true at the same time—then the transitions are said to be in conflict (OMG UML, 2011, p. 575): “Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously.”

When triggers (except the trigger “all,” which means the acceptance of any event that is not explicitly referenced in another transition from the same state) are defined for a transition, then there is no issue with overlapping guard conditions because simultaneous events are not processed in parallel in UML, as pointed out in Section 5.7.3.2.

This is different in ModelicaML, because events are processed in parallel and can overlap. However, it is not clear from the UML specification what should happen if conflicting transitions do not have any trigger or do have the trigger “all” (implying the acceptance of any event that is not explicitly referenced in another transition from the same state) and guard conditions defined, that can evaluate to true at the same time. This issue is addressed in Section 5.7.3.5

Furthermore, if the conflicting transitions are at different hierarchy levels, UML defines the following: “In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an implicit priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy.” This is discussed in Section 5.7.3.6.

5.7.3.5 Priorities for State-Outgoing Transitions

Consider the state machine in Figure 32. If x and y are greater than 2 at the same time, both guard conditions evaluate to true. In ModelicaML, which transition will be taken then is determined by the transition execution priority defined by the modeler.

As for regions (discussed in Section 5.7.3.3), conflicting transitions coming out of a state are prioritized in ModelicaML. The modeling tool sets priorities for transitions by default, but the modeler can change priorities to take control of the execution semantics and to ensure deterministic behavior.

Figure 32: Priority definition for state-outgoing transitions

5.7.3.6 Priority Schema for Conflicting Transitions

If there are no conflicting transitions at different levels, then the behavior of ModelicaML state machines with regards to the triggering of transitions is the same as defined in UML. However, if there are conflicting transitions that are at different levels, UML defines the priority schema as follows (OMG UML, 2011, p. 576): “By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.” No rationale is documented for this decision, as pointed out in (Crane & Dingel, 2005).

Figure 33: Transitions at a higher level have higher priority

Consider the state machine in Figure 33. What should happen when x and y are greater than 2 at the same time? One possible answer could be: transition to state *NOT OK* is taken. Another answer could be: transition to state *NOT OK* is taken and then transition to state *OK* is taken. Yet another answer could be: transition to state

OK is taken and since *State_0* is deactivated, no further reaction inside *State_0* is to be expected. The latter is implemented in ModelicaML.

In ModelicaML the priority scheme is different from UML, in that outgoing transitions of a composite state have higher priority than transitions inside the composite state. The rationale for this decision is as follows:

- This semantics is more intuitive and clear. Hierarchical modeling and encapsulation are crucial features for modeling complex behavior. Complex state machines are typically modeled top-down.

A scheme that gives transitions at a higher level a higher priority reflects the top-down modeling approach. Moreover, it enables the reader to anticipate the behavior of the state machine by only looking at the state machine diagram and not being forced to first get to know the lowest level of detail in order to be able to conclude whether a transition at a higher level will eventually trigger if enabled by the events and the guard specified.

For example, if composite state behavior is not shown at the current diagram level, then the modeler should be able to reason about the state-incoming or -outgoing transitions at this diagram level without having to look inside the diagram that presents the internal behavior of this composite state. Consequently, the UML priority scheme can also lead to behavior where the composite state is never exited because events are consumed at some level further down the hierarchy of the composite state.

- The ModelicaML priority scheme reduces the complexity of the code generator. For example, as already mentioned above, event queues are not used in Modelica. To prevent transitions for a composite state from consuming events that are already consumed inside the composite states, events need to be marked or removed from the queue. Such an implementation would increase the size and complexity of the code generator, as well as of the Modelica code generated.

5.7.3.7 Issues with Inter-Level Transitions

This section discusses issues concerning the execution order of actions as a result of transition executions. Actions can be *entry/do/exit* actions of states or *effect* actions of transitions. The order in which actions are executed is important when actions are dependent, i.e., if different actions set or read the same variables.

Consider the state machine in Figure 34. Assume that each state has entry and exit actions, and that *Region_0* is always given a higher priority compared to *Region_1*, *Region_2*, etc.

When the state machine is in state *a* and *cond1* is true, the question is: in which order are the states activated and the respective entry actions executed? This case is not addressed in the UML specification.

From the general UML state machine semantics definition we can deduce that substates cannot be activated as long as their containing (i.e., composite) state is not activated. For example, it is clear that state *b* has to be activated before states *c* and *i* can become active.

Furthermore, we can argue that since the modeler explicitly created an inter-level transition, state *e2* should be activated first, i.e., before states in neighboring regions at the same (i.e., *f*) or at a higher state hierarchy level (i.e., *g*, *h*, or *i*). Hence, when the inter-level transition from state *a* to state *e2* is taken, the partial states activation sequence and its resulting order of entry-action execution should be: *b*, *c*, *d*, *e2*. However, in which sequence will states *f*, *g*, *h*, and *i* be activated? Possible answers are: *i*, *h*, *g*, *f*, or *i*, *g*, *h*, *f*, or *f*, *g*, *h*, *i*, or *f*, *h*, *g*, *i*.

In ModelicaML, this issue is resolved as follows: first, all states containing the target state and the target state itself are activated. Next, based on the region execution priority, the initial states from neighboring regions are activated. Since the priority in this example is defined for regions from left (highest) to right (lowest) for each composite state, the activation order would be *b*, *c*, *d*, *e2*, *f*, *g*, *h*, *i*. Vice versa, if the region priority would be defined the other way around (from right to left) the activation order would be *b*, *c*, *d*, *e2*, *i*, *h*, *g*, *f*.

A similar issue exists regarding the transition to state *a* when the state machine is in state *e2* and when *cond2* is true. Here the states deactivation and exit-action execution order are involved. The resulting deactivation sequence in ModelicaML would be: *e2*, *f*, *d*, *g*, *h* (based on the region priority definition), *c*, *i*, and *b*.

Figure 34: Inter-level transition example

5.7.3.8 Issues with Fork and Join

Like Section 5.7.3.7, this section addresses issues regarding the activation and deactivation of states. However, in this case, UML fork and join constructs are addressed. Consider the state machine in Figure 35. With the state machine in state *a* the questions are:

- In which sequence are states *b*, *c*, *d*, *e*, and *f* activated when the transitions (fork construct) from state *a* is executed?
- In which sequence are states *b*, *c*, *d*, *e*, and *f* deactivated when the transitions (join construct) to state *g* are executed?

This case is also not addressed in the UML specification. In ModelicaML, first the parent states of the transition target states are activated. Then the target states themselves are activated based on the fork-outgoing transition priority. Next the initial states in the neighboring regions are activated based on the region priority definition.

Again assume that each state has entry and exit actions, and that *Region_0* is given higher priority compared to *Region_1*, *Region_2*, etc. The resulting state activation sequence (and respective execution of entry actions) for the fork construct would be: *b*, *d*, *e* (based on the fork-outgoing transitions priority), *c* and *f* (based on their region priority). The resulting deactivation for the join construct would be: *d*, *e* (based on the join-incoming transitions priority), *c*, *f* (based on their region priority definition), and *b*. In any case, the modeler can define the execution order explicitly.

Figure 35: Fork and join example

5.7.4 Related Work

In previous work, researchers have already identified the need to integrate UML/SysML and Modelica, and have partially implemented such integration. In (Pop, Akhvlediani, & Fritzson, 2007) the basic mapping of the structural constructs from Modelica to SysML is identified.

The SysML Parametrics concept is used in (Johnson, Kerzhner, Paredis, & Burkhart, 2012) for the integration of continuous-time behavior into SysML models.

A concept to use SysML to integrate models of continuous-time dynamic system behavior with SysML information models representing systems engineering problems as well as rules for the graph-based bidirectional transformation of subsets of SysML and Modelica models is presented in (Johnson, Kerzhner, Paredis, & Burkhart, 2012). Finally, the recent OMG standard SyM (OMG SyM, 2012) standard defines a bidirectional translation of SysML and Modelica models. However, none of them include the translation of UML state machines into Modelica.

Regarding state machines, a list of general challenges with respect to statecharts is presented in (Von der Beeck, 1994) and a summary of different statechart variants is provided. The variants are classified and compared with each other. UML state machines are not included because the paper was written before the UML specification was in place. Consequently (Von der Beeck, 1994) addresses features, such as preemptive vs. non-preemptive interrupt or operations vs. denotation style semantics etc., which are not related to the UML state machines specification.

Conversely, it does not address the issues with event queues, concurrent region execution, and constructs such as forks and joins and does not resolve issues with conflicting transitions and the execution order of state actions. UML state machines can be translated into Interval Time Petri Nets in order to take advantage of existing Petri Nets analysis tools for checking consistency and temporal constraints (Hammal, 2005).

This is similar to the approach discussed in this article to translate UML state machines into Modelica in order to leverage the existing Modelica technology for hybrid system dynamics modeling and simulation. Like in this article, (Hammal, 2005) also points out the lack of precise semantics of UML state machine formalisms; to overcome them, Hammal enhances UML state machines, for example, by execution priorities for transitions, although not for regions. as proposed in Section 5.7.3.3.

Moreover, (Hammal, 2005) does not explain how to deal with for and join constructs (see Section 5.7.3.8) and inter-level transitions (see Section 5.7.3.7) (Simons, 2000). stresses the difference between classical state machines and flowcharts, discusses redundant constructs such as fork and join, and points out

what properties of state machine formalisms are required in order to enable independent verification and compositional testing.

Finally, in order to avoid non-determinism, (Simons, 2000) proposes the same approach for handling of inter-level conflicting transitions, as described in Section 5.7.3.6. However, that work does not discuss the issues with overlapping guards or simultaneous events for transitions at the same level.

In (Crane & Dingel, 2005) the fact is stressed that the semantics of different statechart variants are not the same even though the concrete syntax (graphical notation) is the same. They also point out that execution semantics strongly depend on the implementation decisions taken (e.g. the final semantics are defined by the code generator), which are not standardized in UML.

A working draft for the specification of a XML representation of a subset (e.g. for/join, decision, and junction constructs are not supported) of the UML state machines and a normative algorithm for the interpretation of the State Chart XML documents is provided in (W3C, 2013).

In the State Chart XML document, the order of elements (e.g. regions or transitions) is used to resolve potential conflicts. In the event of conflicts the first (in the order they appear in the document) transition or region is always chosen. This approach is similar to the proposal of transition and region priority here.

Also, the notion of eventless transition is similar to the way such transitions are handled in ModelicaML. Queues are used in (W3C, 2013) for sequential event processing and do not include fork/join constructs, which may lead to issues with state entry/exit action execution, as discussed in Section 5.7.3.8.

Modelica does not provide any graphical notation to describe the behavior of a class. In (Schamai, Fritzson, Paredis, & Pop, 2009) an initial version of an approach for using the UML-based notation of a subset of UML state machines and activity diagrams for modeling the behavior of a Modelica class is presented.

Few implementations of the translation of statecharts into Modelica exist (but see (Ferreira & de Oliveira, 1999), and (Carloni, Roberto, & Pinto, 2006)⁸³. However, none implements a comprehensive set of state machines concepts as defined in the UML specification and supported by the approach mentioned in this chapter.

The main focus of this chapter is the resolution of the listed issues that are related to the execution semantics of UML state machines that are to be translated into Modelica code. The main objective of this approach is to enable modeling of hybrid system dynamic behavior (continuous-time, event-based, and discrete-time) by allowing the engineer to choose appropriate formalisms.

⁸³ StateGraph (Otter, Malmheden, Elmqvist, Mattsson, & Johnsson, 2009) uses a different graphical notation compared to the UML notation for state machines.

Discussions in (Von der Beeck, 1994), (Simons, 2000), and (Hammal, 2005) focus on formal aspects that enable compositional semantics. Therefore, limitations are proposed at the price of the expressiveness and concision of the state machines formalism.

However, formal verification is hard to achieve for large models with hybrid dynamics anyway (Carloni, Roberto, & Pinto, 2006). In contrast, the intention here is to eliminate the ambiguities of UML state machine execution semantics and to enable the usage of the state machine formalisms to model and simulate state-based parts of system behavior in combination with other formalisms, such as equations, which enable the modeling and simulation of continuous-time system dynamics.

5.8 ModelicaML Prototype

The ModelicaML prototype⁸⁴ is based on the following architecture and can be downloaded from OpenModelica Project (Open Source Modelica Consortium, 2013):

- Eclipse MDT Tool Papyrus (The Eclipse Foundation - Papyrus Project, 2013) is used for UML-based modeling. ModelicaML builds upon Papyrus and customizes the GUI (e.g. Model Explorer, menus, etc.) and adds new views that are specific to vVDR or Modelica.
- A ModelicaML model can be validated in order to check constraints and possible inconsistencies by using a validator plug-in using (The Eclipse Foundation - EMF Project, 2013), which informs the modeler about inconsistencies or restriction violations. Note that the validation of a ModelicaML model does not (yet) replace the semantic analysis that is performed by a Modelica compiler which checks Modelica semantics application in more detail.
- The ModelicaML code generator that generates Modelica code from ModelicaML models is implemented using (The Eclipse Foundation - Acceleo Project, 2013), which follows the MDA approach and the model-to-text recommendations of the OMG (OMG MOF2T, 2013).
- Finally, Modelica tools such as (Open Source Modelica Consortium, 2013), (Dassault Systèmes Dymola, 2013), or (Wolfram Research, 2013) are used to load the generated Modelica code and to simulate it.

⁸⁴ Building a ModelicaML tool from scratch would not be possible within the scope of a Ph.D. study. There are several decades of man years inside the development of tools and frameworks the ModelicaML prototype builds on.

Figure 36 shows an example of a view of the ModelicaML tool GUI. The default⁸⁵ GUI includes the model explorer view (see Figure 36 left top), the diagram editor (see Figure 36 right top), and a properties view (see Figure 36 right bottom) to modify properties of the element selected in model explorer or on the diagram.

Figure 36: ModelicaML prototype GUI example

The Model explorer and the properties views are adapted to ModelicaML's needs. Moreover, the ModelicaML prototype includes special-purpose views such as:

- class components tree view (see Figure 50 for examples),
- bindings view (see Section 6.2.4 for examples),
- traceability view (see Section 6.2.7 for examples),
- Modelica models synchronization view (mentioned in Section 5.6.2).

⁸⁵ In Eclipse the user can customize the layout of the GUI elements and their appearance.

5.9 Conclusion

This chapter presents a step towards, and a proof of concept for, a unified executable system modeling language and environment using the UML and Modelica languages and open-source UML modeling (The Eclipse Foundation - Papyrus Project, 2013) and simulation tools (Open Source Modelica Consortium, 2013).

UML and Modelica are complementary languages supported by two active communities. By integrating UML and Modelica into ModelicaML, we combine the expressive, formal language for differential algebraic equations and discrete events of Modelica with the expressive UML graphical notation for requirements, structural decomposition, logical behavior, and corresponding crosscutting constructs. In addition, the two communities are expected to benefit from the exchange of multi-domain model libraries and the potential for improved and expanded commercial and open-source tool support.

Section 5.7 presents a proof of concept for the translation of a comprehensive subset of UML state machine concepts into executable Modelica code. It discusses UML semantics issues and presents their resolution, which is implemented in ModelicaML. It also explains how the state machine formalism is used in ModelicaML to model parts of class behavior and integrated with other formalisms to describe continuous-time behavior of the component. In particular, Section 5.7.3.2 questions the use of an event queue that prevents simultaneous events from being processed in parallel. When procedural code is used to implement state machine execution, Section 5.7.3.3 makes a proposal to include priority for regions. A region priority also resolves the issue of state activation or deactivation order in the case of inter-level state transitions (Section 5.7.3.7) or fork/join constructs (Section 5.7.3.8). Section 5.7.3.5 introduces execution priority for conflicting state-outgoing transitions in order to allow the modeler to control the execution and to ensure that the state machine behaves as intended.

