An Integrated Development Environment with Enhanced Domain-Specific Interactive Model Validation

by

Roland Samlaus
Yes, we have to divide up our time like that, between our politics and our equations. But to me our equations are far more important, for politics are only a matter of present concern.
A mathematical equation stands forever.

______________________________
Albert Einstein
Abstract

Modelling of physical systems plays an important role in many engineering domains. For engineers it is important to model real world objects in order to simulate and assess their behavior. This is particularly true for the development of new product components such as, for example, wind turbines. The suitability and durability of components needs to be investigated early in the design process to avoid costs that arise when faulty designs appear during component testing.

Another domain where models are frequently used is model-driven software development. Data structures and data flow are modeled, for example using UML diagrams. Code is automatically generated from the models and is refined by user-written code. Advantages of model-driven software development are the high abstraction of the code by visualization in diagrams and specialized views which display details of design aspects. Additionally, code generators are usually well tested and design patterns are used in the generated code which leads to good quality and readability.

Models for simulation of wind turbines are developed at the Fraunhofer Institute for Wind Energy and Energy System Technology. Models with different levels of detail are used during the design process, starting with simple models for basic design decisions and evaluations to high-detail models (such as Computational Fluid Dynamics (CFD) models for detailed flow analysis of rotor blades). The models defined are parameter-based and do not define a tool specific syntax. Hence, these models must be transferred to a tool-specific format for simulation. Design flaws that are encountered in low-level models have an impact on the detailed models. But equally important, problems encountered in highly detailed models may require changes in the basic models. This leads to the problem that all models that have been defined for a design study, including the transformation to simulation tool models, may need to be altered. To remedy this problem, model-driven software development is used. Models are created once in our simulation environment and simulation models for the different tools are derived from those models.

This dissertation focuses on the development of components with different levels of detail with the language Modelica. In order to develop models quickly, the developer needs to be assisted in creating models that are valid
regarding language compliance and structural constraints and that show the expected behaviour during simulation.

The Modelica Integrated Development Environment (IDE) OneModelica is introduced that provides a data model for Modelica models with the same technology as the general wind turbine models described above. Hence, Modelica code can be generated from the parametric models using existing tools. Features like project-based development allow separating models with different levels of detail, allowing the user to focus on one topic and to create libraries for certain functionality.

Interactive model validation has been implemented to assist the user during development. Hence, the user gets immediate feedback whether his models are correct according to the syntax and semantic rules defined by the language specification. Furthermore, additional semantic information and rules can be defined in a generic way to restrict the way models are composed. This can help identify design errors that can be hard to detect when investigating the code only.

OneModelica allows to create valid Modelica models and to utilize arbitrary Modelica-compliant simulation tools. Simulation results can be investigated inside the IDE and an automatic test framework enables test-driven development and ensures that models behave as expected. The implementation heavily uses model-driven software development and thus shows that it is possible to create powerful IDEs for complex domain specific languages such as Modelica.

This work has been financially supported by the Federal Ministry for the Environment, Nature Conservation and Nuclear Safety based on a decision of the Parliament of the Federal Republic of Germany, by the Swedish Government in the ELLIIT project and by VR, and by the Swedish governmental agency Vinnova in the RTSIM and ITEA2 MODRIO projects.
Fysikalisk modellering i betydelsen matematisk modellering av fysikaliska system spelar en viktig roll inom många ingenjörsdiscipliner. För ingenjörer är det viktigt att modellera verkliga objekt i syfte att simulera och bedöma deras beteende. Detta gäller i synnerhet vid utveckling av nya komponenter till komplexa produkter som exempelvis vindkraftverk. Lämplighet och hållbarhet för komponenter behöver kontrolleras tidigt i designprocessen för att undvika kostnader som uppstår då felaktiga konstruktioner upptäcks vid testning.

Ett annat område där modeller ofta används är modelldriven utveckling av programvara. Till exempel modelleras datastrukturer och dataflöde med UML-diagram. Kod kan automatiskt genereras från modellerna och ytterligare förfinas genom användarskriven kod. Fördelar med modelldriven utveckling av programvara är kodens höga abstraktionsnivå inklusive visualisering i diagram samt specialiserade vyer som visar detaljer för olika designaspekter. Dessutom är kodgeneratorer oftast vältestade och använder designmönster i den genererade koden vilket leder till god kvalitet och läsbarhet.

Modeller för simulering av vindkraftverk utvecklas vid Fraunhoferinstitutet för vindenergi och energisystemteknologi (Fraunhofer IWES). Modeller med olika detaljnivåer används under designprocessen, från enkla modeller för grundläggande designbeslut och undersökningar till detaljerade modeller. Exempelvis används modellering med beräkningar av strömningsdynamik (CFD) för detaljerad flödesanalys av luftflöde över rotorbladen. De övergripande vindkraftverksmodellerna är parametriserade genom parametrar som inte är beroende av syntax i något specifikt språk eller verktyg. Därför måste dessa modeller översättas till ett verktygsspecifikt format före simulering. Konstruktionsfel i de övergripande modellerna har påverkat de detaljerade modellerna. Men lika viktigt är att problem som upptäcks och behöver åtgärdas i de detaljerade modellerna kan kräva ändringar i de övergripande modellerna. Detta leder till problemet att modeller som har skapats för en viss designstudie samt därtill kopplade transformationer till lämpliga modellformat för simuleringsverktyg kan behöva ändras. För att råda bot på
detta problem används modelldriven utveckling av programvara. Modeller skapas ursprungligen i den integrerade simuleringsmiljön. Dessa översätts sedan till specifika simuleringsmodeller i lämpliga format för de olika verktygen.

Denna avhandling fokuserar på verktyg för utveckling av komponenter av olika detaljnivåer med användning av modelleringsspråket Modelica. För att snabbt och effektivt utveckla modeller behöver utvecklaren få hjälp med att skapa modeller som är giltiga både vad gäller modelleringsspråket och de strukturella begränsningarna som ska visa det förväntade beteendet under simuleringsmiljön.


Interaktiv modellvalidering har utvecklats för att hjälpa användaren under utvecklingsfasen. Användaren får alltså omedelbar återkoppling och får reda på om hans eller hennes modeller är korrekta enligt modelleringsspråkets syntaktiska och semantiska regler. Vidare kan ytterligare semantisk information samt regler definieras på ett generiskt sätt för att begränsa hur modeller sätts samman. Detta kan hjälpa till att identifiera konstruktionsfel som annars kan vara svåra att upptäcka när man endast undersöker modellkoden.

OneModelica gör det möjligt att skapa korrekta Modelicamodeller som kan användas i ett godtyckligt simuleringsverktyg som följer Modelicas- standarden. Simuleringsresultat kan undersökas i den integrerade utvecklingsmiljön och ett automatiserat testramverk möjliggör testdriven utveckling och säkerställer att modellerna beter sig som förväntat. Implementationen använder sig mycket av modelldriven utveckling av programvara och visar således att det är möjligt att skapa kraftfulla utvecklingsmiljöer för komplexa domänspecifika språk såsom Modelica.

Detta arbete har finansierats av det tyska ministeriet för miljö, natur, byggnads- samt kärnkraftssäkerhet, av den svenska staten genom projektet ELLIT; av Vetenskapsrådet; samt av Vinnova genom projekten RTSIM och ITEA2 MODRIO.
I would like to express my greatest appreciation to Peter Fritzson for his valuable and constructive support during the development of this research work. Our fruitful discussions lead to new ideas on how the quest to simplify and aid model development for engineers could be enhanced. With his help, I managed to get through many review processes which allowed me to significantly enhance the quality of my publications.

I would also like to thank Adrian Pop and Martin Sjölund who reached out with a helping hand when I was struggling with the Modelica Language Specification. Our discussions were always friendly and smiles and laughter often accompanied the discovery of stumbling blocks.

My sincerest gratitude is extended to my colleagues Adam Zuga and Philipp Thomas who courageously took the plunge and actually used the tools that I developed for them. Being tough as old boots is surely one attribute describing them well as new releases almost drove them nuts. But their appreciation of my work and their many hints and pointers to glitches within my work helped me in continuing to press on.

Also much appreciated was the assistance provided by my colleague Mareike Strach-Sonsalla, especially when it came to understanding how engineers think. The collaboration with her led to great insights that aided in the continuation of my research work.

And last but not least I want to thank my family for supporting me and helping me carry on with my work.
# Contents

1 Introduction 1
   1.1 Iterative Component Model Development 2
   1.2 Model Development with Modelica 3
   1.3 Motivation 4
      1.3.1 Create Highly Detailed Models 4
      1.3.2 Derive Low Level of Detail Models Automatically 5
      1.3.3 Generate Tool Specific Models 5
      1.3.4 Simulate with Arbitrary Simulation Tools 5
      1.3.5 Analyze Simulation Results 6
      1.3.6 Adjust Highly Detailed Models 6
   1.4 Research Method 6
   1.5 Research Questions 7
   1.6 Contributions 8
   1.7 Publications 10
   1.8 Structure 11

2 Background 13
   2.1 The Modelica Language 13
      2.1.1 Object-Orientation 14
      2.1.2 Component-Based Development 15
      2.1.3 Modelica Models at Fraunhofer IWES 16
      2.1.4 Modelica Tools and Solvers 17
   2.2 Model-Driven Software Development 18
      2.2.1 Models of the Object Management Group 18
      2.2.2 Domain Specific Languages 20
      2.2.3 Using Textual DSLs 21
      2.2.4 Xtext Grammar for Modelica 22
      2.2.5 Modelica Meta Model 25

3 Static Validation of Modelica Models for Language Compliance 29
   3.1 Validating Modelica ASTs 31
   3.2 Validating Modelica Language Specification Compliance 32
      3.2.1 OCL Constraints 32
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2</td>
<td>Java Constraints</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Language Concept Comparison</td>
<td>37</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Performance</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Validation of Structural Constraints</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Possible Model Structures for a Horizontal-Axis Wind Turbine</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Introduction of Structural Constraints</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Modelica Type System</td>
<td>42</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Type Caching</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Type Validation</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>The International System of Units</td>
<td>44</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Limitations of Unit Validation</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Code Style Validation</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Related Work</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusion and Future Work</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Static Source Code Analysis of Physical Model Semantics using Role Models</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Concept of the Role Validation Framework</td>
<td>54</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Supporting Various Modeling Languages</td>
<td>54</td>
</tr>
<tr>
<td>4.1.2</td>
<td>The Validation Framework’s Languages</td>
<td>55</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Performing the Validation</td>
<td>56</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Granularity of Rule Definitions</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Role Models and Constraints</td>
<td>57</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Context Objects, Role Models, and Roles</td>
<td>57</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Rules</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Mapping the Role Models to Target Languages</td>
<td>60</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Relation of Role Models and Modeling Languages</td>
<td>61</td>
</tr>
<tr>
<td>4.3.2</td>
<td>The Role Model Mapping Language</td>
<td>61</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Querying Roles from Context Objects</td>
<td>63</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Query Language Concepts</td>
<td>63</td>
</tr>
<tr>
<td>4.4</td>
<td>The Validation Algorithm</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation</td>
<td>66</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Role Models for Structural Validity and Rated Power Design Aspects</td>
<td>67</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Mapping the Role Models to Modelica</td>
<td>69</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Mapping the Role Models to UML2</td>
<td>72</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Results</td>
<td>75</td>
</tr>
<tr>
<td>4.6</td>
<td>Related Work</td>
<td>75</td>
</tr>
<tr>
<td>4.7</td>
<td>Conclusion and Future Work</td>
<td>77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>MoUnit – A Framework for Automatic Modelica Model Testing</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Testing Modelica Models Within OneModelica</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation of the Test Framework</td>
<td>84</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Test Definition Language</td>
<td>84</td>
</tr>
</tbody>
</table>

---

x
## 8 Conclusions and Future Work

8.1 Contributions

8.1.1 Use of DSL Technologies for the Development of a Modelica IDE

8.1.2 Model Validation for Language Compliance

8.1.3 Domain Specific Model Semantics

8.1.4 Testing the Behaviour of Physical Models

8.1.5 Code Generation for Iterative Model Development

8.2 Future Work

### A Modelica Syntax

### B Role Mapping Syntax

### C Role Model Syntax

### Bibliography

### Index
List of Figures

2.1 Typical wind turbine design ........................................ 16
2.2 Hierarchy of meta-models defined by OMG ....................... 19
2.3 Creating Modelica-specific components and syntax trees with Xtext ..................................................... 22
2.4 Simplified Modelica meta model .................................... 26