ModelicaML and the implemented prototype are primarily used to show how the concepts of the new approach for models-based verification of system against requirements can be implemented. The following Chapter 6 illustrates the use of the ModelicaML prototype in case studies.

Chapter 6

Application Examples

6.1 Introduction

This chapter shows the application of the vVDR method presented in Chapter 3 on examples. The examples are modeled using ModelicaML prototype, introduced in Chapter 5. In addition, the examples presented show the application of the automated model composition approach enabled by the framework presented in Chapter 4.

The first example in Section 6.2 is a complete run through the application of the vVDR method and the usage of the proposed automation approach covering different aspects. It shows requirement and scenario formalizations, presents examples for binding specifications and explains how the model generation algorithm (see Section 4.1) is implemented in the ModelicaML prototype.

The second example in Section 6.3 is a case study that was performed as part of the OPENPROD project (ITEA2, 2013). This case study is focusing more on requirements formalization. Another case study, not included in this text, is published in (Liang, Schamai, Rogovchenko, Sadeghi, Nyberg, & Fritzson, 2012).

6.2 Two-Tank System

This example is concerned with a typical control systems problem. A system composed of several tanks, sensors, and controllers shall maintain a predefined level of liquid in tanks⁸⁶ (see Figure 37).

⁸⁶ This example is inspired by (Fritzson, 2004, p. 385 ff)

Figure 37: Application example 1: Simple two-tank system

6.2.1 Formalizing Requirements

Assume that the following requirements are imposed on the system:

- **Req. 001:** The volume of each tank shall be at least 2 m^3 .
- **Req. 002:** The level of liquid in a tank shall never exceed 80% of the tank height.
- **Req. 003:** With all tanks full, the maximum time to drain all tanks shall be 50 seconds.
- **Req. 004:** After each change of the tank input flow, the controller shall, within 20 seconds, ensure that the level of liquid in each tank is equal to the reference level with a tolerance of $\pm 0.05 \text{ m}$.

Using vVDR, a requirement analyst will formalize requirements by identifying quantifiable properties and expressing when a requirement is violated and when not (see Section 3.7.3.2).

Figure 38 and Figure 39 show the formalized model, i.e., the corresponding requirement violation monitor for the first two requirements, and the Modelica code generated from these ModelicaML models. Note that these requirements do not have any evaluation precondition. Their violation monitors will only switch between *violated* or *not violated* status right from the very first simulated time instant.

The volume of each tank shall be at least 2 m3.

Figure 38: Application example 1: Req. 001 formalized

The corresponding generated Modelica code for Req. 001:

```

within TwoTanksExample.Requirements;
model Volume_of_a_tank "The volume of each tank shall be at
least 2 m3. "
  input Modelica.SIunits.Volume tankVolume= 0;
  parameter Modelica.SIunits.Volume requiredVolume= 2;
  output Integer status= if tankVolume < requiredVolume then 2
  else 1;
end Volume_of_a_tank;
  
```

The level of liquid in a tank shall never exceed 80% of the tank height.

Figure 39: Application example 1: Req. 002 formalized

The corresponding generated Modelica code for Req. 002:

```

within TwoTanksExample.Requirements;
model Max_level_of_liquid_in_tank "The level of liquid in tank
shall never exceed 80% of the tank height."
  input Real levelInTank= 0;
  input Real tankHeight= 0;
  parameter Real limit= 0.8;
  output Integer status= if (levelInTank > limit*tankHeight)
  then 2 else 1;
end Max_level_of_liquid_in_tank;
  
```

By contrast, Figure 40 and Figure 41 show violation monitors⁸⁷ for the third and fourth requirement, which do have preconditions. These monitors are modeled using ModelicaML state machines⁸⁸.

With all tanks full, the maximal time to drain all tanks shall be 50 seconds.

Pre-condition: *All tanks are full and system is being drained*

Figure 40: Application example 1: Req. 003 formalized

⁸⁷ The generated Modelica code is omitted for the sake of brevity.

⁸⁸ The semantics of ModelicaML state machines are described in Section 5.7.

After each change of the tank input flow, the controller shall, within 20 sec., ensure that the level of liquid in tank is equal to the reference level with a tolerance of +/- 0.05 m.

Figure 41: Application example 1: Req. 004 formalized using a state machine

As already pointed out in Section 3.7.3.2, this requirement formalization is independent of a specific system design model. It can be reused for any particular design—for example, with different numbers or tanks and different controlling strategies—assuming that the identified requirement violation monitor inputs are provided by the system design model to be verified.

6.2.2 Formalizing Design

The system design model can be created by the system designer using the ModelicaML tool. Alternatively the model can be created using a Modelica tool and then imported into the ModelicaML environment (using the approach mentioned in Section 5.6.2) to perform system design verification.

The Modelica code for this system design model⁸⁹ is the following:

```

within TwoTanksExample.Design;
model SystemEnvironment
    TwoTanksExample.Design.Components.TanksConnectedPI
    twoTanksSystem;
end SystemEnvironment;



---



within TwoTanksExample.Design.Components;
model TanksConnectedPI
    TwoTanksExample.Design.Components.LiquidSource source;
    TwoTanksExample.Design.Components.Tank tank1(area = 1);
    TwoTanksExample.Design.Components.Tank tank2(area = 1.3);
    replaceable
    TwoTanksExample.Design.Components.PIcontinuousController
    controller1(ref = 0.25);
    replaceable
    TwoTanksExample.Design.Components.PIcontinuousController
    controller2(ref = 0.4);
    equation
        connect(source.qOut, tank1.qIn);
        connect(tank1.tSensor, controller1.cIn);
        connect(tank1.tActuator, controller1.cOut);
        connect(tank1.qOut, tank2.qIn);
        connect(tank2.tSensor, controller2.cIn);
        connect(tank2.tActuator, controller2.cOut);
    end TanksConnectedPI;



---



within TwoTanksExample.Design.Components;
model LiquidSource
    TwoTanksExample.Design.Interfaces.LiquidFlowOut qOut;
    input Real flowLevel= 0;
    equation
        qOut.lflow = flowLevel;
    end LiquidSource;



---



```

⁸⁹ Adopted from (Fritzson, 2004, p. 385 ff)

```

within TwoTanksExample.Design.Components;
model Tank
    TwoTanksExample.Design.Interfaces.ReadSignalOut tSensor;
    TwoTanksExample.Design.Interfaces.ActSignalIn tActuator;
    TwoTanksExample.Design.Interfaces.LiquidFlowIn qIn;
    TwoTanksExample.Design.Interfaces.LiquidFlowOut qOut;
    parameter Real flowGain= 0.05;
    parameter Real minV= 0;
    parameter Real maxV= 10;
    parameter Real area= 1;
    parameter Real height= 2;
    Real volume= area * height;
    Real levelOfLiquid;

    equation
        // Mass balance equation
        der(levelOfLiquid) = (qIn.lflow - qOut.lflow)/area;
        qOut.lflow =
TwoTanksExample.Design.Library.limitValue(minV, maxV, -
flowGain*tActuator.act);
        tSensor.val = levelOfLiquid;
end Tank;

```

```

within TwoTanksExample.Design.Components;
model PIcontinuousController
    extends TwoTanksExample.Design.Library.BaseController ;
    Real x;
    equation
        der(x) = error/T;
        outCtr = K*(error + x);
end PIcontinuousController;

```

```

within TwoTanksExample.Design.Library;
partial model BaseController
    TwoTanksExample.Design.Interfaces.ReadSignalIn cIn;
    TwoTanksExample.Design.Interfaces.ActSignalOut cOut;
    parameter Real K= 2;
    parameter Real T= 10;
    input Real ref= 0.25;
    Real error;
    Real outCtr;
    equation
        error = ref - cIn.val;
        cOut.act = outCtr;
end BaseController;

```

```

within TwoTanksExample.Design.Library;
function limitValue
    input Real pMin;
    input Real pMax;

```

```

    input Real p;
    output Real pLim;
    protected

    algorithm
        pLim := if p>pMax then pMax
                else if p<pMin then pMin
                else p;
    end limitValue;

```

```

within TwoTanksExample.Design.Interfaces;

connector ActSignalIn
    input Real act;
end ActSignalIn;

connector ActSignalOut
    output Real act;
end ActSignalOut;

connector LiquidFlowIn
    input Real lflow;
end LiquidFlowIn;

connector LiquidFlowOut
    output Real lflow;
end LiquidFlowOut;

connector ReadSignalIn
    input Real val;
end ReadSignalIn;

connector ReadSignalOut
    output Real val;
end ReadSignalOut;

```

6.2.3 Formalizing Scenarios

The tester (see Section 3.7.2.3) will define verification scenarios. For example, in the scenario models shown in Figure 42 the system will be drained or filled with different input flow levels. Each of the scenario models contains action code to stimulate the system model.

Figure 42: Example of a verification scenario

6.2.4 Specifying Bindings

As pointed out in Section 4.2.1, the bindings specification is stored in such a way that none of the related artifacts is modified. Figure 43 shows examples of bindings defined for requirement monitor clients.

Figure 43: Application example 1: Bindings for requirements

Figure 44 shows bindings for system design and scenarios.

Figure 44: Application example 1: Bindings for design and scenarios

Figure 45 shows an example of a client operation. The client is the predefined reference level that `controller1` should have. The controller model has the component `ref` that is the client. However, the client is assumed to be the component of the `controller1`. Thus, instead of defining the class component `ref` component to be the client, we define the component `controller1` (instance of the controller model in the `TwoTanksSystem` model that is one level up in the instantiation hierarchy) to be the client and reference the `ref` component in the client operation.

Figure 46 shows an example of a provider operation. The provider for the mediator *all tanks are empty* has to be the overall tank system because we need to determine the level in all tanks in order to express when all tanks are empty. For that the component `twoTanksSystem` (instance of the model `TwoTanksSystem` in the `SystemEnvironment` model) is defined to be the provider. Further, it refers in its provider operation to the level of liquid of each tank.

Note that the provider is specific to the given system design model. To verify another system design alternative (e.g. a design that includes 3 tanks) only a new

additional provider (possibly with an operation) will have to be specified. The mediator as well as the clients will remain the same.

Figure 45: Application example 1: Client operation

Note that the component `levelOfLiquid` (in `tank1`) in Figure 46 is client and provider at the same time. It is a client because in some scenarios its start value is provided by the scenario model. At the same time it is a provider for some requirement violation monitor clients.

Figure 46: Application example 1: Provider operation

6.2.5 Generating Verification Models

After having defined bindings, the verification models can be generated. In the ModelicaML prototype there are several ways to generate models:

1. Combining scenarios and requirements only based on explicit relations between scenarios and requirements,
2. Discovering relations between scenarios and requirements,
3. Considering all possible combinations of scenarios and requirements.

All three features can be used independently. However, it is also possible to combine them as motivated in Section 4.3.2 in a multistep approach. For example, we could discover relations between scenarios and requirements and store new relations, before generating verification models based only on explicit relations. Alternatively, we can use the third option that will consider all possible combinations, discover new relations and perform verification at the same time.

The first feature above is supposed to generate optimal verification models (see Section 3.7.6). Other features will generate suboptimal verification models because they will combine scenarios and requirements without knowing whether the scenarios are appropriate for testing the selected requirements, i.e., if requirements will be evaluated in the selected scenarios.

The downside of using a multistep approach here is that each step requires the same amount of time to execute the models. In simulations that are not time consuming this is not an issue. However, for projects that contain a large number of scenarios and complex system design models with long execution times this approach may be not appropriate.

In the following the first two features are illustrated.

6.2.5.1 Options for Models Generation

Each of the features mentioned in the previous section starts with a dialog for specifying model generation options. The user can select the system design model to be verified. Recall that in vVDR any model can be considered a system design model (see Section 3.7.4). It may be the entire system or a subsystem. This selection will affect the subset of implemented requirements and scenarios to be considered.

In addition, the user can specify where to search for requirements, scenarios, or bindings. This way the user has control over which requirements or scenarios will be considered⁹⁰.

The tool will search recursively in any nested structure of the specified package. For example, the user may decide to include all requirements or scenarios or only normal usage scenarios, or only stress scenarios, etc. Similar, the user may want to include a specific set of bindings (i.e., mediators) in order to use a particular

⁹⁰ Assuming that requirements and scenarios are organized in packages.

configuration (if mediators are used to capture configurations as discussed in Section 3.7.6).

Optionally, the user may specify a super model from which each of the verification models will inherit. This option may be of interest if generated verification models will have to implement particular interfaces for further integration.

6.2.5.2 Discovering Relations Between Scenarios and Requirements

The following options are available and can be selected to discover relations between scenarios and requirements:

- Whether the algorithm should, for each scenario, include all requirements which are referenced by scenarios by the *use to verify* relation
- Whether the algorithm should, for each scenario, exclude all requirements that are referenced by scenarios by the *do not use to verify* relation.

Thus, if a requirement is referenced by both relations, *do not use to verify* has higher priority assuming the second option is selected.

Figure 47: Options dialog: the model generation options for relations discovery

Clearly, the option “to include all requirements with unknown relations” is preselected because the main purpose of this feature is to discover new relations.

When invoked, the tool will determine which scenarios can be used and which requirements are implemented. The subset of found scenarios and requirements is presented for use, and the user can modify the proposed selection (see Figure 48 and Figure 49).

Figure 48: Selecting scenarios

Now that the verification models are created (see example of generated models in Figure 51) followed by the option to execute the models in order to generate the report provided that bindings could be generated automatically for all models (i.e., without the need for manual decisions, as explained in Section 4.1).

If not all bindings could be generated automatically and manual decision is required to, for example, select providers (see Section 4.1), the tool will still generate verification models and indicate for which components manual decisions are subsequently required. The user can make manual decisions by using the

components tree view⁹¹ of the ModelicaML prototype as illustrated in Figure 50. Note, as explained in Section 4.2.5, the manual decisions made can be stored in preferred bindings (see the option in the bottom right dialog in Figure 50).

Figure 49: Selecting requirements

⁹¹ Components tree instantiates a model as explained in Section 4.2.1

Figure 50: Defining bindings manually

The model execution and results analysis feature (see Figure 51) may be invoked at any time by selecting the package containing verification models⁹². ModelicaML is integrated with the OpenModelica compiler (OMC) (Open Source Modelica Consortium, 2013), which is used for simulating models.

Optionally, another simulation environment can be used. Here the user would generate Modelica code from the generated verification models and simulate using some other environment. The generated simulation result files (e.g. .mat) can then be used directly (see option “skip simulation” in Figure 51) for results analysis.

⁹² For one selected package, the tool will collect all verification models which are identified based on the «VerificationModel» stereotype (see Section A.8)

Figure 51: Created verification models and simulation options

After simulating models and post-processing results, the tool presents an overview of the relations discovered (see Figure 52). A new *use to verify* the relation between scenario s and requirement r is discovered if the scenario s stimulates the system model in such a way that the requirement r is evaluated (i.e., it is either *violated* or *not violated* for at least one simulated time instant). New discovered relations can be stored in the model (see Figure 52) and used to generate models based on explicit relations described in the following section.

Figure 52: Relations discovered between scenarios and requirements

6.2.5.3 Generation of Verification Models Based on Explicit Relations

Generating models based on explicit relations between scenarios and requirements, such as *use to verify* or *do not use to verify*, is a procedure which is similar to the one presented in Section 6.2.5.2. Figure 53 shows the preselected options. In the next step the user can still modify the scenario selection (similar to Figure 48).

Figure 53: Options dialog for model generation based on explicit relations

6.2.5.4 Generated Verification Models

Each verification scenario contains all simulation related parameters such as start and stop time of the simulation, etc. (see Annex A.8). These settings are used to automatically invoke the Modelica simulation engine.

The following naming conventions are used when generating models (see an example in Figure 54).