3.1 Structural constraint for the connection of Tower and Nacelle objects ......................................................... 41
3.2 Simplified representation of an expression in the AST ............ 44
3.3 Unit validation in Modelica editor ..................................... 46
3.4 Warnings for code style violations .................................... 47

4.1 Semantic errors marked in the Modelica editor .................... 53
4.2 Technologies hierarchy of the role validation framework .......... 54
4.3 RMCL metamodel .......................................................... 58
4.4 Relation of Role Models and Models ................................... 61
4.5 RMML metamodel .......................................................... 62
4.6 Relation of the role validation components ......................... 66
4.7 Simplified Modelica metamodel ......................................... 70
4.8 Parts of the UML2 metamodel ........................................... 73
4.9 An erroneous ModelicaML wind turbine model with error markers ................................................................. 74
4.10 Example of a wind turbine model with rated power and wind speed constrained components ................................ 76

5.1 Model projects for both model logic and test models for regular water waves ...................................................... 82
5.2 Validation of MoUnit test definitions ................................... 86
5.3 Structure of the Modelica test framework .............................. 88
5.4 Result for test case TestRegularWave .................................. 91
5.5 Result for test case TestIrregularWave ................................. 91

6.1 Model transformation and code generation .......................... 94
6.2 Structure of laminate blade models ................................... 97
List of Figures

6.3 A simple beam model .................................. 97
6.4 An idealized blade cross section ....................... 98
6.5 Calculation of the beam model .......................... 99
6.6 EDD-model of a rotor, rotor blade, blade element and airfoil 100
6.7 The OneWind library and generated Modelica code .......... 101

7.1 Modelica IDE based on Eclipse ........................ 110
7.2 The Modelica editor ...................................... 111
7.3 Modelica syntax highlighting ............................ 112
7.4 Modelica syntax error in editor ........................ 113
7.5 Index data structure ..................................... 116
7.6 Example of code completion in the Modelica editor ....... 117
7.7 Example of code folding in the Modelica editor ........ 119
7.8 Package refactoring for Modelica models ............... 120
7.9 The Modelica perspective ............................... 121
7.10 Project explorer with Modelica specific extensions ....... 122
7.11 Physical model instance tree ......................... 123
7.12 The outline view ....................................... 125
7.13 HTML documentation of the DoublePendulum model .... 126
7.14 The Modelica project wizard ........................... 127
7.15 New Modelica project in project explorer ............... 128
7.16 New Modelica element wizard ........................... 129
7.17 Physical model wizard .................................. 130
7.18 Experiment wizard location page ....................... 132
7.19 Experiment wizard simulation tool selection page ....... 132
7.20 Experiment wizard simulation tool settings page ....... 132
7.21 Transforming structure models to modal format .......... 133
7.22 Configuring the model nature for existing projects .... 134
7.23 Experiments configuration dialog ....................... 135
7.24 Dialog for creating a new plot ......................... 136
7.25 XY time series plot .................................... 137
7.26 Simulation tool location settings ....................... 138
7.27 Installed libraries configuration dialog ............... 139
7.28 Open model dialog ..................................... 140
7.29 Data structure of the Modelica IDE ................... 141
7.30 Export library dialog .................................. 142
7.31 Documentation and icon generation ................... 145
7.32 Prototype of a Modelica connection editor in OneModelica 146
<table>
<thead>
<tr>
<th>Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Example of a replaceable element in Modelica</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Xtext header in grammar files</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Modelica import clause definition with Xtext</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Reference definition with Xtext</td>
<td>24</td>
</tr>
<tr>
<td>2.5 SimpleComponentName definition with Xtext</td>
<td>24</td>
</tr>
<tr>
<td>2.6 Comment definition with Xtext</td>
<td>25</td>
</tr>
<tr>
<td>2.7 Terminal ID definition with Xtext</td>
<td>25</td>
</tr>
<tr>
<td>3.1 OCL constraint restricting the use of operators</td>
<td>33</td>
</tr>
<tr>
<td>3.2 OCL helper method for checking whether a class is an operator record</td>
<td>33</td>
</tr>
<tr>
<td>3.3 OCL constraint restricting functions to have at most one algorithm section</td>
<td>34</td>
</tr>
<tr>
<td>3.4 OCL constraint checking whether a stream connector has exactly one scalar variable with the flow prefix</td>
<td>35</td>
</tr>
<tr>
<td>3.5 OCL helper methods checking whether a component is a flow or stream variable</td>
<td>35</td>
</tr>
<tr>
<td>3.6 Java constraint restricting the use of operators</td>
<td>36</td>
</tr>
<tr>
<td>3.7 Optimized Java constraint checking the stream connector restriction</td>
<td>37</td>
</tr>
<tr>
<td>3.8 Simple Modelica expression</td>
<td>43</td>
</tr>
<tr>
<td>4.1 Defining a context object and ports</td>
<td>57</td>
</tr>
<tr>
<td>4.2 Defining role models, roles and rules</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Defining the target model and the mapped role model</td>
<td>62</td>
</tr>
<tr>
<td>4.4 Mapping connections and ports to the target language</td>
<td>63</td>
</tr>
<tr>
<td>4.5 Query concepts of the mapping language (RMML)</td>
<td>64</td>
</tr>
<tr>
<td>4.6 Roles for structural validity</td>
<td>68</td>
</tr>
<tr>
<td>4.7 Roles for rated power design</td>
<td>69</td>
</tr>
<tr>
<td>4.8 Mapping for Modelica</td>
<td>71</td>
</tr>
<tr>
<td>4.9 Mapping for UML2</td>
<td>72</td>
</tr>
<tr>
<td>5.1 Example for a model and a corresponding test model</td>
<td>83</td>
</tr>
<tr>
<td>5.2 Example of a test case definition</td>
<td>85</td>
</tr>
<tr>
<td>5.3 Example of a test suite definition</td>
<td>85</td>
</tr>
<tr>
<td>5.4 Test definition for regular waves</td>
<td>90</td>
</tr>
<tr>
<td>5.5 Test definition for irregular waves</td>
<td>92</td>
</tr>
<tr>
<td>Listings</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>6.1 Generated blade data record</td>
<td>102</td>
</tr>
<tr>
<td>6.2 Load element containing the airfoil descriptions</td>
<td>103</td>
</tr>
<tr>
<td>6.3 Generic type profile</td>
<td>103</td>
</tr>
<tr>
<td>6.4 Concrete profile record Profile_NACA64</td>
<td>104</td>
</tr>
<tr>
<td>6.5 Generated blade model</td>
<td>105</td>
</tr>
<tr>
<td>6.6 Redeclaration of blade element data</td>
<td>106</td>
</tr>
<tr>
<td>6.7 Main wind turbine class with redeclared components</td>
<td>106</td>
</tr>
<tr>
<td>7.1 Simple Modelica example</td>
<td>114</td>
</tr>
<tr>
<td>7.2 Modelica documentation annotation</td>
<td>126</td>
</tr>
</tbody>
</table>
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree. 9, 21–23, 25, 31–33, 40, 43, 44, 48, 55, 99, 112, 117</td>
</tr>
<tr>
<td>CST</td>
<td>Concrete Syntax Tree. 21, 23, 112</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language. 2, 3, 6–8, 13, 20, 21, 47, 55, 77, 110, 111, 144, 147</td>
</tr>
<tr>
<td>EDD</td>
<td>Engineer Design Data. 19–21, 94, 95, 99, 107, 134</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework. 3, 4, 6, 8, 9, 18–21, 23, 31, 44, 52, 54–56, 63, 64, 75, 95, 96, 99, 109, 115, 116, 130, 141, 148, 149</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite Element Method. 1, 16, 94, 96, 101</td>
</tr>
<tr>
<td>Fraunhofer IWES</td>
<td>Fraunhofer Institute for Wind Energy and Energy System Technology. 1, 2, 5, 9, 11–13, 16, 19, 20, 30, 89, 93, 97, 150</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment. 3, 4, 8, 17, 18, 29, 39, 144, 147</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility. 18–20</td>
</tr>
<tr>
<td>MSL</td>
<td>Modelica Standard Library. 16, 24, 27, 38, 40, 42, 44, 45, 78, 113, 117, 123, 124, 126, 128, 136, 138, 139, 149, 150</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language. 29–35, 37–40, 47, 48, 148</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group. 18, 32</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>RMCL</td>
<td>Role Model Constraint Language. 54, 55, 57, 60, 63, 64, 69, 71, 77, 78</td>
</tr>
<tr>
<td>RMML</td>
<td>Role Model Mapping Language. 54, 55, 61, 63, 64, 69, 71, 77</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language. 20, 21, 32</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier. 23, 116</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Fraunhofer Institute for Wind Energy and Energy System Technology (Fraunhofer IWES) has developed models for the design of new wind turbine components and the simulation of full-scale wind turbines. In order to investigate the performance in large scale (e.g., when simulating load cases for the entire wind turbine) as well as in small scale scenarios (e.g., when optimizing the laminate layers of a rotor blade model to increase the strength of the blades) various models with different levels of detail are used during iterative design steps. When the design of a lesser level of detail is completed, a model with higher detail is usually developed and refined. Low level of detail models can, on the other hand, usually automatically be derived from models with higher level.

Models with high detail are needed for various tasks during turbine development. Buckling analysis is, for example, needed to verify that rotor blade designs withstand occurring forces during operation. The analysis is performed using Finite Element Method (FEM) models. FEM models are too detailed for direct use in load calculation. However, modal reduced blade models can automatically be derived from the FEM models. The level of detail is reduced but the behaviour under load is retained by calculation of the eigenforms. This makes detailed analysis like for buckling impossible but accelerates load calculation with flexible blades.

Abstraction of models is essential when various tools for simulation shall be used. In wind turbine design it is important to compare simulation results from different tools to ensure that the results are correct. Modal reduced blade models therefore can be calculated and stored in an abstract way. When it comes to simulation with a selected tool, a concrete input format can be created from the abstract data. This ensures flexibility during the development process and reduces dependencies from single tool vendors.

Simulation results of the detailed models are not only used for improving the highly detailed model, but may also have an influence on previously designed models. Since different simulation tools are utilized in this process,
the design cycles can be cumbersome and refinements may be omitted by the designer due to time limitations.

1.1 Iterative Component Model Development

To overcome the issue of using various model types during the design process of new wind turbine components, Fraunhofer IWES uses Domain-Specific Languages (DSLs) for wind turbine designs in the OneWind project [Strobel et al., 2012]. In the past a wide range of programming languages has been developed in order to provide a suitable way of defining and solving problems by computation. The languages can be categorized into general purpose languages like Java and C++ on the one hand and DSLs on the other. While it is possible it is also difficult to define processes for arbitrary problems with general purpose languages. DSLs exploit their strengths in defining problems and solutions for a specific domain.

The possibility of describing domain specific problems by experts in custom domain specific languages makes the use of DSLs increasingly popular. Tools assist developers at defining the language grammar and then automatically generate editors and views for documents that the domain experts edit. As a result it has become easier to define specialized languages for custom domains, for example for software composition with the Universal Composition Language [Johannes, 2010] or Aircraft design with CPACS [Nagel et al., 2012].

The concept of defining DSLs is used in the aforementioned OneWind project — languages are defined for special kinds of models by domain experts. The OneWind user can then create models by providing parameters for wind turbine components using specialized languages. In many cases, simple models can automatically be derived from highly detailed models. The parameterized and potentially transformed models are then used for code generation for arbitrary simulation tools. Simulation results are transferred back to the OneWind environment and can be used to assess the models for further refinement. This results in following workflow:

I. Create or edit highly detailed models,
II. Derive low level of detail models automatically,
III. Generate tool specific models,
IV. Simulate with arbitrary simulation tools,
V. Analyze simulation results,
VI. Adjust highly detailed models, and
VII. Models are appropriate or continue workflow with step I.
The models of wind turbine components must be defined with the same data structures to enable this workflow. For the OneWind project, the Eclipse Modeling Framework (EMF) is used since it is a mature and open source framework for development of DSLs. Transformations between models can be defined in a declarative way with tools like Query View Transformation (QVT) [OMG, 2011] or Atlas Transformation Language (ATL) [Jouault et al., 2008]. Model to text transformations, for example the generation of simulation models, can be done with tools like Xtend [Bettini, 2013].