Naming convention for package:

- ReD_SYSTEMMODELNAME_TIMESTAMP.
 - ReD means relations discovery.
 - DESIGNMODELNAME is the name of the system design model used.

- `TIMESTAMP` is the time (year month day hour seconds) when the model was generated.
- `aVeMs_SYSTEMMODELNAME_TIMESTAMP`.
 - `aVeMs` refers to automatically generated verification models.
- `VeMs_SYSTEMMODELNAME_TIMESTAMP`.
 - `VeMs` refers to verification models generated based on explicit relations.

Naming convention for models:

- `aVeM_SCENARIOMODELNAME`
 - `aVeM` refers to an automatically generated verification model.
 - `SCENARIOMODELNAME` is the name of the scenario used.
- `aVeM_SCENARIOMODELNAME`
 - `VeMs` means that this verification model was generated based on explicit relations.

Naming convention for components in verification models:

- `sm_SYSTEMMODELNAME`: `sm` prefix stands for system model
- `vs_SCENARIOMODELNAME`: `vs` prefix stands for verification scenario
- `req_ID_REQUIREMENTMODELNAME`:
 - `req` prefix means that this component is an instantiation of a requirement violation monitor model.
 - `ID` is the requirement id.
 - `REQUIREMENTMODELNAME` is the name of the requirement violation monitor model.

Bindings that are inferred for each client in a verification model are stored in its first-level components using Modelica modifiers. For example, Figure 55 shows a generated verification model instantiated in the ModelicaML prototype components tree view. Below the generated Modelica code is shown.

Figure 54: Naming conventions for generated models

ModelicaML verification model

Modelica code

```
model VeM_Change_of_input_flow_1
  annotation(Diagram(coordinateSystem(extent = {{-148.5,-105.0},{148.5,105.0}}, preserveAspectRat
    TwoTanksExample.Design.SystemEnvironment sm_systemenvironment(twoTanksSystem.controller1.ref =
    TwoTanksExample.Scenarios.Change_of_input_flow_1 vs_change_of_input_flow_1;
    TwoTanksExample.Requirements.Max_level_of_liquid_in_tank req_002_0_max_level_of_liquid_in_tank
      (levelInTank = sm_systemenvironment.twoTanksSystem.tank1.levelOfLiquid, tankHeight = sm_sys
    TwoTanksExample.Requirements.Max_level_of_liquid_in_tank req_002_1_max_level_of_liquid_in_tank(
    TwoTanksExample.Requirements.Settling_time_and_bounds_after_a_change_of_input_flow req_004_0_se
    TwoTanksExample.Requirements.Settling_time_and_bounds_after_a_change_of_input_flow req_004_1_se
    TwoTanksExample.Requirements.Volume_of_a_tank req_001_0_volume_of_a_tank(tankVolume = sm_system
    TwoTanksExample.Requirements.Volume_of_a_tank req_001_1_volume_of_a_tank(tankVolume = sm_system
end VeM_Change_of_input_flow_1;
```

Binding stored using
Modelica modifiers

Figure 55: Storing bindings in Modelica

The inferred binding is stored in the first-level component `req_002_0_max_level_of_liquid_in_tank` using Modelica modifiers (see (Modelica, 2013, p. 70)) which enable overwriting values of subcomponents.

```
model VeM__Change_of_input_flow_1
...
... req_002_0_max_level_of_liquid_in_tank (levelInTank =
    sm_systemenvironment.twoTanksSystem.tank1.levelOfLiquid, ...)
...
```

Figure 56: Verification session report GUI and HTML

6.2.6 Generating Verification Session Reports

Figure 56 shows the *verification session report* (both in the GUI tool and the HTML report) generated after executing verification models and post-processing simulation results.

The report is presented for each requirement the verification models used for testing it. In addition, the description of each verification model contains the reference to the system model, scenario used, and for all components the clients, and their bindings. This is the minimum set of information required for the vVDR step *Analyze Verification Results* (see Section 3.7.9).

Note that the report in Figure 56 presents only partial evidence (see Section 4.8). The inference of complete evidence based on partial evidence is straightforward. There is not much of added value in illustrating it in a prototype. In the course of this dissertation project the prototyping of that feature's implementation was skipped.

6.2.7 Traceability Between Requirements, Design, and Scenarios

Figure 57 shows the traceability view in the ModelicaML prototype. It is the implementation of the idea presented in Section 4.3.3. The view lists which requirements are addressed by the selected design alternative (left picture) and can be verified using which scenarios. Or, from a scenario perspective, which scenarios can be used to stimulate the selected system design (right image) and can be used to test which requirements. The illustrated example⁹³ is taken from (Schamai, Fritzson, Paredis, & Helle, 2012).

⁹³ Note, this is not the TwoTanksSystem example, which has too few different scenarios to be able to show the traceability view adequately.

Figure 57: GUI for traceability based on bindings

6.3 Power Plant Cooling System^{94 95}

This section describes a case study that was performed in collaboration with the EDF – R&D department in the frame of the OPENPROD project (ITEA2, 2013). Some content was provided by (Jardin, Bouskela, Nguyen, & Ruel, 2011) and (Jardin, Nguyen, & Ruel, 2011).

This example is focusing on realistic examples of natural-language requirements and their formalization in vVDR. The system under consideration is the SRI system (Intermediate Cooling System, see figure below) which is located in the turbine hall of a nuclear power plant. Its main purpose is to ensure the cooling of certain auxiliary equipment that belongs to the conventional part of the plant and requires demineralized water (e.g. alternator, circulation pump, etc.). The system will provide the following main functions:

- Cooling, ensured by two plate heat exchangers and their control valves, and by one bypass valve;
- Water circulation, ensured by three pumps in parallel, the third one being used as backup in case of the failure of one of the other two pumps, and hence stopped under normal operation conditions;
- Water feed, ensured by one tank and its water supply.

The instrumentation and control functions are:

- Control of the temperature setpoint at the outlet of the heat exchangers;
- Control of the tank water level between two limits;

⁹⁴ Some text is incorporated from (Schamai, Jardin, & Bouskela, 2013).

⁹⁵ The numerical values given here are realistic in the sense that they are coherent from the physical point of view. However, those are merely examples and are not the real values from EDF processes or products.

- Logics of the startup of the backup pump;

Figure 58: Application example 2: Schematic overview

6.3.1 Formalizing Requirements

Assume that we want to verify a system design against the following requirements:

- **Req. 002:** The setpoint of the SRI water temperature must be held at a minimum value of 17 °C.
- **Req. 003:** In normal operating mode, the water temperature of the SRI circuit should be between $T_s - e$ and $T_s + e$ (T_s : setpoint temperature).
- **Req. 0083:** A pump must not start more than 3 times per hour.
- **Req. 013:** In normal operating mode, there must not be less than 2 operating pumps for more than 2s.
- **Req. 007:** The water temperature must not vary more than 10 °C/hour.

Let us start with **Req.002**. Its formalization is straightforward. First we identify quantifiable properties:

#002: The set point of the SRI water temperature must be held at a minimum value of 17° C.

Quantifiable properties:

Figure 59: Application example 2: Req. 002 properties

Now we can express when the requirement is considered to be violated and when not, as follows.

Figure 60: Application example 2: Req. 002 status

In contrast, **Req. 003** has a precondition as shown in Figure 61.

#003: In a normal operating mode, the water temperature of the SRI circuit should be between $T_s - e$ and $T_s + e$ (T_s : set point temperature; e : admissible temperature difference of 1° C).

Figure 61: Application example 2: Req. 003 precondition

Req. 0083 is a bit more complicated. It has no precondition. Further, it is imposed on any pump contained in the design.

“A pump must not start more than 3 times per hour.”

In order to monitor this requirement we need to access values for the past 1 hour. In Modelica we could use the *delay()* operator (Modelica, 2013, p. 23). However, it would need to store values for the last 1 hour, increasing the memory consumption during simulation. Moreover, it would detect the violation 1 hour too late, as illustrated below.

Figure 62: Application example 2: Req. 0083 issue

Instead, to express the violation conditions of this requirement we proceed as follows:

- Store the last 4 activations (timestamps) of a pump
- Compute the time difference between the 1st and the 4th activation
- If the difference is less than 1 h -> *violation*

This approach will detect violations when they happen.

Figure 63: Application example 2: Req. 002 issue resolution

Again, we start by identifying quantifiable properties:

#0083: A pump must not start more than 3 times per hour.

Figure 64: Application example 2: Req. 0083 properties

activationTimePoints is an array of size 4 that is filled with timestamps of the last pump activations. The array is updated each time a pump is activated:

- All entries are shifted by one position up (the first gets deleted)
- The new entry is inserted at the last position

The following figure shows the algorithm and function for managing the sliding window of 4 pump activations.

Figure 65: Application example 2: Req. 0083 functions

Finally, we are able to express when this requirement is violated and when not:

Figure 66: Application example 2: Req. 0083 status

The following figure shows the simulation results when this violation monitor is tested.

Figure 67: Application example 2: Req. 0083 simulation results

The next requirement to be formalized is **Req. 013**. The following figure shows the identified quantifiable properties:

#013: In normal operating mode, there must not be less than 2 operating pumps for more than 2s.

Figure 68: Application example 2: Req. 013 properties

Furthermore, we need to know how many pumps are in the system (`numberOfPumps`) in order to keep track of their activation status represented by `pumpStatus` (array of size equal to the number of pumps) and calculate continuously the number of pumps that are currently active (`numberOfPumpsCurrentlyActive`).

Figure 69: Application example 2: Req. 013 calculations

The implementation of the function `getActivePumpsNumber()` is shown below.

Figure 70: Application example 2: Req. 013 functions

Moreover, we need to detect the time when there are too few pumps in operation (see the algorithm below).

```

/* This evaluation does only work for positive time. */
when numberOfPumpsCurrentlyActive < minNumberOfActivePumps then
  tooFewPumpsDetectedAt := time;
elsewhen numberOfPumpsCurrentlyActive >= minNumberOfActivePumps then
  tooFewPumpsDetectedAt := -1;
/* Note, -1 indicates that there was no detection of too few active pumps. */
end when;
  
```

Figure 71: Application example 2: Req. 013 calculations

Now we are able to compute the status as follows.

```

= if not isNormalOperatingMode then 0
  else if isNormalOperatingMode
    and tooFewPumpsDetectedAt > -1 and (time-tooFewPumpsDetectedAt > maxDuration )
  then 2 else 1
  
```

Figure 72: Application example 2: Req. 013 status

The test results are shown in the figure below.

Figure 73: Application example 2: Req. 013 simulation results

The last requirement, **Req. 007** is approached as follows: Determine the difference between the lowest and the highest temperature value within the 1 hour sliding window. If the difference is more than 10 °C, a violation is detected. The figure below shows the identified properties.

Figure 74: Application example 2: Req. 007 properties

We cannot rely on the continuous temperature function because of sampled sensor values. The temperature needs to be sampled and kept in an array with a sufficient number of temperature values for the past 1 hour. How to compromise on the sample rate? At this point a domain expert must get involved in order to decide the minimum acceptable sampling rate by considering how fast the temperature may change. Assume that the answer of the expert is that it is very unlikely that the temperature will increase and decrease significantly within 10 seconds.

Consequently, it is sufficient to sample with a period of 10 seconds (see figure below). The `samplePoints` is of size 360 and is updated every 10th second with a new value. At that point the oldest value will be removed and the new value added at the top of the array list.

Figure 75: Application example 2: Req. 007 calculations

Finally, we can express the violation by defining the status to be as follows:

```

= if  $\text{abs}(\min(\text{sampledPoints}) - \max(\text{sampledPoints})) > \text{allowedDifference}$ 
  then 2 else 1

```

Figure 76: Application example 2: Req. 007 status

The status is violated as soon as the difference between the min and max values in the `samplePoints` list is more than the allowed difference (i.e., 10 °C).

The next figure shows testing results for this requirement violation monitor. It illustrates how a violation is detected the first time.

Figure 77: Application example 2: Req. 007 simulation results 1

The figure below explains why this requirement is not violated starting from second 5100. It is not violated after second 5100 because within the last 1 h the difference of none of the min and max value pairs is greater than the allowed difference of 10 °C.

Figure 78: Application example 2: Req. 007 simulation results 2

6.3.2 Formalizing Design

The design model was built using ThermoSysPro library⁹⁶ (see also (El Hefni, Bouskela, & Lebreton, 2011)) in a Modelica tool and then imported into ModelicaML environment using the Modelica code synchronization feature (see Section 5.6.2).

⁹⁶ ThermoSysPro library was developed by EDF R&D and is released under an open source license.

Figure 79: Application example 2: System design model import

6.3.3 Formalizing Scenarios

The following scenarios are used for verification:

- sc01 Normal operating mode: No failure, two pumps on, two heat exchangers in operation.
- sc02 Incorrect setting of the temperature regulation.
- sc03 Pump switching: Pumps start several times due to occurrences of sequential failures.
- sc04 Pump recovery: If a pump stops because of a failure, the first available pump starts; if it is not on after 2 seconds, the next available pump is started.
- sc05 Cold winter: Weather outside is too cold, and so the SRI keeps a water temperature above 15 °C.
- sc06 Overheating: The auxiliary equipment delivers more heat than the SRI is sized for.
- sc07 Fast water filling: In case of a malfunction in the automatic feedwater system, the manual valve should allow the water to fill the circuit in less than 2 hours.

The listed scenarios were modeled using the ModelicaML prototype. The figure below shows an example.

Figure 80: Application example 2: Formalized scenarios

6.3.4 Specifying Bindings

Some bindings for requirements are shown below.

Figure 81: Application example 2: Example of requirement bindings

The following figure shows an example of bindings between scenarios and the system design models.

Figure 82: Application example 2: Example of system model and scenario bindings

6.3.5 Generating Verification Models and Verification Results

The following figure shows an example of the verification model generated.

Figure 83: Application example 2: Verification model generated

As pointed out in Section 4.1 when large, complex⁹⁷ system design models are involved, it becomes crucial to reuse instantiations of models once they are created. Figure 83 shows that the number of nodes in the instantiation tree (see Section 4.2.1) in this verification model is ca. 4900. The ModelicaML prototype enhancement for reusing instantiations improved the time for verification model generation from minutes to seconds in this case study.

Furthermore, in this case study additional models are used. An additional model is used that calculates whether the system is in normal mode by analyzing the status of each pump contained in the system. This system is said to be in normal mode if at least 2 pumps are active at the same time.

This information, whether the system is in normal mode or not, is needed by some violation monitor clients. However, there is no component of the system design model that can provide this information. Thus, an additional model was created and related to the system design model in order ensure that the system mode status will be made available for any client (e.g. requirement clients).

Figure 84: Application example 2: Example of an additional model

Figure 83 shows that this addition model was included and is used by, for example, *req_013...* client *isInNormalOperatingMode*.

After simulating all generated verification models, a verification report was generated (shown in Figure 85).

⁹⁷ Models that result in a large number of components when instantiated.

Verification Session Report

Verification Report
Creation date: 2013.01.24 11:57:06

[Violated Requirements \(2 of 7\)](#)
[Not Violated Requirements \(5 of 7\)](#)
[Not Evaluated Requirements \(0 of 7\)](#)
[Not Implemented Requirements \(2 of 7\)](#)
[Used Scenarios \(7 of 7\)](#)
[Not Used Scenarios \(0 of 7\)](#)
[New Positive Relations \(49\)](#)
[New Negative Relations \(0\)](#)
[All Verification Models \(7\)](#)
[Simulated Models \(7 of 7\)](#)
[Not Simulated Models \(0 of 7\)](#)

Violated Requirements

Requirements that were evaluated and violated.
Metric: 2 of 7

003: Water temperature in normal mode

ModelicaMLModel::Requirements::Water temperature in normal mode
Text: In a normal operating mode, the water temperature of the SRI circuit should be above 10°C.