1.2 Model Development with Modelica

Elmqvist designed a language for modeling and simulation of physical models in his PhD thesis [Elmqvist, 1978]. The idea was to provide engineers with a powerful language that supports object-oriented modeling as well as component-based model design, which is common in engineering. In the fall of 1996 a working group was started to design a new language based on several current equation-based object-oriented modeling languages including Elmqvist’s language. In September 1997 the Modelica Design Group published on the web a document describing the first version of this language, Modelica 1.0. Since then, this language has been further developed and enhanced during a number of design meetings.

One benefit that this language offers is the support of equation-based systems as they occur regularly in the computation of models of physical systems. Therefore models of physical components can be connected and the interaction between these components can be modeled using equations. This is different from implementation in general purpose languages where equation systems are not supported and equations need to be transformed into algorithmic code to represent causality explicitly. Modelica therefore provides the engineer with a method to model components in a natural, equation-based way. Since equations cannot be computed directly with standard computer architectures, it is the task of modeling and simulation environments to either interpret the models to produce simulation results or to transform the equations into an efficiently computable form, for example by compilation into C-code which can be compiled and executed. In this way, the implementation details are hidden from the engineer who can instead focus on the design of models of physical systems.

One advantage of general purpose languages is user support by feature-rich Integrated Development Environments (IDEs) like the Eclipse Java Development Tools (JDT) for Java or Microsoft Visual Studio for the C language family. User support of the aforementioned tools is comprehensive because there is a high number of potential users of the respective language. On the contrary, DSLs will hardly have a comparable number of users as they target a specific domain. However, the number of Modelica users has grown considerably in the past which has led to an increased interest in a
sophisticated Modelica IDE. 

One goal of this dissertation is to show that it is possible to develop efficient IDEs by using modern model driven software development techniques. Modelica, therefore, is an eligible candidate as the grammar is quite complex. The language also comes with special requirements that cannot be generalized for arbitrary languages. This includes a specific package and class structure as well as replaceable classes for easy replacement of model components. Besides creating common functionality and development support, focus is set on the correctness of Modelica models.

1.3 Motivation

This thesis partially touches on all aspects I-VI of the iterative model development steps described in Section 1.1. The mathematical modeling language Modelica has been chosen as the first target language to show that the workflow can be performed. Due to the decision to use the EMF framework as the data structure for all models employed during the different design steps, a grammar for Modelica has been defined with Xtext. The requirements for the different steps of the described workflow are detailed below.

1.3.1 Create Highly Detailed Models

For the design of highly detailed models it is vital to have comprehensive tool support to ensure that the models are correct regarding the modeling language’s syntax and semantics. These aspects need to be checked efficiently during model development to provide immediate feedback to the user and to avoid deceleration of the modelling process due to errors in the models.

Complex models are usually built from various components. A wind turbine for example consists of a tower, a rotor, and rotor blades. The assembly of these components should be checked for compatibility to prevent the user from accidentally connecting to mismatched components. For example it is illogical to connect a tower to a rotor blade. Specialized tools for the design of wind turbines can detect such errors immediately or even prevent the user from creating wrong connections. More general languages like Modelica, however, do not provide an automatic mechanism to prevent such faulty model configurations.

Models of physical systems are designed with an expected behavior. This behaviour needs to be verified by simulation in different environmental conditions. Hence, a modelling environment should provide a way to automatically check simulation results against expected results, especially after model components have been changed. This is mostly done by regression tests which compare simulation results to previously created results. However, analytic results may also be used for small components that can prove the correct behaviour of the components.
1.3.2 Derive Low Level of Detail Models Automatically

In wind turbine design low level of detail models can often be automatically derived from high level of detail models. As an example, finite element models (FEM) of rotor blades that describe the structure in detail can be used for the generation of modally reduced flexible blades [Shabana, 2010]. The structure together with material properties are taken into account for the calculation of eigenfrequencies and eigenforms of the blade. From the resulting data the flexible behaviour can be derived with reduced calculation cost compared to the detailed source model.

The model development environment should hence provide the possibility to derive the fast low detailed models automatically from detailed model instances. The modal reduction implemented at Fraunhofer IWES has been integrated into OneModelica. Input parameters can be defined as a Modelica record and a modal reduced representation is generated that can be used for simulation purposes.

1.3.3 Generate Tool Specific Models

Abstract models as defined in the OneWind project cannot be simulated directly since the design of the data structure is aimed at being generic. Parameterized instances of the generic model must be transformed into a tool-specific format in order to be simulatable. For Modelica it turned out to be beneficial to implement a fully functional library for wind turbines. Templates are provided that can be parameterized for simulation of individual turbine designs together with general functionality from the library.

The parameter files must be derived from the generic OneWind models. Therefore access to the templates and the Modelica grammar must be provided in order to generate valid Modelica code that can be used for the simulation of wind turbines.

1.3.4 Simulate with Arbitrary Simulation Tools

One goal of the design workflow described above is to use the most suitable simulation tool for any design step. Therefore, generic models need to be transferred to the language of the simulation tool. As an example, tower models that are based on stations for structural data and tower elements for aerodynamics can be transferred to formats that can be simulated by the domain specific tools FAST\(^1\) and Bladed\(^2\) for wind turbine simulation, or to Modelica to be simulated with the OneWind Modelica library. These tools can be used since the tower declaration is done in a similar way.

Instead of a dedicated language it would also be possible to implement a simple GUI for model parameterization. However, using models has the benefit that model to model transformations can be executed, for example

\(^1\)https://nwtc.nrel.gov/FAST

with QVT or ATL. The target model can, for example, be a FAST language defined with Xtext. By unparsing the transformed model, a FAST compatible input file could be created by the generated serializer.

Another possibility is the use of the template language Xtend for generating (text based) input files from the parameterized models. This avoids the need to define a possibly complex target language grammar.

1.3.5 Analyze Simulation Results

Simulation results are usually stored in result files and different formats are used, for example comma separated values (CSV) or the MATLAB™MAT-File format. Parsers need to be provided for the analysis that make the results accessible. Furthermore, results need to be visualized by plots. Assessments can help to further analyze the results, for example, by fast Fourier transformation (FFT) or the calculation of minimum and maximum values.

1.3.6 Adjust Highly Detailed Models

Adjusting highly detailed models is directly related to Section 1.3.1. The same environment that is used for the creation of detailed models can also be used for their refinement.

1.4 Research Method

The approach used in this dissertation addresses different types of questions that occur during research. Two question types are answered in this dissertation: Conceptual questions consider problems that are defined “by incomplete knowledge or flawed understanding” [Booth et al., 2009].

The second type of questions considered throughout this dissertation is about practical questions. Solving this kind of questions leads to solutions that can directly help others in achieving their goals.

An important question investigated in this dissertation is whether DSL tools have become mature enough for use with complex languages. This question is answered by the development of a DSL for the mathematical modelling language Modelica. Other examples of questions investigated in this dissertation are whether domain specific constraints can be defined and checked efficiently on large models, the behaviour of models of physical systems can be tested automatically and the proposed model transformation can be performed with models defined with Modelica. The implemented frameworks allow users to define custom validation rules and test cases. The implementation of Modelica as a DSL with EMF tools can be seen as a prototype for other languages which should be integrated into the iterative model development workflow from Section 1.1.
1.5 Research Questions

This dissertation addresses several questions that arise when trying to establish the workflow from Section 1.1. They are listed in the following:

I Can the technology of DSLs be used for complex languages like Modelica to create a development environment that aids the model developer appropriately during modeling?

Following the approach of [Booth et al. , 2009], this will lead to an answer to the conceptual questions if the DSL tools used are mature enough to be utilized with languages and applications of high complexity. Although many examples of DSLs exist, they are often designed for small domains which results in simple languages. Tools for complex languages, however, mostly provide few features like parsing, syntax validation and printing [Heidenreich et al. , 2009].

II How can language-specific validation rules be formulated and efficiently checked on models such that they comply to the rules defined on the language’s specification?

Modelica is a general purpose language for mathematical modeling. Equations are used to describe physical behavior of real world objects. Thus, it has language specific validation rules which are different to those of general purpose programming languages. However, the rules should be checked interactively during model development to ensure that the user is creating models which are correct with respect to the language specification.

III Is it possible to create and check rule definitions for DSLs by adding user defined semantics to models?

Another drawback of a general purpose mathematical language is that user defined semantics of the models defined with this language cannot be checked automatically. The user should be able to define rules to prevent erroneous model creation as in the example of a tower of a wind turbine connecting to a rotor blade. A semantic validation framework is needed since general languages like Modelica are not sufficiently restrictive for certain domains like wind turbines.

IV Can we create an automatic test environment for mathematical modelling with Modelica that allows us to check if the model represents the behavior of the modelled object correctly?

Besides, rules that can be statically checked, further tests must be performed on models to validate that the expected behaviour is achieved. For composed models this can become a complex task since simulation results must be taken into account and compared to expected values. Besides it must be possible to compare simulation results of modified models to the results of previous versions in order to recognize design flaws that may have been implemented in the new version.
V How can the parameterized wind turbine models be transformed into models with less detail and into executable simulation code?

This last question is important for the iterative model development. Following [Booth et al., 2009] this leads to a solution of an important practical problem. The developed Modelica IDE will provide the necessary basic functionality for the generation of parameterized wind turbine models with the Modelica modeling language. Therefore, Modelica models are represented with EMF in order to be utilized with the tools mentioned above.

1.6 Contributions

This dissertation introduces OneModelica — an IDE that supports Modelica users in creating correct and simulatable models. OneModelica makes it possible to integrate Modelica models into the design workflow for wind turbines as described in Section 1.1. The research contributions directly target the questions raised in Section 1.5:

I. Xtext is used as a DSL framework for the creation of a Modelica grammar. Based on the grammar definition an editor an views can be generated automatically. Besides, project management including support for libraries, the definition of model instances for simulation and the simulation environment that supports several simulation tools has been implemented. The necessary data structures have therefore been modeled. The settings can be persisted and shared with other model developers. OneModelica is an example of an IDE that is heavily relying on model driven software technologies. The Modelica IDE additionally provides support for the development of high detailed models with Modelica and the generation of tool specific models as needed for the iterative model development. Different simulation tools can be used and the simulation results can be analyzed directly inside OneModelica which is important for iterative model development. This approach answers the research question I and shows that DSL tools can be used with complex languages.

II. Language specific validation for Modelica has been implemented in OneModelica. Rules have been extracted from the textual language specification and checking has been implemented with two technologies: OCL OMG [2012], which is the standard constraint language for EMF models and plain Java code. The performance and readability of the constraints have been compared to investigate whether constraint checking is fast enough to be performed interactively during development. The validation supports the model developer in creating and refining high detailed models as defined in the iterative model
development paradigm. Hence, it has been demonstrated how efficient
language-specific model validation can be implemented.

III. A problem that can occur during model development, particularly if
many variants of a model component exist, is erroneous connections
between the model components. For physical systems semantic re-
strictions exist defining which components can be used with each other.
General languages like Modelica do not restrict connections sufficiently
in the language grammar. With the role based validation framework
integrated into OneModelica, domain specific restrictions can be de-
fined and models can be annotated accordingly. Thereby it can be
checked interactively, if the connected components can be used with
each other. The framework is not bound to Modelica and can be used
with other languages defined with EMF. Development of high detailed
models can be assisted by semantic validation since the developer im-
mediately recognizes erroneous model configurations. This approach
shows how models can be validated semantically and thereby answers
research question III.

IV. The behaviour of models can not be checked statically. Investigation on
expected behaviour of physical models can be automated with a Mod-
elica unit test framework that has been implemented for OneModelica.
Simulation execution and validation of simulation results, e.g. by using
regression tests can be automated. This ensures that model changes
do not have a negative effect on simulation results. This is especially
useful for complex models where changes in one model component can
affect the behaviour of manifold composed models.

V. This dissertation shows how the parameterized models from Fraun-
hofer IWES can be transferred into simulatable Modelica code. Prob-
lems with ragged arrays that occurred during the model generation
have been detailed. Besides, a model transformation from high to low
detailed Modelica models by modal reduction has been integrated into
Modelica. The required parameters are extracted from a Modelica
template and code is generated that can, for example, be used with
the OneWind Modelica library for wind turbine simulation. The code
performing the modal reduction has been re-used from the OneWind
project and is not part of this dissertation.

The Modelica editor generated from the Modelica grammar defined with
Xtext is the main feature of the developed IDE. The developer is sup-
ported by views during model development which present filtered informa-
tion for certain aspects of the edited document like the structure of the
document, or other information like documentation or defined projects in-
side the workspace. Most views implemented for Modelica retrieve their
information from the Abstract Syntax Tree (AST) of the parsed document.
The documentation view, for example, gets the annotations from the selected model and displays the HTML-based documentation that has been defined inside the annotations.

1.7 Publications

The content of this dissertation was partly presented in the following publications:

   - Main idea: Roland Samlaus
   - Text and editing: Roland Samlaus
   - Discussion, validation and proofreading: Birgit Demuth
   - Claudio Hillmann
     - Partial implementation of experiment and project management.
   - Martin Krebs
     - Partial implementation of the OCL constraints.

   - Main idea: Roland Samlaus
   - EDD concept: Michael Strobel, Claudio Hillmann
   - Text and editing: Roland Samlaus, Adam Zuga
   - Discussion, validation and proofreading: Peter Fritzson

   - Main idea: Roland Samlaus
   - Text and editing: Roland Samlaus
   - Discussion, validation and proofreading: Mareike Strach

- Main idea: Roland Samlaus
- Text and editing: Roland Samlaus, Mareike Strach
- Discussion, validation and proofreading: Claudio Hillmann, Peter Fritzson
- Mareike Strach
  - Text about the need for tests in model development.


- Main idea: Roland Samlaus
- Text and editing: Roland Samlaus
- Discussion, validation and proofreading: Peter Fritzson


- Main idea: Philipp Thomas, Xin Gu
- Text and editing: Philipp Thomas, Roland Samlaus
- Discussion, validation and proofreading: Roland Samlaus, Claudio Hillmann, Urs Wihlfahrt
- Roland Samlaus
  - Implementation of information retrieval for modal reduction and serialization as Modelica models.

### 1.8 Structure

The Modelica language is introduced in Chapter 2. The wind turbine models that are modeled using Modelica at Fraunhofer IWES are briefly described and the procedure of compilation and simulation of Modelica models is explained. Furthermore, the Chapter 2 introduces model-driven software development, which is the basis for the development of OneModelica.

The topic of Chapter 3 is the validation of models regarding language compliance. Different kinds of constraint declarations are discussed and the
readability and performance of the constraints defined with the constraint language OCL is compared to an implementation with the Java language. A first attempt of semantic validation for the restriction of how model components can be used with each other is proposed. Furthermore, type and unit validation is discussed and their implementation in OneModelica described.

Chapter 4 introduces a role-based approach for semantic validation. Model components can be annotated with roles and rules that restrict how the roles can collaborate. The validation framework is implemented in such a way that the roles and constraints can be used with arbitrary languages and not only for the Modelica language.

A framework for automatic Modelica model testing is presented in Chapter 5. Unit tests can be created and assembled in suites for automatic testing. Custom tests can be added by developers via an extension mechanism. Likewise, it is possible to add custom test components for the visualization of test results.

Chapter 6 shows how wind turbine models are defined in an abstract way at Fraunhofer IWES. The abstract models can then be transformed into executable simulation code. As an example, the transformation to Modelica code is described. Problems due to polymorphism which appear during the transformation are discussed.

This dissertation includes a description of the implementation of OneModelica. The components of OneModelica, including the editor, views, wizards and dialogs are described in Chapter 7. It is explained how projects can be used to separate models for different design aspects from each other and how libraries are created and used to distribute models to other developers.

Finally, Chapter 8 concludes the dissertation and gives an outlook for future work.
Chapter 2

Background

This chapter summarizes the basic technologies used for OneModelica. The main features of the mathematical modeling language Modelica and the use of Modelica for the development of wind turbine models at Fraunhofer IWES are described. Furthermore, using model-driven software development as the technological basis for the software development part is explained, including the design of DSLs for textual languages.

2.1 The Modelica Language

Modelica has its roots in a language called dynamic modeling laboratory (DyMoLa) that was initially presented in the PhD thesis of Hilding Elmqvist [Elmqvist, 1978], together with several equation-based object-oriented languages from groups which participated early in the Modelica design effort (some examples are Neutral Model Format (NMF) [Sahlin et al., 1992], ObjectMath [Viklund et al., 1992] and Smile [Kloas et al., 1995]). Later on, the idea of allowing modeling and simulation of equation-based physical systems was further developed and eventually led to the language Modelica that is defined and managed by the Modelica Association. Mathematical models created with Modelica can be of various kinds. They can, for example, represent components of mechanical, electrical or hydraulic systems. One major advantage is that multi-domain models can be connected to each other for simulation [Fishwick, 2007, pp. 36.1 - 36.27]. This makes it possible to fully assess the behavior of complex models, such as wind turbines, including the influence of wind, water and electrical grids.

The Modelica language is an open standard and thus independent from tool vendors. Several tools exist that are able to translate Modelica model files to a computable form (e.g. C or Java). In the last years, modifications were applied to the language in order to increase the expressiveness, reusability, and portability between tools. The language version discussed in this chapter is 3.3 [Modelica Association, 2012].
within OneWind.Components.Rotor.Blade;
model ModalBlade

public
replaceable record BladeParameter = ModalBladeData
  constrainedby BladeDataPartial
"Structural blade parameters and profile data."
BladeParameter bladeParameter "Instance of blade parameters.";
end ModalBlade;

Listing 2.1: Example of a replaceable element in Modelica

2.1.1 Object-Orientatio

The Modelica language is object oriented. The concepts of classes and packages including inheritance are supported to enhance the re-usability of the code. This provides high flexibility and re-usability since variants of models can easily be created by extending existing models and adapting the models to custom requirements. Models with the keyword partial are incomplete class templates that cannot be instantiated directly, but must be sub-classed. Classes in Modelica can use multiple inheritance in order to reuse declarations and equations from multiple classes. Besides object orientation, the component-based design paradigm is supported by using connectors (mostly physical connectors, flow and potential) and connections between model components. This again facilitates re-usability since ready-to-use components can be distributed between developers and complex system models can be composed by connecting the components to each other. Hence, two types of mechanisms are provided by Modelica to create physical behavior based on existing components: by class inheritance and by the composition of components (see Section 2.1.2).

Another language element available in Modelica that improves the reuse of code is replaceable elements. Components inside classes can be defined as replaceable, which means that the types and/or definitions of the components can be changed when an instance of the class is created. This mechanism can be compared to the concept of generics in languages like C++ or Java. Replaceable components can be constrained in the way that replacing classes must be of a certain sub-class. This enforces that the replaced class contains class elements that are necessary for simulation. Listing 2.1 displays an example of a replaceable class. The record BladeParameter is defined as ModalBladeData inside the class ModalBlade.

An instance of the blade parameter is created and can be used, for exam-
Chapter 2. Background

2.1. The Modelica Language

ple in equations of the model. The record can be replaced but the replacing record must be a subtype of `BladeDataPartial`.

The encapsulation of the states and behavior of models into classes reduces the complexity during development. Instances of classes, in the Modelica Language Specification (MLSpec) [Modelica Association, 2012] called components (but with a different meaning from components of model libraries, which are classes), are used inside models, for example for calculations expressed by algorithms or equations. Modelica provides several kinds of specialized classes (class, model, package, record, block, connector, type, function and operator) which, from the Modelica grammar syntactic point of view, differ only in the keyword that declares the kind of the class. Additional restrictions, enhancements, and meanings of such specialized classes are defined in the MLSpec. Functions, for example, cannot declare any equations. This is really a syntactic restriction, which, however, is not enforced by the Modelica grammar to keep a slender Modelica grammar complexity. Violations of this constraint would lead to compilation errors when trying to compile the model for simulation.

2.1.2 Component-Based Development

In software development the idea of component-based development is well-known [McIlroy, 1968] and composition systems exist [Assmann, 2003; Johannes, 2010] which allow the composition of programs from code fragments. For the modeling of physical systems, the separation into model components and composition of larger systems based on model instances is even more intuitive. Real world systems are usually assembled from components, likely from third-party vendors, and connected to each other. An example in the mechanical system domain are wind turbines which combine components such as rotor blades and hubs. The MultiBody library from the Modelica standard library provides the concept of frames for connector classes to connect such components if they are modeled as multi body systems.

Composition is done by usage of the same kinds of ports and connections. The language element `connector` (an instance of a connector class) represents a port in Modelica and connections between `connectors` are established by `connect equations`. This mechanism allows the components to interact with each other without the need to know about any implementation details. The connections are established by coupling two variables to each other by an equation. Three different types of coupling are provided (potential, flow, and stream) that, for potential and flow, are based on Kirchhoff’s first and second law [Fritzson, 2014, Sec. 5.3.1p]. The black-box composition allows the ability to create multi-domain models since it doesn’t matter how the models are defined as long as the connectors match.
2.1.3 Modelica Models at Fraunhofer IWES

Modelica is used at Fraunhofer IWES for the development of a library for aero-servo-hydro-elastic wind turbines. One benefit of using Modelica is the availability of predefined model components for mechanical as well as electrical systems in the Modelica Standard Library (MSL). This allows for the fast development of models that can be used for the simulation of behavior in different environmental conditions, such as varying wind. Additionally, offshore turbines are modeled to simulate the behavior of wind turbines with different kinds of substructures under the influence of wind and wave loads.

A typical wind turbine (Figure 2.1) consists of a tower that is connected to a nacelle. The nacelle contains a drive train that is connected to the generator that generates electrical power from the torque produced by the rotor. The drive train is connected to a rotor mounting one or more rotor blades (the typical design has three rotor blades). At the bottom, the tower is connected to the ground or in case of offshore wind turbines to a substructure that is influenced by water of varying depth. All these components need to be developed in order to perform load case simulations that are needed for the certification of wind turbines. Additionally, different types of models are needed, such as FEM models for the calculation of buckling of rotor blades or modal reduced blade models for fast simulation of load cases with flexible blades. Many of the models are available in the OneWind library [Strobel et al., 2011; Thomas et al., 2014] and developed at Fraunhofer IWES.

Models from the OneWind library are used in the workflow described in Section 1.1. The models are defined in the way that the parameters are defined in data records. Hence, the parameters can be changed by defining
new records and instantiating the models using these records. The Modelica records are generated from parameterized, tool-independent wind turbine models from the OneWind framework.

### 2.1.4 Modelica Tools and Solvers

This section introduces simulation tools allowing to create, compose and simulate Modelica models. Since Modelica is a language specification, models could potentially be created with simple text editors. However, model development can become complex, especially when creating or working with large scale libraries. Specialized editors and IDEs aid in developing Modelica models. The editors are able to highlight keywords of models which make it easier for the user to understand the structure of the code. Convenient IDEs support the user by providing automatic syntax and semantic validation of the source code. Additionally, views on the edited documents representing the class structures or displaying documentation of reused components simplify the development process.

The developed physical models are used in simulations in order to analyse and assess the behavior. Usually, simulation results are plotted and examined by engineers or automatically processed, like for the calculation of a mean value during simulation. Since Modelica is an equation-based language that allows the definition of time-continuous systems, the models cannot be translated directly to executable byte code as done with general purpose languages like Java. General purpose languages are usually not compiled directly to byte code, either, since code optimization is performed on different intermediate language levels. However, general purpose language’s structure in principle allow direct compilation. The intermediate languages usually only reduce the complexity of the language syntax and semantics and represent the program in a way that improves execution performance.