Simulated in the following verification models:

VeM: aVeM: Sc02 Incorrect setting of the temperature regulation

VeM: aVeM: Sc05 Cold winter

ModelicaMLModel::aVeMs: Global_encapsulated_20130124115526::aVeM: Sc05 Cold winter
startTime:0, stopTime:150, tolerance:0.000001, outputFormat:mat

System Model: sm_global_encapsulated
Design::SRIN4_v3::Global_encapsulated

Diagram(graphics)

Client : environment.Command_Valve_FeedWater.K = vs_sc07_fast_w
 Client : environment.Exchanger1Valve.K = vs_sc01_normal_operal
 Client : environment.Exchanger2Valve =
 Client : environment.FeedWater_Valve_Failure =
 Client : environment.Heat_AuxiliaryEquipment.k = vs_sc07_fast_w
 Client : environment.Leak.k = vs_sc07_fast_water_filling.Leak_y_s
 Client : environment.ManualValve_Circulation.k = vs_sc01_normal
 Client : environment.Pump1Failure.K = vs_sc03_pump_switching.f
 Client : environment.Pump2Failure.K = vs_sc04_pump_recovery.P
 Client : environment.Pump3Failure.K = vs_sc03_pump_switching.f
 Client : environment.SetPoint_Temperature.k = vs_sc02_incorrect_setting_of_the_temperature_regulation.S

Scenario: vs_sc05_cold_winter
ModelicaMLModel::Scenarios::Sc05 Cold winter
External weather is too cold so that the SRI keeps a water temperature above 10°C.

Requirement (003): req_003_water_temperature_in_normal_n
ModelicaMLModel::Requirements::Water temperature in normal mode

Client : isInNormalOperatingMode = add_0_NormalModeCalculate
 Client : setPointTemperature = sm_global_encapsulated.enviroi
 Client : waterTemperatureOfSRICircuit = sm_global_encapsulat

Figure 85: Application example 2: Verification session report generated

6.4 Conclusion

The case studies presented have shown the applicability of the vVDR method to industrial problems. ModelicaML has turned out to be a promising implementation to support the vVDR method. However, it is still in a prototype phase and needs to be improved in terms of usability and stability in order to be used in industrial projects.

The definition and reuse of bindings enables on-demand binding code generation and automated model composition. *Bindings do not modify client or provider models.* Moreover, the binding definitions are not affected by the change

of client and provider names or model structure, except when binding operations are used that include references to subcomponents with instance paths.

The binding code can be generated on demand when models need to be integrated—for example, in a verification model. In doing so, the verification models generated become artifacts that are created on demand. This way, each of the roles involved (e.g., requirement analyst, system designer, or tester) can focus on creating the main artifacts and does not need to maintain the verification models.

This paradigm is in line with the overall ModelicaML approach, in which Modelica code is generated from the ModelicaML model only when it is needed for simulations. A ModelicaML model contains all the information needed to regenerate code at any time. This is advantageous because it reduces the number of models that need to be maintained.

However, it might become a disadvantage for a verification session when the generation of a large number of models would take a long time and therefore block the session to start. This possible issue is subject to empirical observations to be addressed in future work, in which large models will be used for design verification following this approach.

Along with the automated composition of models, the added value of this approach is that, when applied to vVDR, it enables traceability of requirements that are implemented in a given system design, identifies scenarios that can be used to verify design against a set of requirements, and supports impact analysis in order to determine which model parts are affected when a component is changed. All this can be achieved by following the dependencies between clients and providers and by determining whether binding code can be generated for a given client.

Chapter 7

Conclusion and Future Work

7.1 Review

This dissertation presents a new approach for the verification of system designs against requirements using models.

Any model is an abstraction of a real system, existing or to be built. Furthermore, any given model may address only a part of the system (e.g. a subsystem) or only a specific aspect (e.g. the dynamic behavior, dysfunctional behavior, behavior from electrical or thermal point of view, etc.). Consequently, each model contains uncertainty that results from the abstraction and simplification of the parts or phenomena that are modeled, and from the incompleteness of the knowledge and context considered. When using the proposed vVDR method, the quality of and the evidence in verification results primarily depend on the accuracy of the predicted system behavior, i.e., on the correctness of system design models. The more precise and complete the system design model is, the more confidence we can have in the verification results produced.

The answer to the question of which requirements can be verified using vVDR in a particular project depends on the design models that are in place or that are planned to be created. By using the proposed formalization approach and the binding concept it is possible to determine whether particular requirements can be verified for a given design. For example, if the system design model does not provide data required for all mandatory clients of the requirement violation monitor, then the requirement is not completely addressed in this design and cannot be evaluated.

In order to determine which scenario can stimulate the system we only need to see if there is one provider from the scenario model that is used by at least one client in the design model. The developed algorithm for automated model composition can then find valid combinations of scenarios and requirements for a given design alternative model and generate verification models.

In system development projects that deal with several design alternatives which evolve throughout the system development process, involving dozens of requirements and a scenario, the approach presented is expected to significantly

improve process efficiency thanks to automation and the fact that there is no need to maintain models that can be re-generated.

The research question stated in Section 1.3 are addressed as follows. Research Question 1 is mainly addressed in Chapter 3 which presents a new method and the main modeling artifacts to be created in order to enable a model-based design verification. Research Question 2 is discussed in Section 3.7.3 which shows how requirements are formalized to be appropriate for design verification. Research Question 3 is addressed in Section 3.8 and elaborated in Chapter 4 showing the possible degree of automation. Chapter 5 presents the answer to Research Question 4 by introducing a new language and tool that support the new design verification approach.

7.2 Major Contributions

This dissertation presents a new approach to design verification using simulation models, including formalized requirements. The main contributions are a new *method* that is supported by a new *language* and a *tool*.

The tasks of verifying system designs against natural-language requirements and, in particular, of formalizing requirements, are nowadays not enabled by the state of the art methods and tools (see Chapter 2). The new design verification method (vVDR, see Chapter 3) enables verification of system dynamic behavior designs against requirements using simulation models. It shows how natural-language requirements and scenarios are formalized. Furthermore, the new design verification method moves the responsibility for defining pass/fail criteria from the tester role to the requirement analyst role (see Section 3.7.2.4) and allows the same scenario to be reused to verify multiple requirements (see discussion in Section 3.7.5).

The method is supported by a new language: ModelicaML (see Chapter 5). ModelicaML integrates UML and Modelica and addresses the need for an integrated hardware and software system modeling and simulation. Engineers can formalize requirements, scenarios, and system design models independently in a concurrent engineering fashion. The proposed framework (see Chapter 4) enables models to be integrated in an automated fashion. The proposed formalization of requirements supports the automated drawing of conclusion from the generated simulation results and sharing of the new knowledge inferred in verification sessions.

The language and the algorithms developed (see Section 4.6 and Section 4.7) for automation are implemented in a prototype that is used to illustrate the applicability of the new method to examples from industry. The case studies presented (see Chapter 6) start with sets of natural-language requirements and show how they are translated into models. Then designs and verification scenarios are modeled, and

simulation models are composed and simulated automatically. The simulation results produced are then used to automatically draw conclusions on requirements violations; this knowledge is shared using semantic web technology.

7.3 Validity

For addressing the research questions and testing the hypothesis formulated in Section 1.3, three⁹⁸ public case studies were conducted (see Chapter 6). They illustrate and explain the ideas and are used to assess the applicability of the developed approach to industrial problems. The case studies were selected to be as simple as necessary in order to be able to get the ideas across, and, at the same time, to be as realistic as necessary in order to reflect adequately industrial problems in terms of different types of natural-language requirements, scenarios, and designs to be modeled, as well as the model size and computational complexity.

The new method to design verification using models (see Chapter 3), the prototype developed (see Chapter 5), and case studies presented (see Chapter 6) show that different types of natural-language system requirements can be formalized, and that models, which adequately reflect complex physical systems, can be used for an automated verification models composition and simulations results evaluation (see Chapter 4) using the approach presented in this dissertation. This gives us a higher confidence that this approach will be applicable to larger industrial problems and will scale in terms of model size without significant performance penalties when analyzing and composing models.

However, the applicability and scalability beyond the presented examples is still an open question. The developed prototype and the case studies presented enable testing of the hypothesis from Section 1.3. The evidence gathered from the case studies is an indication that this approach is promising. However, additional experiments will be necessary to be confident that this approach will be applicable to a wide range of related industrial problems.

7.4 Future Work

The work presented offers further possibilities and applications that are not investigated in this thesis.

The approach for formalizing natural-language requirements could be enhanced as following:

⁹⁸ Two of them are presented in Chapter 6.

- vVDR does not fully prescribe how monitors are to be modeled, as long as they comply with the specification in Section 3.7.3.1. In this dissertation, and in the presented ModelicaML prototype, requirement violation monitors are modeled in an imperative way: i.e., the monitors define *how* to compute the violation status. Typically an imperative way is not as concise as a declarative way.

It is worth investigating whether there is a better way: for example, by enhancing the Modelica language or developing a new domain-specific language for modeling requirement violation monitors in a concise and declarative way.

- Approaches that use natural-language processing, such as presented in (Boddu, Guo, Mukhopadhyay, & Cukic, 2004), can help in extracting quantifiable properties from requirements statements (see Section 3.7.3.2) and assist requirement analyst in modeling requirement violation monitors.
- Requirement violation monitors could be modeled in a parameterized fashion for reuse as templates or library components.
- Moreover, it may be useful to compute differences between or tolerances for expected and provided values by means of, for example, mean squared error. In doing so, the violation monitors may be used for optimization. The status attribute could be used as an optimization constraint, and optimization objectives could be defined for requirement violation monitor attributes.
- The proposed approach could support discovering similar or conflicting requirements within a set of requirements. This idea is sketched in Section 4.3.4.2.

Similar to requirements, the step of formalizing scenarios does not prescribe how scenarios are created. Scenarios can be generated automatically based on Model-Based Testing techniques or created completely manually by experts.

In addition, techniques from knowledge management or optimization can be used to generate scenarios in approaches that are partially automated (e.g. based on scenario templates), fully automated (see Model-Based Testing in Section 2.4.1), or generated in an entirely manual manner by experts. Combining formalized requirements violation monitors and optimization techniques may be of great use in producing sets of scenarios that maximize the number of violated requirements for stressing the system and to ensure robustness of a particular system design alternative.

The verification approach proposed in this dissertation is based on deterministic models. It does not yet take into account uncertainties in system components or scenarios or tolerances in requirements. The approach presented here should be extended to include capabilities to handle uncertainties (e.g. by means of probability distributions for variables) in order to reach conclusions on the probability of violations using Monte Carlo simulation.

Similar, the proposed method may be enhanced to leverage statistical model checking (see (Legay & Delahaye, 2010), (Sen, Viswanathan, & Agha, 2004) and (Zuliani, Platzer, & Clarke, 2010)) that is based on Monte Carlo simulation, in order to determine whether a property holds with a certain probability. The specification of properties to be checked would be based on the requirement violation monitor status values. This approach will be particularly interesting for checking models that include failure behavior (see example in (Zuliani, Platzer, & Clarke, 2010)).

The technology presented is based on UML and Modelica. However, the approach presented that can be extended to allow the use of different specialized tools for creating the main artifacts (i.e., requirements, system designs, and scenarios). The Functional Mock-up Interface standard (Modelica Association Project “FMI”, 2012) could be used to exchange or integrate models. The ModelicaML tool could be enhanced to import FMI based models (similar to the existing Modelica models synchronization feature; see Section 5.6.2) and the binding concept (see Section 4.2) will enable to automatically compose models from Functional Mock-Up Units (FMUs). Such enhancements will widen the range of applications for the proposed approach in industrial projects.

Appendix A

ModelicaML Profile

This appendix presents all stereotypes of the ModelicaML UML profile. ModelicaML is designed to enable model creation using standardized graphical UML notation and their translation into executable Modelica code.

The following subsections represent the structure of the ModelicaML profile, each containing extensions of the UML meta-model. The last section proposes enhancements for the concrete syntax used, i.e., graphical notation for UML diagrams.

Profile Version: **1.3.2**
Note: In each section the stereotypes are sorted alphabetically.

A.1 Class Constructs

Figure 86: ModelicaML class constructs

Stereotype: «Block»

Description: This stereotype is represents the Modelica restricted class "block".

Extends the UML metaclass: **Class**

Inherits the following properties from the

stereotype «**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

Stereotype: «CalculationModel»

Description: A CalculationModel does not represent a physical system part. When instantiated, it calculates values that are used by other components.

Inherits the following properties from the stereotype «**Model**»: encapsulated, partial, final, replaceable, iconImagePath

Stereotype: «Connector»

Description: This stereotype represents the Modelica restricted class "connector". A "connector" cannot contain equation or algorithm sections, but it can have causality and flow specification. All ports must be typed by a "connector" class.

Extends the UML metaclass: **Class**

Inherits from the following properties from the

stereotype «**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

Property: **expandable** (multiplicity of entries: 1..1), type: Boolean.

Stereotype: «Function»

Description: This stereotype represents the Modelica restricted class "function".

Extends the UML metaclass: **FunctionBehavior**

Inherits the following properties from the

stereotype «**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

Property: **externalLanguageSpecification** (multiplicity of entries: 0..1), type: String.

Property: **externalFunctionCall** (multiplicity of entries: 0..1), type: String.

Property: **externalAnnotation** (multiplicity of entries: 0..1), type: String.

Property: **scope** (multiplicity of entries: 0..1), type: ModelicaScope (inner, outer).

Stereotype: «Model»

Description: This stereotype represents the Modelica restricted class "model".

Extends the UML metaclass: **Class**

Inherits the following property from the stereotype

«**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

Stereotype: «ModelicaClass»

Description: This stereotype represents the Modelica "class".

Extends the UML metaclass: **Class**

Inherits the following properties from the stereotype

«**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

Stereotype: «ModelicaClassDefinition» (is abstract)

Description: This stereotype has attributes that are common to all Modelica restricted classes.

Property: **encapsulated** (multiplicity of entries: 1..1), type: Boolean.

Property: **partial** (multiplicity of entries: 1..1), type: Boolean.

Property: **final** (multiplicity of entries: 1..1), type: Boolean.

Property: **replaceable** (multiplicity of entries: 1..1), type: Boolean.

Property: **iconImagePath** (multiplicity of entries: 0..1), type: String. Property

description: This is the path to the icon image relative to the location of the model file.

Stereotype: «ModelicaEnumeration»

Extends the UML metaclass: **Enumeration**

Inherits the following properties from the stereotype

«**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable,
iconImagePath

Stereotype: «ModelicaPackage»

Description: This stereotype represents the Modelica restricted class "package".

Extends the UML metaclass: **Class**

Inherits the following properties from the stereotype

«**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable,
iconImagePath

Stereotype: «Operator»

Description: This stereotype is represents the Modelica restricted class "operator".

Extends the UML metaclass: **Class**

Inherits the following properties from the stereotype

«**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable,
iconImagePath

Stereotype: «Record»

Description: This stereotype represents the Modelica restricted class "record".

Extends the UML metaclass: **Class**

Inherits the following properties from «**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

Stereotype: «Type»

Description: This stereotype is used to represent the Modelica restricted class "type".

Extends the UML metaclass: **PrimitiveType**

Inherits the following properties from stereotype

«**ModelicaClassDefinition**»: encapsulated, partial, final, replaceable, iconImagePath

A.2 Composite Constructs

Figure 87: ModelicaML composite constructs

Stereotype: «CalculatedProperty»

Description: A **CalculatedProperty** does not represent a physical system part. This component provides other components with calculated values.

Inherits the following properties from the stereotype «**Component**»: final, modification, arraySize, replaceable, iconImagePath, conditionalExpression

Stereotype: «Component»

Description: This stereotype is used to represent a Modelica component of a composite type.