Several tools are available that are capable of compiling equation-based Modelica models into algorithmic code, mostly C. In the compilation process, intermediate steps need to be performed. At first, the Modelica models need to be flattened. This means that all class inheritances are resolved resulting in a single model that contains all variables, equations and algorithms, including those of the inherited classes. Moreover, modifiers are expanded, connection equations are expanded to constituent equations, etc.. Based on the flattened models optimizations can be performed, for example by removing common sub-expressions to prevent unnecessary redundant calculations.

When typical mechanical model components are used, index reduction must be performed. Therefore index reduction is part of the standard compilation approach. After index reduction to index 0 form (an explicit ODE), the equations can be calculated by ODE solvers. Different solvers are available that are suitable depending on the type of models that are simulated. The result of this process is usually executable code that can be distributed.
to others. If only parameters change, models usually need not be re-compiled (except when so-called structural parameters are changed) and can directly be reused which accelerates the simulation and enhances re-usability. The generated code can also be used when exporting pre-compiled models in the Functional Mock-up Interface (FMI) format. The interface of the Modelica model is provided as an XML-file, the FMI model description [Blochwitz et al., 2011, 2012]. The model description together with the compiled code represents a Functional Mock-up Unit (FMU) that can be coupled to other FMUs or simulation tools.

The implementation of the flattening, compiling and simulating Modelica models as described above is out of the scope of this dissertation. However, it is possible to use simulation tools (e.g. OpenModelica) with OneModelica. The models can directly be simulated with the tools and the simulation results are returned back to OneModelica. This allows the developer to use different simulation tools and to analyse the behavior of the models directly in a single IDE.

2.2 Model-Driven Software Development

When developing software, problems arise when the customer that orders a new software product and developers that are responsible for the development of the software are experts in distinct domains. When discussing desired features of the software product, both parties have different opinions what certain terms mean and how they should be implemented. Hence, software developers often misunderstand what the ultimate goal of the customer for the software being developed is.

This discrepancy is targeted by model-driven software development since it introduces models as the main concept for defining data structures and processes of software. Different technologies are available for the definition of such models. This chapter describes the definition of models and modeling languages as defined by the Object Management Group (OMG), which is an international consortium for definition of certain industry standards in this area. Furthermore, the definition of a modeling language (meta-model) and a textual representation of the models is explained.

2.2.1 Models of the Object Management Group

The workflow in Section 1.1 describes how physical models are created in a generic way in the OneWind framework. EMF is used for the definition of the generic models. EMF implements the Meta-Object Facility (MOF), a meta-modeling approach that is proposed by the OMG. The MOF [OMG, 2006] defines four modeling meta levels (Figure 2.2) on how real world objects can be modeled. The meta levels correspond to the concepts of objects that can be abstracted by programs, languages and grammars for the definition of languages.
The following list gives an overview about the meta levels in the domain of wind turbine models:

- **M0 (instance)** is the first level and contains model instances with a state that represent real world objects and their collaboration, like rotor blades, towers or entire wind turbines.

- **M1 (model)** contains the models that are created to describe the structure and behavior of physical models (e.g., Modelica models of rotor blades, towers, wind turbines, etc., or Java classes as templates for M0 objects). The actual program or model text is at this level.

- **M2 (metamodel)** describes the structure of the modeling language used to create models. For physical systems, this can for example be the general purpose language Java or a dedicated physical modeling language like Modelica. A grammar defining the syntax of a language is a meta model for the language.

- **M3 (meta metamodel)** defines how the metamodels of M2 can be described. This is done by the concepts class, reference and attribute. The grammar of the syntax of a grammar definition language, such as BNF notation, is an example of a meta metamodel.

Since Eclipse is used as the framework for the products created at Fraunhofer IWES, the MOF implementation EMF [Steinberg et al., 2009] is used for the definition of modeling languages (e.g. Modelica and Engineer Design Data (EDD)).
EMF implements a basic version of MOF, called EMOF. Hence, the parametric data layer used in OneWind is built by various meta-model definitions created with EMF. EDD models specified in the meta-models implement a common interface. One strength of the meta-model approach is that one can define automatic transformations between two models of the same level, if one can define relations between features on the meta level. EMF is widely used in the Eclipse community and hence many tools exist that simplify the definition and use of the models. EMF allows generic processing of model instances making it possible to create functionality for a wide spectrum of diverse models. As an example, a generic editor has been implemented at Fraunhofer IWES enabling the user to edit arbitrary models that are based on an EMF meta-model. Generic SWT-composites for basic data types like double, int and String are available. Special composites can be easily registered as an OSGi [McAffer et al., 2010] service in order to provide a convenient way to edit custom data types.

### 2.2.2 Domain Specific Languages

The abstract syntax of a DSL is described by its meta-model, which defines how the language is logically structured. A stored model definition in the Modelica language, for example, is defined as the root element of a document, which can be a source file containing the model starting with an optional within statement that defines in which package the model is contained. Inside the stored definition, packages, classes, sub-classes etc. are defined according to the rules of the abstract syntax.

The MLSpec [Modelica Association, 2012] defines an accompanying concrete syntax grammar for textual representation of the language. Parsers use the grammar definition to recognize the elements of the language and to build a tree based representation of Modelica models. For OneModelica, the textual DSL Modelica is defined using Xtext [Köhnein & Efftinge, 2014] by a grammar file. Xtext generates specialized editors, views, parsers, etc. based on the provided grammar files. The generated editors recognize syntax errors and are able to provide code completion proposals based on the grammar definition, assisting the developer in creating syntactically correct code.

A textual representation of DSLs is one way of model representation. Another is a graphical notation that can be defined to visualize and edit models with the help of images and graphical primitives (ellipses, rectangles, lines, ...). A popular example is the graphical notation language Unified Modeling Language (UML) [Fowler, 2003]. It defines graphical elements like rectangular boxes for classes or lines between classes representing associations.

EMF-based models can be edited with the framework’s tree-based editors. The standard serialization is based on the XMI file format. However, if graphical representation is desired, powerful frameworks like Graphical
Modeling Framework [Gronback, 2009], Epsilon [Kolovos et al., 2013] or Graphiti³ allow the definition of graphical notations similar to the previously mentioned UML for custom meta-models defined with EMF. This results in three ways how DSLs can be viewed and edited:

1. Abstract syntax without graphical notation,

2. Abstract syntax with textual concrete syntax, and

3. Abstract syntax with icon based concrete syntax.

Including these, there are many ways of editing where the language is defined with EMF based meta-models. Hence, it is possible to use the appropriate way of representation for each kind of data. The data can still be processed in a similar way since the underlying data model is the same. In the OneWind project, the first form of DSLs without graphical notation for the parameterizable components of wind turbines is used for EDD models. Besides, the tree based editor that is provided by EMF, a more convenient and extensible editor based on SWT composites has been implemented which can be extended easily by providing new composites via a OSGi service. The textual DSLs currently supported are Modelica, the data format “ANSYS Parametric Design Language” ⁴, a definition language for airfoils, a language for Modelica test case and test suite definitions, and languages for role based model validations.

### 2.2.3 Using Textual DSLs

For the textual syntax definition of Modelica, the tool Xtext is used, which is part of the Eclipse Modeling Project. Xtext uses a EBNF-like syntax to define language grammars. Based on this grammar several components are automatically generated (Figure 2.3). A tokenizer splits the given text documents into parts that can be interpreted by a parser. The parser that is generated by the parser generator framework ANTLR [Parr & Quong, 1995; Parr, 2007] creates an AST and a Concrete Syntax Tree (CST) of the given text document. Thereby, an editable tree-like data structure that can be processed by additional software components can be obtained. The AST represents the structure of a Modelica document. The CST additionally keeps all information about the concrete representation inside the document, such as literals and white spaces. Usually, only the AST is used for processing, for example when validating the correctness of a model. The CST is used by the Xtext serialization to retain the formatting of the textual documents. Xtext allows to manipulate the AST. If, for example, an element is added to the AST, the new part is formatted and inserted into the existing CST of the model. Hence, the formatting of the existing elements is retained and the new element is inserted into the formatted code.

---

³https://www.eclipse.org/graphiti/
⁴http://www.apdl.de/
Figure 2.3: Creating Modelica-specific components and syntax trees with Xtext

Other generated components make use of the AST. There is, for example, an outline view that represents the inner structure of opened documents, such as inner classes, sub-packages and component declarations. One of the main amenities of Xtext is the integrated resolution of references (called linking). Based on this mechanism, the developer can jump to class definitions to investigate implementation details. References are also used as types for component declarations or as extended classes. The references connect ASTs of different documents and thereby form a kind of overlay graph. If referenced objects cannot be found, the affected part of the document again is marked with an error.

However, because of the complex structure of Modelica documents and for performance reasons, special index and linking mechanisms had to be implemented. Based on the index information, additional help functions were implemented, such as the proposal of classes that can be referenced at a certain position inside the document.

### 2.2.4 Xtext Grammar for Modelica

In the following the usage of Xtext for grammar definitions is explained by a small example. The listed code is a small excerpt from the grammar definition. The complete grammar is defined in Appendix A. Xtext grammar definitions start with the keyword `grammar` followed by the full qualified name of the grammar (Listing 2.2). This name is used during generation
of classes as the package declaration. Terminal symbols can be declared as hidden and will be ignored by generated parsers.

```plaintext
1 grammar on.modelica.xtext.Modelica
2  hidden(WS, ML_COMMENT, SL_COMMENT)
3
4 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
5 generate modelica "http://www.onewind.de/language/Modelica"
```

Listing 2.2: Xtext header in grammar files

For Modelica, white spaces and comments are semantically irrelevant and are thus ignored by the parser for the analysis of the document structure when creating the AST. The actual textual representation including spaces, comments etc., are stored in a second tree, the CST. This is for example necessary to preserve the formatting when serializing the model.

Other meta models can be imported for reuse in Xtext. this is indicated by the `import` keyword. For Modelica the basic Ecore model shipped with EMF is used to express basic data types like Strings or Integers. Properties of the generated Ecore model are set in line 5 starting with the keyword `generate`. The Ecore model gets the name `modelica` and the Uniform Resource Identifier (URI) is set to `http://www.onewind.de/language/Modelica`. URIs are used to identify the language definitions. It is often changed when a new version is released to distinguish between different versions of the same language. This would be necessary if the Modelica language is released in version 4 with a new structure that breaks backwards compatibility.

The next element is the data rule for import clauses (Listing 2.3).

```plaintext
1 ImportClause:
2   'import' (name=SimpleName name='=')?
3   importreference=Reference
4   wildcard?='.*'?
5   comment=Comment?;
```

Listing 2.3: Modelica import clause definition with Xtext

Note that by convention the first data rule is defined as the entry rule for parsing documents. However, for the sake of readability, most of the data rules are omitted at this point.

An import clause in Modelica extends the scope of visible objects in a class. Single classes as well as full package contents can be imported. The import clause definition in the grammar starts with the keyword `import`. Keywords are defined with single or double quotes. The `import` keyword definitions are followed by an optional abbreviation definition. Rules can be defined as optional by appending a question mark symbol to the rule.

In many Modelica documents SIunits are imported with the definition of an abbreviation: `SI = Modelica.SIunits;`. This allows the ability to write shorter expressions when using type definitions from the SIunits package. Hence, the user can write `SI.Angle` instead of the fully qualified name `Modelica.SIunits.Angle` and still knows that the SIunits
defined in the MSL are used. This abbreviation is defined in line 2 in the Listing. Line 3 defines the reference to the imported element. If a package is referenced, it is allowed to use a wild card (line 4). When a wild card is used, all subclasses of the referenced package are imported. This can reduce the number of necessary import statements in a Modelica document. An optional comment completes the ImportClause data rule. The data rule for References, as used by the ImportClause is defined in Listing 2.4.

```
1 Reference:
2   (prefix=AddOp)?
3   qn=QualifiedName?
4   (reference=[Component|ID]| reference=[AbstractContent|ID])
5   (arraysubscripts=ArraySubscript)?
6   ("." componentreference=Reference)?;
```

Listing 2.4: Reference definition with Xtext

The reference may be used in expressions and thus can be preceded by an operator. Line 3 defines an optional QualifiedName. The name is only set when the reference uses a qualifier, like for the import reference to the type Modelica.SIunits.Angle the qn attribute is set to Modelica.-SIunits.. The reference in line 4 can either point to a component or to a class declaration (AbstractContent). AbstractContent is another grammar rule that represents different kinds of Modelica class contents, such as the standard class structure or an extends clause. Using a reference to AbstractContent does not respect the fact that only packages can be referred to, when using wild cards. In this definition it is not distinguished between Modelica classes, models, packages and so forth, on the syntax level. Therefore, the correctness of the rule has to be provided by the semantics of Modelica. Syntactic and semantic rules are surveyed in Chapter 3.