Extends the UML metaclass: **Property**

Inherits the following properties from stereotype «**InstanceOfAClass**»: final, modification, arraySize, replaceable, iconImagePath, conditionalExpression

Stereotype: «Connection»

Description: This stereotype is used to represent the Modelica "connection" clause and implies its associated semantics (Kirchhoff's laws).

Extends the UML metaclass: **Connector**

Property: **explicitConnectionEnds** (multiplicity of entries: 0..1), type: String. Property description: This property can be used to define the content of a Modelica connect clause. The keyword "connect" and the parentheses are omitted. The string must contain paths to both ports (using the dot notation) separated by a comma. Example: component1.port1.variable1 , component2.port2.variable2

Stereotype: «ConnectionPort»

Description: This stereotype is used to represent a Modelica class instance of the type "connector".

Extends the UML metaclass: **Port**

Inherits the following properties from stereotype «**InstanceOfClass**»: final, modification, arraySize, replaceable, iconImagePath, conditionalExpression

Property: **causality** (multiplicity of entries: 0..1), type: ModelicaCausality (input, output).

Stereotype: «**FunctionArgument**»

Description: This stereotype is used to represent Modelica “function” arguments.

Extends the UML metaclass: **Parameter**

Inherits the following properties from stereotype «**InstanceOfClass**»: final, modification, arraySize, replaceable, iconImagePath, conditionalExpression

Property: **flowFlag** (multiplicity of entries: 0..1), type: ModelicaFlowFlag (flow, stream).

Property: **scope** (multiplicity of entries: 0..1), type: ModelicaScope (inner, outer).

Property: **causality** (multiplicity of entries: 1..1), type: ModelicaCausality (input, output).

Property: **variability** (multiplicity of entries: 1..1), type: ModelicaVariability (continuous, discrete, parameter, constant).

Property: **declarationEquationOrAssignment** (multiplicity of entries: 0..1), type: String. Property description: This refers to the shorthand notation in Modelica. The result must be of the same type as the Modelica component itself. In this case the value is the right-hand expression of the equation or assignment. The "=" or "!=" sign must be included.

Stereotype: «**InstanceOfClass**» (is abstract)

Description: This stereotype has attributes that are common to all Modelica class instances.

Property: **final** (multiplicity of entries: 1..1), type: Boolean.

Property: **modification** (multiplicity of entries: 0..*), type: String. Property description: Each entry represents one instance of a modification of type "instance name = modification expression". The parenthesis "(" and ")" and the commas are

omitted.

Property: **arraySize** (multiplicity of entries: 0..*), type: String. Property description: Each entry string contains the full specification of the array: for example, "[1,3,:]".

Property: **replaceable** (multiplicity of entries: 1..1), type: Boolean.

Property: **conditionalExpression** (multiplicity of entries: 0..1), type: String. Property description: Only if the condition is true then the Modelica component is created; the condition is only evaluated at compilation time and is often used for configuration changes. The result of the expression must be of type Boolean. This string contains the expression. The keyword "if" is omitted.

Property: **iconImagePath** (multiplicity of entries: 0..1), type: String. Property description: This is the path to the icon image relative to the location of the model file. If no icon is defined for a property then the icon of the property type is displayed (if defined, otherwise no icon is displayed).

Stereotype: «Variable»

Description: This stereotype is used to represent a Modelica class instance of a primitive type (Real, String, Boolean, Integer) or a subtype of the Modelica restricted class "type".

Extends the UML metaclass: **Property**

Inherits the following properties from stereotype «**InstanceOfA Class**»: final, modification, arraySize, replaceable, iconImagePath, conditionalExpression

Property: **flowFlag** (multiplicity of entries: 0..1), type: ModelicaFlowFlag (flow, stream).

Property: **scope** (multiplicity of entries: 0..1), type: ModelicaScope (inner, outer).

Property: **causality** (multiplicity of entries: 0..1), type: ModelicaCausality (input, output).

Property: **variability** (multiplicity of entries: 1..1), type: ModelicaVariability (continuous, discrete, parameter, constant).

Property: **declarationEquationOrAssignment** (multiplicity of entries: 0..1), type: String. Property description: This refers to the shorthand notation in Modelica. The result must be of the same type as the Modelica component itself. In this case the value is the right-hand expression of the equation or assignment. The "=" or "!=" sign must be included.

A.3 Behavior Constructs

Figure 88: ModelicaML constructs for capturing Modelica code

Figure 89: ModelicaML constructs for state machine

Figure 90: ModelicaML constructs for conditional equations or algorithm

Stereotype: «Algorithm(Code)»

Description: This stereotype is used to represent a set of statements in an algorithm section.

Extends the UML metaclass: **OpaqueBehavior**

Inherits the following properties from stereotype «**AlgorithmLoopProperties**»:
loopKind, forIndices, whileExpression

Property: **initial** (multiplicity of entries: 1..1), type: Boolean.

Stereotype: «AlgorithmLoopProperties» (is abstract)

Description: This stereotype has properties that are inherited by other stereotypes in order to be able to capture the "for" or a "while" loop with the "for-indices" or "while expression", respectively.

Property: **loopKind** (multiplicity of entries: 0..1), type: ModelicaAlgorithmLoopKind (for, while).
 Property: **forIndices** (multiplicity of entries: 0..1), type: String.
 Property: **whileExpression** (multiplicity of entries: 0..1), type: String.

Stereotype: «Assert»

Description: This stereotype represents the Modelica assert() function.

Extends the UML metaclass: **Constraint**

Property: **condition** (multiplicity of entries: 1..1), type: String.
 Property: **message** (multiplicity of entries: 1..1), type: String.
 Property: **level** (multiplicity of entries: 1..1), type: ModelicaAssertionLevel (warning, error).

Stereotype: «ConditionalAlgorithm(Diagram)»

Description: This stereotype is used to represent an algorithm section. An Activity Diagram is used to model the conditional constructs (loop, if, when). Any OpaqueActions, in such an Activity Diagram, must contain Modelica statements in the "body" attribute.

Extends the UML metaclass: **Activity**

Inherits the following properties from stereotype «**AlgorithmLoopProperties**»:
 loopKind, forIndices, whileExpression

Property: **isSubActivity** (multiplicity of entries: 1..1), type: Boolean. Property description: This attribute indicates that this activity is a subactivity. It is invoked by another activity using CallBehaviorAction.

Property: **initial** (multiplicity of entries: 1..1), type: Boolean.

Stereotype: «ConditionalEquations(Diagram)»

Description: This stereotype is used to represent an equation section. An Activity Diagram is used to model the conditional constructs (loop, if, when). Any OpaqueActions, in such an Activity Diagram, must contain Modelica equations in the "body" attribute.

Extends the UML metaclass: **Activity**

Inherits the following properties from stereotype «**EquationsLoopProperties**»: loopKind, forIndices

Property: **isSubActivity** (multiplicity of entries: 1..1), type: Boolean. Property description: This attribute indicates that this activity is a subactivity. It is invoked by another activity using CallBehaviorAction.

Property: **initial** (multiplicity of entries: 1..1), type: Boolean.

Stereotype: «Equations(Code)»

Description: This stereotype is used to represent a set of equations in an equation section.

Extends the UML metaclass: **OpaqueBehavior**

Inherits the following properties from stereotype «**EquationsLoopProperties**»: loopKind, forIndices

Property: **initial** (multiplicity of entries: 1..1), type: Boolean.

Stereotype: «EquationsLoopProperties» (is abstract)

Description: This stereotype has properties that are inherited by other stereotypes in order to be able to capture the "for" loop with the "for-indices".

Property: **loopKind** (multiplicity of entries:

0..1), type: ModelicaEquationsLoopKind (for).

Property: **forIndices** (multiplicity of entries: 0..1), type: String.

Stereotype: «ExecutionOrderPriority»

Description: “ExecutionOrderPriority” specifies the order in which the code for the transition, control flow, or region will be executed relative to the neighbors.

Extends the UML metaclass: **Transition, ControlFlow, Region**

Property: **priority** (multiplicity of entries: 1..1), type: Integer. Property description: Execution priority in code. A lower number implies a higher execution priority. Element with the lowest priority is executed first.

Stereotype: «If»

Description: This stereotype is used to represent the Modelica "if" keyword.

Extends the UML metaclass: **DecisionNode**

Stereotype: «ModelicaMLStateMachine»

Description: This stereotype indicates that this is a special case of the UML StateMachine. It uses a limited subset of the UML StateMachine metaclasses and can be translated into executable Modelica code.

Extends the UML metaclass: **StateMachine**

Property: **samplingExpression** (multiplicity of entries: 0..1), type: String.

Stereotype: «StateSpecification»

Description: This stereotype defines additional features that are not included in UML.

Extends the UML metaclass: **State, StateMachine**

Property: **freezeExpression** (multiplicity of entries: 0..1), type: String. Property description: Indicates whether the state or state machine should be active but does not react and does not execute actions as long as the expression evaluates to true.

Stereotype: «StateTransitionSpecification»

Description: This stereotype captures additional features for state transition behavior that are not included in UML.

Extends the UML metaclass: **Transition**

Property: **suppressExitActions** (multiplicity of entries: 1..1), type: Boolean. Property description: If set to true then exit actions of the transition source state and its substates are not executed.

Property: **suppressEntryActions** (multiplicity of entries: 1..1), type: Boolean. Property description: If set to true then entry actions of the transition target state (not its substates) are not executed.

Stereotype: «When»

Description: This stereotype is used to represent the Modelica "when" keyword.

Extends the UML metaclass: **DecisionNode**

A.4 Relations Constructs

Figure 91: ModelicaML relations constructs

Stereotype: «ConstrainedByRelation»

Description: This stereotype represents the Modelica relation “constrainedBy”.

Extends the UML metaclass: **Dependency**

Property: **modification** (multiplicity of entries: 0..*), type: String.

Stereotype: «ExtendsRelation»

Description: This stereotype has attributes which allow the capturing of type modifications.

Extends the UML metaclass: **Generalization**

Property: **modification** (multiplicity of entries: 0..*), type: String. Property description: Each entry represents a Modelica type modification. Commas and the parentheses "(" and ")" are omitted.

Stereotype: «Import»

Description: This stereotype represents the Modelica “import” relation.

Extends the UML metaclass: **Dependency**

Property: **alias** (multiplicity of entries: 0..1), type: String.

Stereotype: «PartialDerivativeOffFunctionRelation»

Description: This stereotype represents the Modelica relation from a function (source) that is a derivative of another function (target).

Extends the UML metaclass: **Dependency**

Property: **arguments** (multiplicity of entries: 1..*), type: String.

Stereotype: «RedeclareRelation»

Description: This stereotype represents the Modelica “redeclare” relation.

Extends the UML metaclass: **Dependency**

Property: **modification** (multiplicity of entries: 0..*), type: String.

Stereotype: «TypeRelation»

Description: This stereotype has properties that allow capturing type modifications. It can only be used between classes of primitive type.

Extends the UML metaclass: **Generalization**

Property: **modification** (multiplicity of entries: 0..*), type: String. Property description: Each entry represents a Modelica type modification. Commas and the parentheses "(" and ")" are omitted.

Property: **arraySize** (multiplicity of entries: 0..*), type: String. Property description: Each string contains the full array specification (e.g. "[1,3,:]")

A.5 Requirement Constructs

Figure 92: ModelicaML requirement constructs

Stereotype: «Requirement»

Description: The “Requirement” concept, identified by this stereotype, represents a natural-language requirement and the corresponding violation monitor.

Extends the UML metaclass: **Class**

Inherits the following properties from stereotype

«**ModelicaClassDefinition**»: `encapsulated`, `partial`, `final`, `replaceable`, `iconImagePath`

Property: **id** (multiplicity of entries: 1..1), type: String. Property description: Unique requirement identifier.

Property: **text** (multiplicity of entries: 1..1), type: String. Property description: Natural-language statement of the requirement.

Stereotype: «RequirementInstance»

Description: This stereotype indicates that this component is an instantiation of a requirement.

Inherits the following properties from stereotype «**Component**»: final, modification, arraySize, replaceable, iconImagePath, conditionalExpression

A.6 Simulation Constructs

Figure 93: ModelicaML simulation construct

Stereotype: «Simulation»

Description: This stereotype contains the properties which are used in the "simulate()" command in Modelica tools.

Extends the UML metaclass: **Class**

- Property: **startTime** (multiplicity of entries: 0..1), type: String.
- Property: **stopTime** (multiplicity of entries: 0..1), type: String.
- Property: **numberOfIntervals** (multiplicity of entries: 0..1), type: String.
- Property: **tolerance** (multiplicity of entries: 0..1), type: String.

A.7 Annotation Constructs

Figure 94: ModelicaML annotation constructs

Stereotype: «Annotation(CodeGeneration)»

Description: Annotation specific to Modelica code generation (not ModelicaML code generation).

Extends the UML metaclass: **Comment**

Property: **Evaluate** (multiplicity of entries: 0..1), type: Boolean.

Property: **HideResult** (multiplicity of entries: 0..1), type: Boolean.

Property: **Inline** (multiplicity of entries: 0..1), type: Boolean.

Property: **LateInline** (multiplicity of entries: 0..1), type: Boolean.

Property: **smoothOrder** (multiplicity of entries: 0..1), type: Integer.

Stereotype: «Annotation(Documentation)»

Description: See the Modelica specification.

Extends the UML metaclass: **Comment**

Property: **info** (multiplicity of entries: 0..1), type: String.

Property: **revisions** (multiplicity of entries: 0..1), type: String.

Stereotype: «Annotation(Icon)»

Description: Graphical icon annotation (Modelica code).

Extends the UML metaclass: **Comment**

Property: **iconAnnotationString** (multiplicity of entries: 0..1), type: String.

Stereotype: «Annotation(SimulationExperiment)»

Description: The experiment annotation defines the default start time, stop time, and default relative integration tolerance for simulation experiments to be carried out with the model at hand.

Extends the UML metaclass: **Comment**

Property: **StartTime** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Property: **StopTime** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Property: **Tolerance** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Stereotype: «Annotation(VersionHandling)»

Description: A top-level package or model can specify the version of top-level classes it uses, its own version number, and if possible how to convert from previous versions. This can be used by a tool to guarantee that consistent versions are used, and if possible to upgrade usage from an earlier version to a current one.

Extends the UML metaclass: **Comment**

Property: **version** (multiplicity of entries: 0..*), type: String. Property description: Defines the version number of the model or package. All classes within this top-level class have this version number.

Property: **conversion** (multiplicity of entries: 0..*), type: String. Property description: The first "(" and the last ")" are omitted.

Property: **uses** (multiplicity of entries: 1..1), type: String. Property description: The first "(" and the last ")" are omitted.

Stereotype: «Annotation»

Description: Annotations are intended for storing extra information about a model, such as graphics, documentation or versioning, etc. A Modelica tool is free to define and use other annotations, in addition to those defined here, according to section 14.1. of the Modelica specification. The only requirement is that any tool will save files with all annotations from this chapter and all vendor-specific annotations intact. To ensure this, annotations must be represented with constructs according to the Modelica grammar. The specification in this document defines the semantic meaning if a tool implements any of these annotations.

Extends the UML metaclass: **Comment**

Property: **fullAnnotationString** (multiplicity of entries: 0..1), type: String. Property

description: This is a qualified annotation string. The first "(" and the last ")" are omitted.

A.8 Verification

Figure 95: ModelicaML constructs for design verification

Stereotype: «DoNotUseToVerify»

Description: This relation indicates that the scenario (source) should not be used to verify the requirement it points to.

Extends the UML metaclass: **Dependency**

Stereotype: «RequiredFor»

Description: This stereotype is the opposite of the *requires* relation. The two stereotypes are mutually exclusive and should be used depending on the desired direction of the dependency and the resulting modification of the dependency client.