The reference is indicated by square brackets and indicate for the Xtext framework that the reference needs to be resolved. The terminal ID after the pipe symbol says that the syntax of the reference must conform to the character sequence defined by the terminal rule ID. Hence, the character sequence must start with a literal or underscore and can continue with arbitrary literals, numbers and underscores. References to component instances can contain array subscripts if the component represents a set of value as an array. If a single component of the array is selected by the subscript, another reference can follow.

```
1 SimpleComponentName returns ComponentName:
2   name=ID;
```

Listing 2.5: SimpleComponentName definition with Xtext

The Comment data rule in Listing 2.6 consists of two elements: A string comment that is used to describe components and classes by a short text
in quotes and an annotation. Annotations are usually used for detailed documentation and graphical definition and layout.

Listing 2.6: Comment definition with Xtext

The curly braces have the effect that the parser will always create a Comment object if it is referenced by a data rule. This is needed to prevent potential null pointers access since all attributes of the Comment are optional. Finally Listing 2.7 shows the definition of the terminal ID.

Listing 2.7: Terminal ID definition with Xtext

Altogether, the Modelica grammar definition consists of over 100 rules (see Appendix A).

2.2.5 Modelica Meta Model

The previous section explains how the grammar file for Modelica has been defined with Xtext. From the grammar file an Ecore-based meta model is generated that is used to provide the features of OneModelica. This section provides an overview of the structure of the meta model. The Modelica language is complex and hence a simplified language will be discussed at this point, which is shown in Figure 4.7.

ASTModelicaSourceFile represents the root of a Modelica document and is the entry point for the parser creating the parse tree. A Modelica document can contain several classes which are contained in the reference with the name elements. The within statement that defines in which package the classes of the document are defined is saved inside the String attribute importedNamespace. The source file references classes of type AbstractMoClass which is an abstract super type of the different class concepts in the Modelica language (e.g., MoClass, MoModel or MoRecord). Other class types like MoOperator are omitted for simplicity in the diagram. The operations of the language elements are added during code generation. They cannot be defined inside the grammar definition with Xtext. The operations are used to simplify the utilization of the AST. The operation getAllComponents(), for example, returns all instance declarations of the flattened model, such as the components that are declared inside an extended class. Operations like this are frequently used, for example when validating models against language compliance.
Figure 2.4: Simplified Modelica meta model
The AbstractMoClass owns an object of type AbstractContent which is again an abstract class. Four types of contents are defined for Modelica:

1. EnumerationContent contains elements of an enumeration. For example the StateSelect enumeration allows to define whether a variable should be used as a state or not. Five states (never, avoid, default, prefer and always) are defined inside the enumeration.

2. An AssignmentContent reuses a previously defined content and allows to adjust the content for a new purpose. The Length type defined inside the Modelica.SIunits package of the MSL, e.g., reuses the class declaration of Real and sets the quantity to “Length” and the unit to “meter” to represent the SIunit Length.

3. The ExtensionContent extends a previously defined class similar to the AssignmentContent and allows to further extend the class with additional content inside the body declaration, like new variable declarations or equations.

4. A CommonContent is commonly used for the declaration of new classes. Variables, equations, etc., are defined inside the body declaration. Classes with a CommonContent can also extend other classes by the use of ExtendsClauses.

Body contains elements that define the state and behavior of the class. Algorithms are declared inside AlgorithmSections and equations are defined inside EquationSections which need to be transformed to algorithmic code by a simulation tool. A ConnectClause is a special equation that connects two variables based on Kirchhoff’s first and second law. The two variables are set as a Reference. An Annotation inside the Body is usually used for documenting the class and for the definition of the graphical representation in connection editors. ImportClauses influence the visibility of class declarations in other packages. Hence, classes can be instantiated directly without the need of using full qualified names. This increases the readability of Modelica code and improves refactoring. ImportClauses can be abbreviated, therefore a ComponentName can be set that can be referenced inside the class. The ImportClause as well as the ExtendsClause have a reference to a class declaration. References are resolved and can be followed inside the Modelica editor. Components own one or more ComponentNames which have a type. The type is a Reference to a class declaration. Comments are used for Components to provide a brief description. Additionally, an Annotation of the comment can contain more information, like the graphical representation of the component inside a connection editor or detailed documentation. Expressions are, for example, used in Equations. The types of the elements that are used in the Expressions need to be type compatible which is modeled by a
type system. The operation `getModelicaType()` returns the type of an Expression during validation. For details see Chapter 3.
Chapter 3

Static Validation of Modelica Models for Language Compliance

Enhanced validation of model code is desired during development to give immediate feedback to improve the correctness of the created models regarding compliance to the MLSpec. Two kinds of validation can be distinguished - dynamic and static validation. Dynamic validation of Modelica models is, at the moment, a difficult topic because models have to be processed by the development environment, for example by interpretation or incremental compilation [Fritzson, 1984]. For instance to ensure that the value of a parameter does not exceed a specified value, the equations altering the variable must be calculated. But currently these calculations are done by translation of Modelica source code to a different programming language like C or C++ and execution of the resulting program code.

However, static validation is currently done for a wide range of models (e.g. Modelica, UML, Java, ...). Several techniques are available for the validation of models. One could write custom Java-Code that performs the validation of models or utilize specialized languages such as the Object Constraint Language (OCL) [OMG, 2012], which is available as an Eclipse plug-in.

The IDE OneModelica can be used by engineers to create large scale Modelica libraries for different purposes like wind turbine development. The requirement for immediate feedback on errors arose due to the extent and size of the developed libraries. Changes in a model can affect others, but since full validation is only done during simulation, errors are often detected late and therefore the cause of the errors may not be obvious any more.

Subsequent fixes can introduce further semantic errors leading to more design iterations. The errors discussed here are caused by violation of rules
Chapter 3. Static Validation

defined by the MLSpec. Since models are usually composed of various components and extension of components is allowed and desired, validation of models is often expensive. Models concerning a wide range of extended models need to be flattened in order to be validated. Hence, validation of models needs to be performed sufficiently well. Additionally, fast constraints need to be distinguished from slow constraints in order to be checked by separate triggers - expensive constraints may only be checked when the user saves the edited document or even may need to be triggered by hand, while fast constraints can be checked during editing. For the checks existing compilers could be employed, but since the OneModelica tool is aimed at being independent from any third party tool, this solution is not applicable. Additionally, the Modelica code needs to be parsed and the models linked by any tool used, which would lead to performance loss and increased consumption of resources and may cause delayed error feedback during editing.

Another aspect of model validation appeared at Fraunhofer IWES, especially for engineers unfamiliar with Modelica. Many errors are made by combining predefined components, for example from a library, that from a physical domain point of view do not fit together. This is because Modelica is a language for mathematical modeling and thus does not restrict the use for domain specific design aspects. There is no support yet for the definition of semantic rules regarding the combination of model components, which is of special interest for frameworks. At the moment, two arbitrary Modelica components can be connected to each other, no matter if they fit, as long as the connector types match. This can either lead to a model that cannot be simulated — for example because it is structurally singular — or to a physically incorrect model. In this case, the error messages do not provide sufficient help for the user as the source of errors are on a physical level that cannot be captured by library unspecific error messages.

This Chapter describes an approach for static model validation, enforcing the rules defined by the MLSpec. The rules of the MLSpec relate to the meta model of Modelica and thus need to be explicitly defined. Additionally, methods are described that were added to the meta model classes in order to simplify the access to model data, which is mainly used when flattening models. The constraint definition language OCL is introduced and a selection of constraints checking the conformance of models to the MLSpec is explained. Furthermore, the implementation of the constraints in Java is explained and compared to OCL regarding readability, re-usability and performance. Further rules regarding establishing structural constraints are discussed and the implementation of a prototype with Java for wind turbine models is presented. The structural constraints discussed in this chapter are particularly defined for the Modelica language and cannot be reused with other languages. A more flexible way of defining constraints using role models is presented in Chapter 4.
3.1 Validating Modelica ASTs

Utilizing the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009] is advantageous since additional tools built for this Framework can be used directly. As an example, implementations of OCL interpreters are available, supporting the interpretation of OCL constraints on any language whose meta model is based on EMF.

The Modelica meta model that is the basis for the constraint definitions is described in Section 2.2.5. Classes extending other classes inherit the declared model components and the behavior defined by equations and algorithms. This makes the validation of objects harder since the inherited attributes need to be taken into account. Collecting all attributes and behavioral elements and merging them into one class representation is called flattening.

Each model component has a type defined by its class declaration. Again, no restrictions are made by the syntax definition of the language. However, this needs to be restricted since type compatibility must be enforced, especially when using equations that reference instances of such model components. It is important to check whether the types fit in order to be able to do calculations. Furthermore, connect equations connect components to each other that must be instances of a connector class [Modelica Association, 2012]. A type system has been implemented for Modelica that simplifies the validation of expressions regarding compatible types (see Section 3.4).

The class concept in the grammar is not very restrictive. Hence, each specialized class (class, package, function, record, block, ...) can from a syntactic point of view contain other classes, algorithms and so on. The restrictions and enhancements of specialized classes are instead defined in the MLSpec in textual form. Hence, constraints needed to be extracted and must be checked on instances of the meta model. By parsing a Modelica document and representing it as an AST, it is possible to check the semantic constraints on that representation.

Since the parse tree can be large, querying is a bottle neck and makes the definition of constraints complex and error-prone. When using an interpreter like OCL, querying will significantly slow down the validation process. Since OCL performs badly when processing large tree-based data structures, methods for easier access to the elements of the AST were implemented in Java. Xtext provides the possibility of adding methods to elements of the meta model with the language Xtend (see Section 2.2.5). The methods that were added mainly implement flattening of classes, which means that they collect all components defined in a class including its inherited classes. With the help of the additional methods, the size of the constraints is significantly reduced and the readability enhanced. Details about the validation of Modelica models are discussed in Section 3.2.
3.2 Validating Modelica Language Specification Compliance

This section describes the static Modelica model validation regarding correctness as defined by the MLSpec. Two methods of constraint definitions are compared and implemented using both OCL and Java. The two approaches are compared to each other in order to check whether dedicated languages for constraint definition, like OCL, are better suited regarding readability and re-usability compared to general purpose languages like Java. Finally, the performance loss caused by interpretation of OCL constraints on models compared to Java is analyzed. The constraints that are validated are arbitrarily chosen to reflect a wide range of constraint types of the MLSpec.

3.2.1 OCL Constraints

OCL was initially designed by OMG to constrain UML diagrams. The standard was later extended and can now be used with various meta modeled languages. Invariants can be specified, checking if a condition is met by a context object (the object being validated). Furthermore, queries can be defined for collecting and analyzing structured data. OCL is also used in transformation languages for the definition of transformation rules between two meta models. In this work, constraints are defined by invariants that sometimes make use of queries in order to collect elements of the Modelica AST. Three OCL constraints are explained, enforcing correctness of Modelica models for the following rules defined by the MLSpec [Modelica Association, 2012] (numbers in parentheses denote the page of the definitions):

- Operators may only be placed in an operator record or in a package inside an operator record [Modelica Association, 2012, page 42].
- A function can have at most one algorithm section [Modelica Association, 2012, page 135].
- A stream connector must have exactly one scalar variable with the flow prefix [Modelica Association, 2012, page 175].

The three constraints are used because they present different kinds of constraints that a) check in which program part the context object can be used, b) analyze what language elements the context object is allowed to define inside its content and c) check whether the conditions of a conditional aspect is met.