Inherits the following properties from stereotype **«Requires»**:
onlyInCombinationWith, always

Stereotype: «Requires»

Description: This dependency indicates that when the client model is instantiated it requires the target model to be instantiated as well.

Extends the UML metaclass: **Dependency**

Property: **always** (multiplicity of entries: 1..1), type: Boolean. Property description: This property means that the target model will always be instantiated together with the dependency client model, regardless of whether the providers for the target model are used by other components or not.

Stereotype: «UseToVerify»

Description: This relation indicates that the scenario (source) can be used to verify the requirement (target).

Extends the UML metaclass: **Dependency**

Stereotype: «VerificationModel»

Description: A verification model is composed of one design that is to be verified against a set of requirements by means of one scenario. Some additional models may need to be included as well.

Extends the UML metaclass: **Class**

Stereotype: «VerificationScenario»

Description: VerificationScenario is a model that stimulates the design in such a way that requirement violation monitors are evaluated.

Extends the UML metaclass: **Class**

Inherits the following properties from stereotype «**Simulation**»: startTime, stopTime, numberOfIntervals, tolerance

A.9 Bindings

Figure 96: ModelicaML bindings constructs

Stereotype: «ObtainsValueFrom»

Description: This relation specifies that the referenced mediator (source) will be used for inferring binding from the referenced provider (target).

Extends the UML metaclass: **Dependency**

Inherits the following properties from stereotype «**ValueComputation**»: operation

Stereotype: «ProvidesValueFor»

Description: This relation specifies that the referenced mediator (source) will be used for inferring binding for the referenced client (target).

Extends the UML metaclass: **Dependency**

Inherits the following properties from stereotype «**ValueComputation**»: operation

Property: **isRequired** (multiplicity of entries: 1..1), type: Boolean. Property description: If set to true, this property means that it is mandatory to bind this client, even if a default value is specified.

Stereotype: «ValueComputation» (is abstract)

Description: Clients, mediators and providers can have associated operations.

Property: **operation** (multiplicity of entries: 0..1), type: String.

Stereotype: «ValueMediator»

Description: A mediator enables binding code generation for clients.

Extends the UML metaclass: **Property**

Inherits the following properties from stereotype «**ValueComputation**»: operation

Property: **variability** (multiplicity of entries: 1..1), type: ModelicaVariability (continuous, discrete, parameter, constant).

Property: **preferredProviders** (multiplicity of entries: 0..*), type: String.

Stereotype: «ValueMediatorsContainer»

Description: A container that groups mediators.

Extends the UML metaclass: **Class**

A.10 Model References

Figure 97: ModelicaML constructs for Modelica code synchronization

Stereotype: «InstalledLibrary»

Description: This stereotype indicates that this element is a library that is available in the ModelicaML environment and can be imported and used.

Extends the UML metaclass: **Model**

Stereotype: «ModelicaModelProxy»

Description: This stereotype represents an existing Modelica model. It reflects all information of that model that is needed for its usage.

Extends the UML metaclass: **Element**

A.11 ModelicaPredefinedTypes

Primitive Type: ModelicaBoolean

Property: **quantity** (multiplicity of entries: 0..1), type: String.

Property: **fixed** (multiplicity of entries: 0..1), type: Boolean.

Property: **start** (multiplicity of entries: 0..1), type: Boolean.

Primitive Type: ModelicaInteger

Property: **quantity** (multiplicity of entries: 0..1), type: String.

Property: **min** (multiplicity of entries: 0..1), type: Integer.

Property: **max** (multiplicity of entries: 0..1), type: Integer.

Property: **fixed** (multiplicity of entries: 0..1), type: Boolean.

Property: **start** (multiplicity of entries: 0..1), type: Integer.

Primitive Type: ModelicaReal

Property: **quantity** (multiplicity of entries: 0..1), type: String.

Property: **unit** (multiplicity of entries: 0..1), type: String.

Property: **displayUnit** (multiplicity of entries: 1..1), type: String.

Property: **min** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Property: **max** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Property: **fixed** (multiplicity of entries: 0..1), type: Boolean.

Property: **nominal** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Property: **start** (multiplicity of entries: 0..1), type: UnlimitedNatural.

Property: **stateSelect** (multiplicity of entries: 0..1), type: ModelicaStateSelect
(default, never, avoid, prefer, always).

Primitive Type: ModelicaString

Property: **quantity** (multiplicity of entries: 0..1), type: String.

Property: **start** (multiplicity of entries: 0..1), type: String.

A.12 ModelicaPredefinedEnumerations

Enumeration **LogicalOperators** (and, or, not)
 Enumeration **ModelicaAccess** (public, protected)
 Enumeration **ModelicaAlgorithmLoopKind** (for, while)
 Enumeration **ModelicaAssertionLevel** (warning, error)
 Enumeration **ModelicaCausality** (input, output)
 Enumeration **ModelicaEquationsLoopKind** (for)
 Enumeration **ModelicaFlowFlag** (flow, stream)
 Enumeration **ModelicaScope** (inner, outer)
 Enumeration **ModelicaStateSelect** (default, never, avoid, prefer, always)
 Enumeration **ModelicaVariability** (continuous, discrete, parameter, constant)

A.13 Enhancements of UML Graphical Notation

Here some enhancements for representing ModelicaML data on UML diagrams are proposed.

Figure 98 contains proposed enhancements for the class notation. This proposal tailors the notation for visualizing Modelica-specific data by distinguishing between constants, parameters, primitive variables and components and displaying the restricted class stereotype for the class and for each composite component. Shorthand notation for keywords *protected* and *input/output* is adopted from UML. Empty compartments should not be displayed.

Neither in UML nor in Modelica does a diagram indicate whether or not all information that can appear on that diagram is displayed. The icon for *incomplete view* in Figure 99 is crucial in order to know that this is only a partial view and there are more elements or diagrams to consider. Domain-specific icons and port causality (see Figure 99) also contribute to faster apperception of the information.

Figure 100, Figure 101, and Figure 102 present a concise notation for loop and conditional equations or statements. The new notation also includes the priority for conditional control flow on diagrams (similar to the discussion for state transition priorities in Section 5.7.3.5).

Legend:

- # = protected
- ~ = flow
- in = input
- out = output

«model» Name {encapsulated}
constants
parameters
variables # ~ in name : type name out name : type name
components name : «model» type name name : «block» type name
ports name : type name name : type name

Figure 98: Proposed class notation

Figure 99: Proposed notation for the indication of incomplete view

Figure 100: Proposed notation for capturing “for loop”⁹⁹ statements

⁹⁹ “While” statements are captured the same way, except that the UML decision node would have the «while-statement» stereotype applied

Figure 101: Proposed notation for capturing “if”¹⁰⁰ statements¹⁰¹

¹⁰⁰ “When” statements are captured the same way, except that the UML decision node would have the «when» stereotype applied.

¹⁰¹ The code example is from (Fritzson, 2004, p. 292).

Figure 102: Proposed notation for capturing if¹⁰² equations¹⁰³

¹⁰² “When” equations are captured the same way, except that the UML decision node would have the «when» stereotype applied

¹⁰³ The code example is from (Fritzson, 2004, p. 570).

Appendix B

Grammar for Binding Operations

This appendix presents the grammar for binding operations. The grammar¹⁰⁴ presented here extends Modelica's concrete syntax grammar in order to support new concepts not provided by Modelica. The grammar below is implemented in the ModelicaML prototype editors.

Client Operation

```
ClientOperation :
  (left_hand=left_hand_component_reference '=' Expr=expression
  ';'*)*;

left_hand_component_reference :
  ('.')? ref=IDENT (subscripts1=array_subscripts)?
  ('.' ref1+=IDENT (subscripts+=array_subscripts)?)*;

primary :
  num=UNSIGNED_NUMBER
  | int=INT
  | str=STRING
  | Bool=BOOL_VAL
  | Name_Function=name Function
  | Initial_ref=initial_ref
  | Expr=ExprDer

  | binding=GetBindingFunction

  | Component_reference=component_reference
  | '(' output_expr_list=output_expression_list ')'
  | '[' Expr_list=expression_list ( ';'
  Expression_list+=expression_list)* ']'
```

¹⁰⁴ The grammar language is explained in (The Eclipse Foundation - Xtext Project, 2013).

```

| '{' f_arguments=function_arguments '}'
| End='end'
;

GetBindingFunction :
'getBinding' Brackets;

Brackets :
'(' ')';

component_reference:
ref=IDENT (subscripts1=array_subscripts)?
('.' refl+=IDENT (subscripts+=array_subscripts)?)*
| ('.')? ref=IDENT (subscripts1=array_subscripts)?
('.' refl+=IDENT (subscripts+=array_subscripts)?)* ;

```

Mediator Operation

```

MediatorOperation :
(Expr=expression)?;

primary :
    num=UNSIGNED_NUMBER
    | int=INT
    | str=STRING
    | Bool=BOOL_VAL
    | Name_Function=name_Function
    | Initial_ref=initial_ref
    | Expr=ExprDer

    | sum=SumFunction
| prod=ProductFunction
    | min=MinFunction
| max=MaxFunction
    | avg=AverageFunction
    | size=SizeFunction
    | toArray=ToArrayFunction
    | and=ANDFunction
    | or=ORFunction
    | xor=XORFunction

    | singleProvider=GetSingleProviderFunction

    | Component_reference=component_reference
    | '(' output_expr_list=output_expression_list ')'
    | '[' Expre_list=expression_list ( ';'
Expression_list+=expression_list)* ']'
    | '{' f_arguments=function_arguments '}'

```

```

    | End='end'
;

//ModelicaML mediator operation reduction function macros
ProductFunction :
    'product' ReductionFunctionCallArgs;
SumFunction :
    'sum' ReductionFunctionCallArgs;
MinFunction :
    'min' ReductionFunctionCallArgs;
MaxFunction :
    'max' ReductionFunctionCallArgs;
AverageFunction :
    'avg' ReductionFunctionCallArgs;
SizeFunction :
    'size' ReductionFunctionCallArgs;
ToArrayFunction :
    'toArray' ReductionFunctionCallArgs;
ANDFunction :
    'AND' ReductionFunctionCallArgs;
ORFunction :
    'OR' ReductionFunctionCallArgs;
XORFunction :
    'XOR' ReductionFunctionCallArgs;
GetSingleProviderFunction :
    'getSingleProvider' Brackets;

ReductionFunctionCallArgs :
    '(' + ':' + ')';
Brackets :
    '(' ')';

component_reference:
    ref=IDENT (subscripts1=array_subscripts)? ('.' ref1+=IDENT
    (subscripts+=array_subscripts)?) *
    | ('.')? ref=IDENT (subscripts1=array_subscripts)? ('.'
    ref1+=IDENT
    (subscripts+=array_subscripts)?) *;

```

Provider Operation

```

ProviderOperation :
    (Expr=expression)?;

```


Appendix C

Ontology for Verification Report

This appendix includes the definition of the vVDR verification ontology and examples of SPARQL queries that enable filtering of data. The proposed ontology can be used to capture and share knowledge generated in verification sessions, as well as to reason about the new knowledge for obtaining particular views on the available data.

vVDR Verification Ontology

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
  <!ENTITY owl2 "http://www.w3.org/2006/12/owl2#" >
  <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
  <!ENTITY swrlx "http://www.w3.org/2003/11/swrlx#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <!ENTITY vvdr
"http://www.openmodelica.org/modelicaml/ontology/vvdr#" >
]>

<rdf:RDF xmlns="http://www.openmodelica.org/modelicaml/ontology/vvdr#"
  xml:base="http://www.openmodelica.org/modelicaml/ontology/vvdr"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:owl2="http://www.w3.org/2006/12/owl2#"
  xmlns:vvdr="http://www.openmodelica.org/modelicaml/ontology/vvdr#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:swrlx="http://www.w3.org/2003/11/swrlx#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology
rdf:about="http://www.openmodelica.org/modelicaml/ontology/vvdr"/>
```

```

<!--
////////////////////////////////////
// Datatypes
////////////////////////////////////
-->

<!-- http://www.w3.org/1999/02/22-rdf-syntax-ns#text -->
<rdfs:Datatype rdf:about="&rdf;text"/>

<!--
////////////////////////////////////
// Object Properties
////////////////////////////////////
-->
<!-- http://www.openmodelica.org/modelicaml/vvdr#consideredDesign -->

<owl:ObjectProperty rdf:about="&vvdr;consideredDesign">
  <rdfs:range rdf:resource="&vvdr;DesignAlternative"/>
  <rdfs:domain rdf:resource="&vvdr;Evaluation"/>
</owl:ObjectProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#consideredRequirement -
->

<owl:ObjectProperty rdf:about="&vvdr;consideredRequirement">
  <rdfs:domain rdf:resource="&vvdr;Evaluation"/>
  <rdfs:range rdf:resource="&vvdr;Requirement"/>
  <owl:inverseOf rdf:resource="&vvdr;evaluated"/>
</owl:ObjectProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#consideredScenario -->

<owl:ObjectProperty rdf:about="&vvdr;consideredScenario">
  <rdfs:domain rdf:resource="&vvdr;Evaluation"/>
  <rdfs:range rdf:resource="&vvdr;Scenario"/>
</owl:ObjectProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#evaluated -->

<owl:ObjectProperty rdf:about="&vvdr;evaluated">
  <rdfs:range rdf:resource="&vvdr;Evaluation"/>
  <rdfs:domain rdf:resource="&vvdr;Requirement"/>
</owl:ObjectProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#usedModel -->

<owl:ObjectProperty rdf:about="&vvdr;usedModel">
  <rdfs:range rdf:resource="&vvdr;DesignVerification"/>
  <rdfs:domain rdf:resource="&vvdr;VerificationResult"/>
  <owl:inverseOf rdf:resource="&vvdr;usedToProduce"/>
</owl:ObjectProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#usedToProduce -->

<owl:ObjectProperty rdf:about="&vvdr;usedToProduce">
  <rdfs:domain rdf:resource="&vvdr;DesignVerification"/>
  <rdfs:range rdf:resource="&vvdr;VerificationResult"/>
</owl:ObjectProperty>

```



```

<!--
////////////////////////////////////
// Data properties
////////////////////////////////////
-->

<!-- http://www.openmodelica.org/modelicaml/ontology/vvdr#name -->

  <owl:DatatypeProperty
rdf:about="http://www.openmodelica.org/modelicaml/ontology/vvdr#name">
    <rdfs:domain rdf:resource="&vvdr;Model"/>
    <rdfs:range rdf:resource="&xsd:string"/>
  </owl:DatatypeProperty>

<!-- http://www.openmodelica.org/modelicaml/ontology/vvdr#text -->

  <owl:DatatypeProperty
rdf:about="http://www.openmodelica.org/modelicaml/ontology/vvdr#text">
    <rdfs:domain rdf:resource="&vvdr;Requirement"/>
    <rdfs:range rdf:resource="&xsd:string"/>
  </owl:DatatypeProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#ID -->

  <owl:DatatypeProperty rdf:about="&vvdr;ID">
    <rdfs:domain rdf:resource="&vvdr;Requirement"/>
    <rdfs:range rdf:resource="&xsd:string"/>
  </owl:DatatypeProperty>

<!-- http://www.openmodelica.org/modelicaml/vvdr#qualifiedName -->

  <owl:DatatypeProperty rdf:about="&vvdr;qualifiedName">
    <rdfs:domain rdf:resource="&vvdr;Model"/>
    <rdfs:range rdf:resource="&rdf;text"/>
  </owl:DatatypeProperty>

<!--
////////////////////////////////////
// Classes
////////////////////////////////////
-->

<!-- http://www.openmodelica.org/modelicaml/vvdr#DesignAlternative -->

  <owl:Class rdf:about="&vvdr;DesignAlternative">
    <rdfs:subClassOf rdf:resource="&vvdr;Model"/>
  </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#DesignVerification -->

  <owl:Class rdf:about="&vvdr;DesignVerification">
    <rdfs:subClassOf rdf:resource="&vvdr;Model"/>
  </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#Evaluation -->

```

```

    <owl:Class rdf:about="&vvdr;Evaluation"/>

<!-- http://www.openmodelica.org/modelicaml/vvdr#Model -->
    <owl:Class rdf:about="&vvdr;Model"/>

<!-- http://www.openmodelica.org/modelicaml/vvdr#NotEvaluatedRequirement
-->

    <owl:Class rdf:about="&vvdr;NotEvaluatedRequirement">
        <rdfs:subClassOf rdf:resource="&vvdr;VerificationResult"/>
    </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#NotViolatedRequirement
-->
    <owl:Class rdf:about="&vvdr;NotViolatedRequirement">
        <rdfs:subClassOf rdf:resource="&vvdr;VerificationResult"/>
    </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#Requirement -->
    <owl:Class rdf:about="&vvdr;Requirement">
        <rdfs:subClassOf rdf:resource="&vvdr;Model"/>
    </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#Scenario -->
    <owl:Class rdf:about="&vvdr;Scenario">
        <rdfs:subClassOf rdf:resource="&vvdr;Model"/>
    </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#VerificationResult -->
    <owl:Class rdf:about="&vvdr;VerificationResult">
        <rdfs:subClassOf rdf:resource="&vvdr;Evaluation"/>
    </owl:Class>

<!-- http://www.openmodelica.org/modelicaml/vvdr#ViolatedRequirement -->
    <owl:Class rdf:about="&vvdr;ViolatedRequirement">
        <rdfs:subClassOf rdf:resource="&vvdr;VerificationResult"/>
    </owl:Class>
</rdf:RDF>

```

Examples of SPARQL Queries

For each SPARQL query listed in the following sections the header is the following:

```

PREFIX vvdr: <http://www.openmodelica.org/modelicaml/ontology/vvdr#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX owl2: <http://www.w3.org/2006/12/owl2#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrl: <http://www.w3.org/2003/11/swrl#>
PREFIX swrlb: <http://www.w3.org/2003/11/swrlb#>
PREFIX swrlx: <http://www.w3.org/2003/11/swrlx#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

```

The resulting figures in the following section were produced using the tool (Dodds, 2013).

Violated Requirements (Partial Evidence) Query

```
SELECT ?reqName ?designName ?scenarioName

WHERE {
  {
    #individuals
    ?requirement rdf:type vvdr:Requirement.
    ?requirement vvdr:evaluated ?evaluation.
    ?evaluation rdf:type vvdr:ViolatedRequirement.
    ?evaluation vvdr:consideredDesign ?design.
    ?evaluation vvdr:consideredScenario ?scenario.

    #names
    ?requirement vvdr:name ?reqName.
    ?design vvdr:name ?designName.
    ?scenario vvdr:name ?scenarioName.
  }
}
```

reqName	designName	scenarioName
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_5 [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Draining tank [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_4 [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_1 [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_6 [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_3 [^] http://www.w3.org/2001/XMLSchema#string
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_2 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_1 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_6 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_3 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_4 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_2 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_5 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_1 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_2 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_4 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_5 [^] http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow [^] http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_3 [^] http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_6 [^] http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_5 [^] http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank [^] http://www.w3.org/2001/XMLSchema#string	SystemEnvironment [^] ...	Change of input flow_4 [^] http://www.w3.org/2001/XMLSchema#string

Figure 103: SPARQL query 1 result for "Violated Requirements" view

Using GROUP BY we get only the requirements.

```
SELECT ?reqName ?designName ?scenarioName

WHERE {
  {
```

```
#individuals
?requirement rdf:type vvdr:Requirement.
?requirement vvdr:evaluated ?evaluation.
?evaluation rdf:type vvdr:ViolatedRequirement.
?evaluation vvdr:consideredDesign ?design.
?evaluation vvdr:consideredScenario ?scenario.

#names
?requirement vvdr:name ?reqName.
?design vvdr:name ?designName.
?scenario vvdr:name ?scenarioName.
}
}
#show only requirement names
GROUP BY ?reqName
```

reqName	design
Max level of liquid in tank [^] http://www.w3.org/2001/XMLSchema#string	
Settling time and bounds after a change of input flow [^] http://www.w3.org/2001/XMLSchema#string	
Volume of a tank [^] http://www.w3.org/2001/XMLSchema#string	

Figure 104: SPARQL query 2 result for "Violated Requirements" view

Not Violated Requirements (Partial Evidence)

```
SELECT ?reqName ?designName ?scenarioName

WHERE {
  {
    #individuals
    ?requirement rdf:type vvdr:Requirement.
    ?requirement vvdr:evaluated ?evaluation.
    ?evaluation rdf:type vvdr:NotViolatedRequirement.
    ?evaluation vvdr:consideredDesign ?design.
    ?evaluation vvdr:consideredScenario ?scenario.

    #names
    ?requirement vvdr:name ?reqName.
    ?design vvdr:name ?designName.
    ?scenario vvdr:name ?scenarioName.
  }
}
```

reqName	designName	scenarioName
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_5~http://www.w3.org/2001/XMLSchema#string
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Draining tank~http://www.w3.org/2001/XMLSchema#string
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_1~http://www.w3.org/2001/XMLSchema#string
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_2~http://www.w3.org/2001/XMLSchema#string
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_6~http://www.w3.org/2001/XMLSchema#string
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_3~http://www.w3.org/2001/XMLSchema#string
Volume of a tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_4~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Draining tank~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_2~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_1~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_6~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_3~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_2~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_4~http://www.w3.org/2001/XMLSchema#string
Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_1~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_1~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_5~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_2~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_6~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_3~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Draining tank~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_4~http://www.w3.org/2001/XMLSchema#string
Input flow limits~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow~http://www.w3.org/2001/XMLSchema#string

Figure 105: SPARQL query result for "Not Violated Requirements" view

Not Evaluated Requirements (Partial Evidence)

```

SELECT ?reqName ?designName ?scenarioName

WHERE {
  {
    #individuals
    ?requirement rdf:type vvd:Requirement.
    ?requirement vvd:evaluated ?evaluation.
    ?evaluation rdf:type vvd:NotEvaluatedRequirement.
    ?evaluation vvd:consideredDesign ?design.
    ?evaluation vvd:consideredScenario ?scenario.

    #names
    ?requirement vvd:name ?reqName.
    ?design vvd:name ?designName.
    ?scenario vvd:name ?scenarioName.
  }
}

```

reqName	designName	scenarioName
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_6~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_5~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_4~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_3~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_2~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow_1~http://www.w3.org/2001/XMLSchema#string
System draining time~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow~http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Draining tank~http://www.w3.org/2001/XMLSchema#string
Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string	SystemEnvironment~http://www.w3.org/2001/XMLSchema#string	Change of input flow~http://www.w3.org/2001/XMLSchema#string

Figure 106: SPARQL query result for "Not Evaluated Requirements" view

Scenarios for Stimulating Designs

Note that by using *DISTINCT* we eliminate duplicates.

```

SELECT DISTINCT ?scenarioName ?designName

WHERE {
  {

```

```
#individuals
?evaluation rdf:type vvdr:Evaluation.
?evaluation vvdr:consideredDesign ?design.
?evaluation vvdr:consideredScenario ?scenario.

#names
?requirement vvdr:name ?reqName.
?design vvdr:name ?designName.
?scenario vvdr:name ?scenarioName.
}
}
```

scenarioName	designName
Draining tank^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow_5^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow_6^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow_3^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow_4^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow_1^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string
Change of input flow_2^^http://www.w3.org/2001/XMLSchema#string	SystemEnvironment^^http://www.w3.org/2001/XMLSchema#string

Figure 107: SPARQL query result for "Scenarios for Stimulating Design" view

Scenarios for Evaluating Requirements

Here we are looking only for scenarios which lead to a violation or do not lead to a violation of requirements. All scenarios that lead to a *not evaluated* status of requirements are excluded because such scenarios most likely do not stimulate the system properly and are not appropriate for verifying requirements.

Note that by using *DISTINCT* we eliminate duplicates.

```
SELECT DISTINCT ?scenarioName ?reqName

WHERE {
  {
    #individuals
    ?evaluation rdf:type vvdr:ViolatedRequirement.
    ?evaluation vvdr:consideredDesign ?design.
    ?evaluation vvdr:consideredScenario ?scenario.
    ?evaluation vvdr:consideredRequirement ?req.

    #names
    ?req vvdr:name ?reqName.
    ?design vvdr:name ?designName.
    ?scenario vvdr:name ?scenarioName.
  }
  UNION
  {
    #individuals
    ?evaluation rdf:type vvdr:NotViolatedRequirement.
    ?evaluation vvdr:consideredDesign ?design.
    ?evaluation vvdr:consideredScenario ?scenario.
    ?evaluation vvdr:consideredRequirement ?req.
```