The first constraint (Listing 3.1) analyzes whether an operator is located in a permitted enclosing class. The MLSpec defines that an operator can only be declared inside an operator record or inside a package that itself is defined in an operator record. Therefore, declaring an
context MoOperator

inv operator_only_in_record_or_package:

let cls: AbstractMoClass = getAbstractMoClass()

in

not cls.oclIsUndefined() and
(cls.oclIsKindOf(MoRecord) and
cls.oclAsType(MoRecord).operator)
or
(cls.oclIsKindOf(MoPackage) and
cls.parentIsOperatorRecord())

Listing 3.1: OCL constraint restricting the use of operators

operator elsewhere (e.g. inside a function) is not allowed. The context object of this constraint is defined by the keyword context and is only applied to objects of the stated type. An invariant is defined by the keyword inv and assigns a unique name to the invariant that is later also used as an identifier to retrieve a comprehensive error message in case of a violation. A local variable is defined by the keyword let and, in this constraint, represents an object of type AbstractMoClass.

The method getAbstractMoClass() is implemented using Java, as described in Section 2.2.5, and returns the enclosing class of an AST element. In the case that the operator is defined at the top level of a document the object may be null. Hence, a check by the built-in OCL function oclIsUndefined() needs to be performed resulting in an error displayed indicating that the restriction is not met. The next part of the OCL constraint validates whether the enclosing class is of type MoRecord and whether the operator keyword is used for that instance. Another allowed use of the operator is when the enclosing type is a MoPackage and the package’s parent is defined inside an operator function. This is implemented by a separate function defined using OCL Listing 3.2:

context AbstractMoClass

def: parentIsOperatorRecord(): Boolean =

not getAbstractMoClass().oclIsUndefined()
and getAbstractMoClass().oclIsKindOf(MoRecord)
and getAbstractMoClass().oclAsType(MoRecord).operator

Listing 3.2: OCL helper method for checking whether a class is an operator record

The second constraint, as shown in Listing 3.3, analyses whether a declaration of a MoFunction contains at most one AlgorithmSection. The syntax definition in the MLSpec makes no distinction between the different class concepts. Hence, the definition of several algorithm sections inside a
context MoFunction
inv function_no_multiple_algorithms:
let b: Body = if content.oclIsKindOf(CommonContent) then content.oclAsType(CommonContent).getContentsBody()
else content.oclAsType(ExtensionContent).getContentsBody()
endif in b.oclIsUndefined() or b.bodyelements->select(oclIsKindOf(AlgorithmSection))->size()<2

Listing 3.3: OCL constraint restricting functions to have at most one algorithm section

function syntactically conforms to the MLSpec and thus is not marked as an error by the parser. However, despite not being described by the grammar, this is still syntactically incorrect according to the Modelica language rules and must be prevented.

For validation, the body of the context object is selected. Bodies are only available in type CommonContent and ExtensionContent (see Section 2.2.5). If no body is available (checked by b.oclIsUndefined()), the constraint cannot be violated. If a body is available, all elements of type AlgorithmSection are selected from the list of elements defined inside the body. This is done by the built-in OCL operation select. It can be applied to collections and selects all elements satisfying a user defined boolean expression. Here every object of the list is analyzed to be of kind AlgorithmSection. In case the resulting collection contains more then one object the constraint is violated.

The third OCL constraint in Listing 3.4 checks whether a stream connector has exactly one scalar variable with flow prefix. A stream connector is a class of type connector defining a component (of type Real or an extension of type Real) that is prefixed with the keyword stream. In this case exactly one component with the keyword flow must be present inside the class declaration. The constraint first collects all components of the connector object inside the variable components. For convenience, this method (getAllComponents()) again is implemented in Java, since it is used frequently and thus an inefficient implementation may lead to slow processing of the constraints.

After retrieving all components it is checked whether any of the components is a stream variable. In this case, all components that are flow variables (cIsFlow()) are analyzed if they define exactly one variable name. The count of variable names needs to be taken into account since multiple
components can be defined by a list expression in Modelica. Additionally, all components that are flow variables are selected in order to assure that the size of the resulting collection is exactly 1. The OCL helper functions cIsStream() and cIsFlow() are defined in Listing 3.5:

```java
1 context Component
2
def: cIsFlow(): Boolean =
3 (not connectorprefix.oclIsUndefined())
4 and connectorprefix.isFlow()
5
def: cIsStream(): Boolean =
6 (not connectorprefix.oclIsUndefined())
7 and connectorprefix.isStream()
8
context ConnectorPrefix
9
def: isFlow(): Boolean =
10 (not value.oclIsUndefined()) and
11 value = 'flow'
12 def: isStream(): Boolean =
13 (not value.oclIsUndefined()) and
14 value = 'stream'
```

Listing 3.5: OCL helper methods checking whether a component is a flow or stream variable

### 3.2.2 Java Constraints

To compare the performance of the interpreted language OCL to a general purpose language, the constraints are also defined using Java. The structure of the constraints is somehow comparable. However, it is possible to optimize the performance, since local variables and conditional return statements can be used. A Java constraint that has a similar structure as the corresponding OCL definition is displayed in Listing 3.6.
public boolean isValid(EObject eObject) {
    MoOperator operator = (MoOperator) eObject;
    AbstractMoClass enclosingClass = operator.
    getAbstractMoClass();

    if (enclosingClass != null) {
        if (enclosingClass instanceof MoRecord
            && ((MoRecord) enclosingClass).
            isOperator()) {
            return true;
        } else {
            if (enclosingClass instanceof MoPackage) {
                enclosingClass = enclosingClass.
                getAbstractMoClass();
                if (enclosingClass == null) {
                    return false;
                } else {
                    if (enclosingClass instanceof MoRecord
                        && ((MoRecord) enclosingClass).
                        isOperator()) {
                        return true;
                    } else {
                        return false;
                    }
                }
            } return false;
        }
    }

    return false;
}

Listing 3.6: Java constraint restricting the use of operators
public boolean isValid(EObject eObject) {
MoConnector conn = (MoConnector) eObject;
boolean isStream = false;
int numberScalar = 0;
for (Component component : conn.getAllComponents()) {
    if (isStream && numberScalar > 1) {
        // break if too many scalar variables
        // are defined
        return false;
    }
    if (cIsStream(component)) {
        isStream = true;
    } else {
        if (cIsFlow(component)) {
            numberScalar += component.getComponentnames().size();
        }
    }
}
if (isStream && numberScalar != 1) {
    return false;
}
return true;
}

Listing 3.7: Optimized Java constraint checking the stream connector restriction

In contrast, Listing 3.7 displays a constraint that benefits from the additional language constructs of Java.

The iteration over components of a class can be interrupted when a violation is detected since the number of scalar components can be checked at every iteration. The collection of components needs to be iterated only once since both conditions can be checked inside the loop. Hence they are checked to see whether a component has the prefix stream. As soon as more than one scalar variable is found, a violation of the constraint is indicated. The methods cIsStream() and cIsFlow() are implemented with Java checking for the connection type of the component similar to the OCL functions previously defined.

3.2.3 Language Concept Comparison

When the constraints are compared it is obvious that the readability of OCL constraints is very good for rules that can be defined in a short form. The stated context makes it obvious which object type is being constrained. Local variables can be defined and used for the validation of the context object. Functions allow the definition of re-usable common functionality. Multiple constraints can be defined in the same file allowing the accumulation of con-
straints targeting the same context object in one document. On the other hand, Java as a constraint language can be understood by more software developers. Furthermore, object oriented development is more common to developers than the functional programming representation of OCL. It is possible to define local variables inside the constraint definition causing validation to be quicker, since checks can be done inside loops to return a validation result early.

The most obvious benefit of Java as a constraint language compared to OCL is the tool support. Although OCL editors exist, they mostly lack support for automatic refactoring and robust referencing or only support syntax highlighting. But these features become vital when the meta model of a language that is being constrained is altered.

Hence, it may be beneficial to use OCL constraints for the restriction of languages where the grammar definition is finally set. On the other hand, constraints defined with Java may be a better solution when the grammar is still under development or high performance is required when validating models.

### 3.2.4 Performance

When validation takes place while a user develops a model, high performance is vital. In this section all constraints defined using OCL are compared to their equivalent constraints defined with Java. The performance time is compared by validating the Modelica standard library\(^5\), which contains models for various fields of physical modeling. The library contains most of the language constructs defined by the MLSpec and hence provides good feedback regarding the performance of the constraints. The included models are complex and frequently extend classes, causing more effort in resolving references and elements that need to be considered during the validation.

The MSL used for the performance measurement is version 3.2 beta 5. It consists of:

- 1432 Functions
- 1282 Models
- 694 Types
- 651 Packages
- 302 Blocks
- 289 Classes
- 278 Records
- 108 Connectors
- 3 Operators

\(^5\)https://modelica.org/libraries/Modelica
This includes the definitions of base types such as Real, Integer or Complex that are actually not included in the library but are added for convenience in our IDE. Parsing all 220 files takes approximately 6200 ms. Linking (resolving references, e.g., references between used components and their declaration) is performed in about 14000 ms. This may be enhanced in the future since the linking mechanism is not optimal regarding performance at the moment. Table 3.1 contains the measured performance results of 10 of 37 available constraints in both OCL and Java. The number of calls and the execution time for the invoked constraints are stated. The bottom line displays the number of calls and the overall performance for all 37 constraints. The validation was performed on a computer with an Intel Core i7 870 CPU (4 cores, max 2.93 GHz) with 8 GB of RAM.

As can be seen from the results, there is a tremendous performance difference between both kinds of constraints. This would be even worse if the flattening of classes was performed by OCL instead of using the helper methods implemented in Java as mentioned in Section 2.2.5.

The differences in performance originates in the higher efficiency of Java when handling collections. This is particularly apparent when investigating the performance of the constraint unique_element_names_comp since the implementations have to iterate over lists of components and compare the names in order to check whether a name has been used multiple times. This validation is performed for each of the 5039 types of classes. Since the check needs to be done for flattened classes to check if an extended class already defines a component with the same name, the number of components can be very high.

For less extensive constraints in which few context objects were called,

Table 3.1: OCL and Java validation performance of selected constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>OCL</th>
<th>JAVA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Calls</td>
<td>Time [ms]</td>
</tr>
<tr>
<td>unique_element_names_comp</td>
<td>5039</td>
<td>9896</td>
</tr>
<tr>
<td>protected_variables_dot_reference</td>
<td>225240</td>
<td>3842</td>
</tr>
<tr>
<td>prefixes_structured_component_flow</td>
<td>22186</td>
<td>97</td>
</tr>
<tr>
<td>function_no_multiple_algorithms</td>
<td>1432</td>
<td>77</td>
</tr>
<tr>
<td>flow_subtype_of_real</td>
<td>22186</td>
<td>76</td>
</tr>
<tr>
<td>stream_only_in_connector</td>
<td>22186</td>
<td>28</td>
</tr>
<tr>
<td>stream_connector_exactly_one_flow</td>
<td>108</td>
<td>2</td>
</tr>
<tr>
<td>function_no_equations</td>
<td>1446</td>
<td>1</td>
</tr>
<tr>
<td>nested_when_equations</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>operator_only_in_record_or_package</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>94677</td>
<td>20063</td>
</tr>
</tbody>
</table>
OCL performs sufficiently well. The constraint `function no_multiple_algorithms` that validates the 1432 functions of the MSL (defined in Section 3.2) takes only 77 ms. But even for this simple task, Java performs almost 4 times faster. The performance comparison shows the drawback of an interpreted language compared to a compiled one. Overhead caused by the interpretation and, in the case of OCL, the lack of efficient collection handling and early returns on violation of invariants reduces the performance.

With the fact that about 200 restrictions were found in the MLSpec, it is obvious that the performance will not be sufficient for automatic validation during editing with the recent interpreter implementation of OCL included in the Eclipse MDT project. Even for Java, if the user does not want to be disturbed by long waiting periods for validations taking place while editing Modelica models, focus on high performance is needed during further implementations of the missing constraints.

In OneModelica the constraints that were implemented with Java are used. All constraints are defined as “normal” constraints. This means that the constraints are automatically used for validation by the Xtext framework when a document is saved or a project is being built. Validation of language compliance can also be deactivated by the user.