```

#names
?req vvdr:name ?reqName.
?design vvdr:name ?designName.
?scenario vvdr:name ?scenarioName.
}
}

```

scenarioName	reqName
Change of input flow_5~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_1~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow_6~http://www.w3.org/2001/XMLSchema#string	Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_4~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Draining tank~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_4~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_6~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow_3~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow_4~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow_1~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_3~http://www.w3.org/2001/XMLSchema#string	Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_2~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow_6~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_3~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_5~http://www.w3.org/2001/XMLSchema#string	Setting time and bounds after a change of input flow~http://www.w3.org/2001/XMLSchema#string
Change of input flow_5~http://www.w3.org/2001/XMLSchema#string	Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_2~http://www.w3.org/2001/XMLSchema#string	Volume of a tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_2~http://www.w3.org/2001/XMLSchema#string	Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_1~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Change of input flow_1~http://www.w3.org/2001/XMLSchema#string	Max level of liquid in tank~http://www.w3.org/2001/XMLSchema#string
Change of input flow_5~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Change of input flow_2~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Change of input flow_6~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Change of input flow_3~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Draining tank~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Change of input flow_4~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string
Draining tank~http://www.w3.org/2001/XMLSchema#string	System draining time~http://www.w3.org/2001/XMLSchema#string
Change of input flow~http://www.w3.org/2001/XMLSchema#string	Input flow limits~http://www.w3.org/2001/XMLSchema#string

Figure 108: SPARQL query result for "Scenarios for Evaluating Requirements" view

Bibliography

- Ambriola, V., & Gervasi, V. (1997). Processing Natural Language Requirements. In *Automated Software Engineering. Proceedings., 12th IEEE International Conference* (pp. 36-45). IEEE.
- Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., et al. (2005). Combining Test Case Generation and Runtime Verification. In *Theoretical Computer Science 336, no. 2* (pp. 209-23).
- Baier, C., & Katoen, J.-P. (2008). *Principles of Model Checking*. MIT press.
- Balci, O. (1995). Principles and Techniques of Simulation Validation, Verification, and Testing. In *Simulation Conference Proceedings, 1995. Winter* (pp. 147-154). IEEE.
- Ball, T. (1999). The Concept of Dynamic Analysis. In *Software Engineering—ESEC/FSE '99* (pp. 216-234). Springer Berlin Heidelberg.
- Boddu, R., Guo, L., Mukhopadhyay, S., & Cukic, B. (2004). RETNA: From Requirements to Testing in a Natural Way. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International* (pp. 262-271). IEEE.
- Carloni, L. P., Roberto, P., & Pinto, A. (2006). *Languages and Tools for Hybrid Systems Design*. Now Publishers Inc.
- Cellier, F. E., & Greifeneder, J. (1991). *Continuous System Modeling*. Springer.
- Cellier, F. E., & Kofman, E. (2006). *Continuous System Simulation*. Springer.
- Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model Checking*. MIT press.
- Crane, M. L., & Dingel, J. (2005). UML vs. Classical vs. Rhapsody Statecharts: Not all Models are Created Equal. In *Proc. 8th International Conf. on Model Driven Engineering Languages and Systems* (pp. 97-112). Springer Berlin Heidelberg.
- Dassault Systèmes Dymola. (2013). *CATIA Systems Engineering – Dymola*. Retrieved April 23, 2013, from <http://www.3ds.com/products/catia/portfolio/dymola>
- Denning, P., Comer, D., Gries, D., Mulder, M., Tucker, A., Turner, A., et al. (1989). Computing as a Discipline. In *Communications of the ACM 32, no. 1* (pp. 9-23).
- Dijkstra, E. (1972). The Humble Programmer. In *Communications of the ACM 15, no. 10* (pp. 859-866).

- Dodds, L. (2013). *Twinkle: A SPARQL Query Tool*. Retrieved Mai 15, 2013, from www.ldodds.com/projects/twinkle/
- Dromey, R. G. (2003). From Requirements to Design: Formalizing the Key Steps. In *Software Engineering and Formal Methods* (pp. 2-11). IEEE.
- El Hefni, B., Bouskela, D., & Lebreton, G. (2011). Dynamic Modelling of a Combined Cycle Power Plant with ThermoSysPro. In *Proceedings 8th Modelica Conference*.
- Elmqvist, H., Gaucher, F., Mattsson, S., & Dupont, F. (2012). State Machines in Modelica. In *Proceedings of 9th International Modelica Conference* (pp. 3-5). Munich, Germany.
- ESA. (1996). *ECSS-E-10A Space Engineering*. ESA.
- Estefan, J. A. (2007). *Survey of Model-Based Systems Engineering (MBSE) Methodologies*. Incoose MBSE Focus Group 25.
- Federal Aviation Authority DO-178B. (1999). *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc.
- Ferreira, J. A., & de Oliveira, E. J. (1999). Modelling Hybrid Systems using Statecharts and Modelica. In *Emerging Technologies and Factory Automation, 1999. Proceedings. ETFA'99. 1999 7th IEEE International Conference on*, vol. 2 (pp. 1063-1069). IEEE.
- Firesmith, D. (2003). Specifying Good Requirements. In *Journal of Object Technology 2.4* (pp. 77-87). ETH Zurich.
- Fritzson, P. (2004). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press.
- Fuhrmann, H., & von Hanxleden, R. (2010). Taming Graphical Modeling. In *Model Driven Engineering Languages and Systems* (pp. 196-210). Springer Berlin Heidelberg.
- Gamma, E., Helm, R., & Johnson, R. E. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*. Amsterdam: Addison-Wesley Longman.
- Gause, D. C., & Weinberg, G. M. (1989). *Exploring Requirements: Quality before Design*. New York: Dorset House Pub.
- Hammal, Y. (2005). A Formal Semantics of Uml Statecharts by Means of Timed Petri Nets. In *Formal Techniques for Networked and Distributed Systems-FORTE 2005* (pp. 38-52). Springer Berlin Heidelberg.
- Harel, D. (1987). A Visual Formalism for Complex Systems. In *Science of Computer Programming 8 no. 3* (pp. 231-274).
- Harel, D., & Kugler, H. (2004). The Rhapsody Semantics of Statecharts (or, on the Executable Core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering* (pp. 325-354). Springer Berlin Heidelberg.
- Hazelrigg, G. A. (2012). *Fundamentals of Decision Making*.
- Helle, P., Strobel, C., Mitschke, A., Rivière, A., & Vincent, L. (2008). Improving Systems Specifications a Method Proposal. In *Proceedings of CSER 2008 Conference*.
- Hull, E., Jackson, K., & Dick, J. (2005). *Requirements Engineering*. Springer.

- IBM. (2013, Mai 06). *IBM® Rational® Rhapsody® Designer for Systems Engineers*. Retrieved Mai 06, 2013, from www-01.ibm.com/software/rational/products/rhapsody/designer
- IEEE Std 610.12-1990. (1999). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE.
- IEEE1220. (2005). *IEEE Standard for Application and Management of the Systems Engineering Process*. IEEE.
- INCOSE. (2006). *Systems Engineering Handbook* (Version 3 ed.). INCOSE.
- INCOSE. (2007). *Systems Engineering Vision 2020* (INCOSE-TP-2004-004-02. Version 2.03 ed.). INCOSE.
- INCOSE. (2013). *INCOSE Requirements Management Tools Survey*. Retrieved April 19, 2013, from www.incose.org/products/pubs/products/rmsurvey.aspx
- ISO 9000. (2005). *ISO 9000 - Quality Management*. ISO.
- ISO/IEC 15288. (2008). *Systems and Software Engineering - System Life Cycle Processes* (Second edition ed.). ISO.
- ISO/IEC TR 19760. (2003). *Systems Engineering — A Guide for the Application of ISO/IEC 15288 (System Life Cycle Processes)*. ISO.
- ISO 9000. (2005). *ISO 9000 - Quality Management*. ISO.
- ITEA2. (2013). *OPENPROD - Open Model-Driven Whole-Product Development and Simulation Environment*. Retrieved Mai 16, 2013, from <http://www.itea2.org/project/index/view/?project=1132>
- Jardin, A., Bouskela, D., Nguyen, T., & Ruel, N. (2011). Modelling of System Properties in a Modelica Framework. In *Proceedings of 8th International Modelica Conference 2011*. Dresden, Germany.
- Jardin, A., Nguyen, T., & Ruel, N. (2011). *EUROSYSLIB Project – sWP7.1 – Properties modeling*. EDF technical report, H-P1C-2011-00913-EN.
- Johnson, T. A., Kerzhner, A., Paredis, C. J., & Burkhart, R. (2012). Integrating Models and Simulations of Continuous Dynamics into SysML. In *Transactions of the ASME-S-Computing and Information Science in Engineering* 12(1).
- Lee, E. A. (2008). Cyber Physical Systems: Design Challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium* (pp. 363-369). IEEE.
- Legay, A., & Delahaye, B. (2010). Statistical Model Checking: An Overview. In *Runtime Verification* (pp. 122-135). Springer Berlin Heidelberg.
- Leucker, M., & Schallhart, C. (2009). A Brief Account of Runtime Verification. In *Journal of Logic and Algebraic Programming* 78, no. 5 (pp. 293-303).
- Levy, J., Hassen, S., & Uribe, T. E. (2002). Combining Monitors for Runtime System. In *Electronic Notes in Theoretical Computer Science* 70, no. 4 (pp. 112-127).
- Liang, F., Schamai, W., Rogovchenko, O., Sadeghi, S., Nyberg, M., & Fritzson, P. (2012). Model-Based Requirement Verification: A Case Study. In *9th International Modelica Conference (Modelica'2012)*. Munich, Germany.

- Loniewski, G., Insfran, E., & Abrahao, S. (2010). A Systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development. In *Model Driven Engineering Languages and Systems* (pp. 213-227). Springer Berlin Heidelberg.
- Loucopoulos, P., & Karakostas, V. (1995). *System Requirements Engineering*. McGraw-Hill, Inc.
- Malmheden, M., Elmqvist, H., Mattsson, S., Henriksson, D., & Otter, M. (2008). ModeGraph-A Modelica Library for Embedded Control Based on Mode-Automata. In *Proc. of Modelica'2008 Conference*. Bielefeld.
- Marc-Florian, W., Ina, S., & Alain, V.-F. (2011). Requirements-Driven Testing with Behavior Trees. In *Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 501-510). IEEE.
- Mich, L., Mariangela, F., & Pierluigi, N. I. (2004). Market Research for Requirements Analysis Using Linguistic Tools. In *Requirements Engineering 9, no. 1* (pp. 40-56).
- Modelica. (2013). *Modelica specification, Version 3.3*. Modelica Association.
- Modelica Association. (2012). *Modelica® - A Unified Object-Oriented Language for Systems Modeling*. Retrieved Sept. 25, 2012, from www.modelica.org
- Modelica Association Project "FMI". (2012, August). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. Retrieved June 10, 2013, from www.fmi-standard.org
- Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing*. Wiley.
- Myers, T., Schamai, W., & Fritzson, P. (2011). Comodeling Revisited: Execution of Behavior Trees in Modelica. In *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Zurich, Switzerland.
- NASA. (2007). *NASA Systems Engineering Handbook* (NASA/SP-2007-6105 Rev1 ed.).
- OMG. (2013). *Object Management Group (OMG®)*. Retrieved April 23, 2013, from www.omg.org
- OMG MOFM2T. (2013). *MOF Model To Text Transformation Language (MOFM2T)*, 1.0. Retrieved April 29, 2013, from www.omg.org/spec/MOFM2T/1.0/
- OMG ReqIF. (2011). *Requirements Interchange Format (ReqIF)*. Retrieved Sept. 14, 2013, from www.omg.org/spec/ReqIF/
- OMG SyM. (2012). *SysML-Modelica (SyM) Transformation*. Retrieved Sept. 25, 2013, from www.omg.org
- OMG SysML. (2012). *OMG Systems Modeling Language (OMG SysML™)*. Retrieved Sept. 25, 2013, from www.omgsysml.org
- OMG UML. (2011). *OMG Unified Modeling Language™ (OMG UML), Superstructure*. Retrieved Sept. 25, 2013, from www.omg.org
- OMG UTP. (2012). *UML Testing Profile (UTP)*. Retrieved Sept. 25, 2013, from www.omg.org
- Open Source Modelica Consortium. (2013). *OpenModelica Project*. Retrieved April 19, 2013, from www.openmodelica.org

- Otter, M., Malmheden, M., Elmqvist, H., Mattsson, S. E., & Johnsson, C. (2009). A New Formalism For Modeling of Reactive and Hybrid Systems. In *proceedings of the 7th Modelica'2009 Conference*. Como, Italy.
- Paredis, C., Bernard, Y., Burkhart, R. M., de Koning, H.-P., Friedenthal, S., Fritzson, P., et al. (2010). An Overview of the SysML- Modelica Transformation Specification. In *2010 INCOSE International Symposium*.
- Pop, A., Akhvlediani, D., & Fritzson, P. (2007). Towards Unified System Modeling with the ModelicaML UML Profile. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'07)* (pp. 13-24).
- RTCA/DO-254. (2000). *Design Assurance Guidance for Airborn Electronic Hardware*. RTCA.
- Ryan, K. (1993). The Role of Natural Language in Requirements Engineering. In *In Requirements Engineering, 1993., Proceedings of IEEE International Symposium* (pp. 240-242). IEEE.
- SAE. (1996). *ARP 4754 Certification Considerations for Highly-Integrated or Complex Aircraft Systems*. SAE.
- Schamai, W. (2009). *Modelica Modeling Language (ModelicaML): A UML Profile for Modelica. Technical Report*. Linköping University Electronic Press.
- Schamai, W. (2013). *ModelicaML - UML Profile for Modelica*. Retrieved April 23, 2013, from www.openmodelica.org/modelicaml
- Schamai, W., Fritzson, P., & Paredis, C. (2013). Translation of UML State Machines to Modelica: Handling Semantic Issues. In *SIMULATION* (Volume 89 Issue 4 ed., pp. 498 - 512).
- Schamai, W., Fritzson, P., Paredis, C., & Helle, P. (2012). ModelicaML Value Bindings for Automated Model Composition. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*.
- Schamai, W., Fritzson, P., Paredis, C., & Pop, A. (2009). Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. In *Proceedings 7th Modelica Conference*. Como.
- Schamai, W., Helle, P., Fritzson, P., & Paredis, C. (2011). Virtual Verification of System Designs against System Requirements. In *Models in Software Engineering* (pp. 75-89). Springer Berlin Heidelberg.
- Schamai, W., Jardin, A., & Bouskela, D. (2013). *Industrial Use Cases for Requirements Verification and Model Composition in ModelicaML*. Linköping, Sweden: 7th MODPROD Workshop on Model-Based Product Development.
- Seemann, J. (1997). Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams. In *Graph Drawing* (pp. 415-424). Springer Berlin Heidelberg.
- Sen, K., Viswanathan, M., & Agha, G. (2004). Statistical Model Checking of Black-Box Probabilistic Systems. In *Computer Aided Verification* (pp. 202-215). Springer Berlin Heidelberg.

- Simons, A. (2000). On the Compositional Properties of UML Statechart Diagrams. In *Rigorous Object-Oriented Methods (ROOM 2000)*, Workshops in Computing, BCS.
- Stanford Center for Biomedical Informatics Research. (2013). *protégé*. Retrieved Mai 15, 2013, from www.protege.stanford.edu
- Sterman, J. D. (2002). All Models are Wrong: Reflections on Becoming a Systems Scientist. In *System Dynamics Review* 18, no. 4 (pp. 501-531).
- The Eclipse Foundation - Acceleo Project. (2013). *Acceleo - Transforming Models into Code*. Retrieved April 23, 2013, from www.eclipse.org/acceleo
- The Eclipse Foundation - EMF Project. (2013). *EMF Validation Framework*. Retrieved April 23, 2013, from www.eclipse.org/modeling/emf/?project=validation
- The Eclipse Foundation - Papyrus Project. (2013). *Papyrus*. Retrieved April 23, 2013, from www.eclipse.org/papyrus
- The Eclipse Foundation - Xtext Project. (2013). *Xtext Grammar Language*. Retrieved June 14, 2013, from www.eclipse.org/Xtext/documentation.html#grammarLanguage
- The Eclipse Foundation. (2013). *Eclipse*. Retrieved April 24, 2013, from www.eclipse.org
- Von der Beeck, M. (1994). A Comparison of Statecharts Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems* (pp. 128-148). Springer Berlin Heidelberg.
- W3C. (2013). *OWL Implementations*. Retrieved Mai 15, 2013, from www.w3.org/2007/OWL/wiki/Implementations
- W3C. (2013). *OWL WEB ONTOLOGY LANGUAGE CURRENT STATUS*. Retrieved Mai 15, 2013, from www.w3.org/standards/techs/owl#w3c_all
- W3C. (2013). *Semantic Web*. Retrieved Mai 15, 2013, from www.w3.org/standards/semanticweb/
- W3C. (2013). *SPARQL Current Status*. Retrieved Mai 15, 2013, from www.w3.org/standards/techs/sparql#w3c_all
- W3C. (2013). *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Retrieved Mai 06, 2013, from www.w3.org/TR/scxml/
- W3C. (2013). *XSLT Current Status*. Retrieved Mai 15, 2013, from www.w3.org/standards/techs/xslt#w3c_all
- W3C XML. (2013). *XML Technology*. Retrieved Mai 15, 2013, from www.w3.org/standards/xml/
- Wolfram Research. (2013). *System Modeler*. Retrieved April 02, 2013, from www.wolfram.com/system-modeler
- Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal Methods: Practice and Experience. In *ACM Computing Surveys (CSUR)* 41, no. 4 (pp. 1-40).
- Younes, H. (2005). Verification and Planning for Stochastic Processes with Asynchronous Events. In *Ph.D. Thesis No. CMU-CS-05-105*. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

Zuliani, P., Platzer, A., & Clarke, E. M. (2010). Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control* (pp. 243-252). ACM.

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science
Linköping Studies in Statistics
Linköpings Studies in Informatics

Linköping Studies in Science and Technology

- No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.
- No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.
- No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.
- No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.
- No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.
- No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.
- No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.
- No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.
- No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.
- No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.
- No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.
- No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.
- No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.
- No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.
- No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.
- No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.
- No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.
- No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.
- No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.
- No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.
- No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

- No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.
- No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.
- No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.
- No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.
- No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.
- No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.
- No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.
- No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.
- No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.
- No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.
- No 270 **Ralph Rönquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.
- No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.
- No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.
- No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.
- No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.
- No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.
- No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.
- No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.
- No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.
- No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.
- No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

- No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.
- No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.
- No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.
- No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.
- No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.
- No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.
- No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.
- No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.
- No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.
- No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.
- No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.
- No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.
- No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.
- No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.
- No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.
- No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.
- No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.
- No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.
- No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.
- No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.
- No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.
- No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.
- No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.
- No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.
- No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.
- No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.
- No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.
- No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.
- No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.
- No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.
- No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.
- No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.
- No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.
- No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.
- No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.
- No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.
- No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.
- No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.
- No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.
- No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.
- No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.
- No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.
- No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.
- No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.
- No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

- No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.
- No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.
- No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.
- No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.
- No 688 **Marcus Bjärelund:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.
- No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.
- No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.
- No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.
- No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.
- No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.
- No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.
- No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.
- No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.
- No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.
- No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.
- No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.
- No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.
- No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.
- No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.
- No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.
- No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.
- No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.
- No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.
- No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.
- No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems' Development, 2003, ISBN 91-7373-604-X.
- No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.
- No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.
- No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.
- No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.
- No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.
- No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.
- No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.
- No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.
- No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.
- No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.
- No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.
- No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.
- No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.
- No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5.
- No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.
- No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.
- No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.
- No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.
- No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.
- No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

- No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.
- No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.
- No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.
- No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.
- No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.
- No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.
- No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.
- No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.
- No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.
- No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.
- No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.
- No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.
- No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.
- No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.
- No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.
- No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.
- No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.
- No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.
- No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.
- No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.
- No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.
- No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.
- No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.
- No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.
- No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.
- No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.
- No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.
- No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.
- No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.
- No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.
- No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.
- No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.
- No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.
- No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.
- No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.
- No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.
- No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.
- No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensive Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.
- No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.
- No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.
- No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.
- No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.
- No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.
- No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.
- No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.
- No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

- No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.
- No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.
- No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.
- No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.
- No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.
- No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.
- No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.
- No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.
- No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.
- No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.
- No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.
- No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.
- No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.
- No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.
- No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.
- No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.
- No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.
- No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.
- No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.
- No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.
- No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.
- No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.
- No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.
- No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.
- No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.
- No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.
- No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.
- No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.
- No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.
- No 1359 **Jana Rambusch:** Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.
- No 1373 **Sonia Sangari:** Head Movement Correlates to Focus Assignment in Swedish, 2011, ISBN 978-91-7393-154-0.
- No 1374 **Jan-Erik Källhammer:** Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.
- No 1375 **Mattias Eriksson:** Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.
- No 1381 **Ola Leifler:** Affordances and Constraints of Intelligent Decision Support for Military Command and Control - Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.
- No 1386 **Soheil Samii:** Quality-Driven Synthesis and Optimization of Embedded Control Systems, 2011, ISBN 978-91-7393-102-1.
- No 1419 **Erik Kuiper:** Geographic Routing in Intermittently-connected Mobile Ad Hoc Networks: Algorithms and Performance Models, 2012, ISBN 978-91-7519-981-8.
- No 1451 **Sara Stymne:** Text Harmonization Strategies for Phrase-Based Statistical Machine Translation, 2012, ISBN 978-91-7519-887-3.
- No 1455 **Alberto Montebelli:** Modeling the Role of Energy Management in Embodied Cognition, 2012, ISBN 978-91-7519-882-8.
- No 1465 **Mohammad Saifullah:** Biologically-Based Interactive Neural Network Models for Visual Attention and Object Recognition, 2012, ISBN 978-91-7519-838-5.
- No 1490 **Tomas Bengtsson:** Testing and Logic Optimization Techniques for Systems on Chip, 2012, ISBN 978-91-7519-742-5.
- No 1481 **David Byers:** Improving Software Security by Preventing Known Vulnerabilities, 2012, ISBN 978-91-7519-784-5.
- No 1496 **Tommy Färnqvist:** Exploiting Structure in CSP-related Problems, 2013, ISBN 978-91-7519-711-1.
- No 1503 **John Wilander:** Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.
- No 1506 **Magnus Ingmarsson:** Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.

No 1547 **Wladimir Schamai:** Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.

Linköping Studies in Arts and Science

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Sub-daily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturerings- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.