### 3.3 Validation of Structural Constraints

As mentioned before, the validation of constraints that originate from the physical properties of the model is very beneficial for the user. In the following section, existing components of the OneWind Modelica library according to [Strobel et al., 2011] are used to introduce these kind of constraints. If library developers would provide domain-specific constraints for their models, the intended use of the models could be enforced. By standardising the constraint definition, for example by providing OCL constraints inside annotations, all Modelica tools could benefit from the added semantics. However, for the proposed solution the tools must be able to process the same meta model of Modelica, since the constraints are defined based on the AST representation of Modelica models.

#### 3.3.1 Possible Model Structures for a Horizontal-Axis Wind Turbine

Using the OneWind Modelica library, a conventional three-bladed, horizontal axis wind turbine can be modeled through parametrizing and connecting components that have a specific physical meaning. The structure of such a model is displayed in Figure 2.1.

As can be seen in Figure 2.1, the `Rotor` object is connected to the `Nacelle` object, which is then connected to the `Tower` object. However, a user of the library could directly connect the `Rotor` object to the `Tower`
object. In both objects connector instances are used which makes this connection correct according to the MLSpec. From an engineering perspective, however, this is not reasonable. Furthermore, the components of the OneWind Modelica library are not set up to cover this case, although there is always the possibility to modify the library components correspondingly.

If the user made this direct connection of Rotor and Tower object with the existing library components, the simulation of the model would result in an error due to a division by zero in the tower shadow calculation. However, this error only shows up when the model is already compiled. This takes valuable time from the model development. Depending on how experienced the user is in the field of wind energy, the given error message does not even give a direct hint about the true source of the error.

### 3.3.2 Introduction of Structural Constraints

To support the library user during model development, a constraint that gives a warning can be defined to avoid the direct connection of Rotor and Tower objects. This is realized using the constraints checking implemented in Java. For the sake of brevity, only the constraint of the Tower object needing to be connected to the Nacelle object is covered. The library developer can define this constraint himself using simple annotations directly in the Modelica code during the development of library components.

The definition as well as the appearance of the constraint in the graphical user interface of OneModelica is shown in Figure 3.1. The library developer defines the topFrame of type Frame.b in the PartialTower model as a restricted element through the annotation shown in the upper right side of Figure 3.1. If the user tries to connect the Tower object to a Rotor object as is shown in the upper left side of Figure 3.1, an error message will appear (cf. bottom part of Figure 3.1). The connection of the Tower object to the Nacelle object does not result in this error.

However, if the user decides that he or she would like to connect an object to the Tower object that is not of type Nacelle, he or she can avoid the error message by defining another annotation. This is also shown
in the upper left part of Figure 3.1. In this way, the user is not prohibited from using the library components according to his wishes. However, the constraint helps especially unexperienced users to avoid mistakes which is of high importance.

For this kind of constraints the following steps needed to be performed. For the context type \texttt{ConnectClause} a Java validator has been registered. The validator then checks whether one of the connected elements defines a constraints inside its annotation (\texttt{constraints=“constraintname”}). If this is true, the annotation of the other connected element is queried and it is checked whether it defines a valid constraint accordingly (\texttt{constraints=“constraintname_valid”}). If the constraint does not exist, an error marker is created.

### 3.4 Modelica Type System

The MLSpec defines predefined types that can be used by the developer and must be provided by any Modelica IDE. In OneModelica the type definitions of the MLSpec are directly integrated into the MSL. This way links to the declaration can be resolved which allows the user to investigate the declaration. 5 types are defined by the MLSpec \texttt{Real}, \texttt{Integer}, \texttt{Boolean}, \texttt{String} and \texttt{Enum}. New types can be defined by the Modelica developer. If the newly defined type extends a predefined type, such as Modelica.SIunits.Length extends the type \texttt{Real} and changes the quantity and unit value, then the predefined type is inherited. In that case variables can be checked for compatibility in expressions. In OneModelica two additional types are defined:

1. **NoType**: The type used in an expression does not inherit from a predefined type.

2. **UnresolvedType**: The type cannot be determined since is cannot be resolved. This happens when a reference to a type declaration is not visible or existing. In this case a linking error is indicated and the missing type will have no effect on the validation of the expression.

3. **Incompatible**: Used for expressions that use incompatible types. If an expression is incompatible the parent expressions are not checked to prevent from redundant error marking.

#### 3.4.1 Type Caching

Types that can be affiliated to predefined types are commonly used in equations and algorithms, for example when SIunits from the MSL are used. Hence, type checks must perform well since they should not significantly reduce the validation performance. Therefore, the Modelica meta model has been modified in the way that type information can be added to expressions.
within ;

model Expressions
  
  import Modelica.SIunits::*;

  Length length = 1.0;
  Length result = 0.0;

  algorithm
    result := length + true;

end Expressions;

Listing 3.8: Simple Modelica expression

The time consuming part of type investigation is that the inherited classes of each variable must be analyzed until a predefined type is found.

The type information is stored directly in the AST if it has been found after the first analysis. Once the information is found, for example it is detected that the SIunit Length is of type Real, this information can be reused by other type declarations like Distance that inherits from Length. Changes inside this inheritance hierarchy must be recognized by the type checking. Storing the information directly inside the AST is beneficial in this case. The type information keeps references to the objects that provided the type information. If the Length declaration is changed, the parse tree is updated. This results in a new object representing Length in the AST. Hence, the reference from the type information stored in Distance cannot be resolved any more. During validation it is checked if all extended type declarations still exist. If not, the information is updated. The benefit of this approach is that in the standard case where the type hierarchy has not been changed, only the directly referenced declarations must be checked. In the example of distance, the reference from Distance to Length is checked and subsequently the reference from Length to Real.

3.4.2 Type Validation

Listing 3.8 displays a simple Modelica model with an expression. Two variables of type Length which extends the predefined type Real are defined. The Assignment in line 10 assigns the sum of variable length and the boolean value true to the variable result. Figure 3.2 displays the assignment and how it is parsed in simplified way.

The validation triggered by Xtext starts at the leaf nodes of the parse tree. Hence, the type information for the variable length and the boolean keyword true is retrieved and stored inside the AST. Next the expression length + true is evaluated. The two types of length and true which are Real and Boolean in the example are fetched and is checked whether the types can be used with the operator +. Since this is not the case an
error is indicated and the type information of the Expression is set to Incompatible.

The next element from the AST that is validated is the variable result. The type information is set to Real. Finally, the Assignment is validated. It is checked if the type of the expression length + true can be assigned to the type of the variable result. Since the type of the expression has been set to Incompatible, no error is indicated to prevent from duplicate error marking.

### 3.5 The International System of Units

The International System of Units [Thompson & Taylor, 2008] (abbreviated Si units) is an international system for units. It defines 7 base units (metre, kilogram, second, ampere, kelvin, mole and candela). More units are defined by the combination of the base units, like kg m/s² for Newton. The declaration of the predefined type Real in the MLSpec has an attribute for the unit type that can be defined as a String. The MSL contains a package for SI units that declares the base units as well as many derived units. The user can reuse units from the package or create custom ones by combining base units. For validation the information about units needs to be cached again for better performance. The information is stored as a vector of the base units. The unit information is modeled with EMF and is attached directly to the AST of the parsed documents. The update mechanism is the same as previously explained for the base type checking in Section 3.4.2. The unit information provides integer attributes for each base unit to represent the unit vector as exponents of each base unit. An example of the unit vector for Newton is defined as:
Unit{
  m=1 kg=1 s=−2 a=0 k=0 mol=0 cd=0
}

When two units are multiplied, the resulting unit is calculated by addition of the two vectors. Torque, for example, is defined by the multiplication of Force (which has the unit Newton) and meter resulting in the vector:

Unit{
  m=2 kg=1 s=−2 a=0 k=0 mol=0 cd=0
}

When a unit is divided by another unit the vector of the divider is subtracted from the divisor. For addition and subtractions of units, the units need to be equal. This needs to be checked by comparing the unit vectors in order to provide the user information about the correct use of units.

Figure 3.3 displays a simple Modelica model. The SI units Length and Height are used that are defined in the MSL. The first equation in line 12 multiplies the height and length variables which results in a unit vector where the meter unit is set to 2 and all other units are 0. The resulting value is assigned to the variable noUnit which is a simple Real. Since the unit does not match, an error is displayed.

The second equation in line 13 also multiplies length and height but this time the result is assigned to the variable customArea which is of type Real but sets the unit to the correct value. Instead of the custom declaration, the predefined Si unit Area could also be used.

3.5.1 Limitations of Unit Validation

Although it is desirable to always use units and thus be able to see when expressions are faulty, units may be neglected in some cases. In the modal element of the OneWind library (which is used for fast simulation of flexible structures [Thomas et al., 2014]), for example, matrices of values are used for calculation that use different units. The matrices are defined as basic Real values. Unit neglection is occasionally used in algorithms, hence the validation is not strictly performed in cases where values are assigned to basic Real variables.

For built-in functions the unit of the input is usually reused as output unit. However, exceptions exist, like for the function \texttt{der()} that returns the input unit divided by time. For unit validation it would be helpful if Modelica would provide the possibility to the user to define which unit he or she expects from a function, for example by some kind of cast, and that unit could be used during validation of expressions. For user defined functions the return type is retrieved from the output variables that have been defined inside the functions.
3.6 Code Style Validation

Additional validations have been added to support the developer in models with high quality. The validations include checks if conventions are adhered to, such as the file name of a Modelica document should be the same as the first class declaration therein (except for package.mo files) or that the code is sufficiently documented. Since offending the conventions does not lead to code that cannot be simulated, problems are marked as warnings. The following checks have been implemented:

- A Modelica file shall have the same name as the first class declaration inside the file.
- The `within` statement shall be represented by the folder structure of the defining file.
- A component declaration shall provide a comment for documentation.
- A class declaration shall provide an annotation with documentation.

Figure 3.4 displays a Modelica model that violates all these constraints. Quick fixes have been implemented that assist the user in correcting the errors. In the figure two fixes are provided, the first one adjusts the `within` statement to match the folder structure, the second one moves the file to
the folder that matches the within statement. For the first constraint two fixes are available: Renaming the first class declaration to match the file name or renaming the file to match the first class declaration. For the comment warning of components a comment stub can be added and for the class documentation an annotation is added.

3.7 Related Work

RestrictED [Seifert & Samlaus, 2008] provides constraint checking with OCL for arbitrary DSLs defined with EMFText. Constraints are defined as queries, collecting all objects violating the constraint in the parsed tree of the document. The constraints can be defined by the user during runtime. However, defining queries instead of invariants can be misleading. Moreover, it is harder to define them such that all erroneous objects are collected. This limits the complexity of usable languages and only a few constraints were defined for the example languages.

ModIM [Höger, 2012] — a front-end tool for processing Modelica models — allows to statically analyze models based on a syntax tree representation. For the analysis the visitor pattern [Gamma & Beck, 2003] is applied and custom analysers can be implemented that visit the nodes of the syntax tree. A sample implementation is presented that performs a type check. However, the performance of the tool is not addressed, it is only stated that redeclarations perform badly with the current implementation.

In [Lochau & Günther, 2008] an aspect based validation framework is presented that allows to define elements of the Modelica language that are involved in the validation (join points). The elements can be queried by an aspect language (point cut expressions) and an action language can be used to define what shall be done with the elements of the models (advice). The languages are defined by re-using paradigms of logic programming and can
be transformed into a format that can be evaluated by Prolog. The provided examples show how custom rules like naming conventions and the number of classes defined inside a package can be checked.

For the validation of equation-based components focusing on numerical inconsistencies [Bunus & Fritzson, 2004] proposes a graph-based methodology that provides users with information about under- and over-constrained equation systems. However, the general validity regarding the MLSpec is not aimed at.

Attribute grammars [Knuth, 1968] add context-sensitive aspects to the syntax of a language. Parse trees can be annotated with attributes which can be propagated up and down the parse tree. Adding operations for parse tree querying and type recognition imitates the functionality of attribute grammars. Together with operational semantics [Pierce, 2002] the correctness of expressions can be checked automatically. The attribute grammar implementation JastAdd is used by the Modelica tool JModelica [Åkesson et al., 2010].

3.8 Conclusion and Future Work

The validation of Modelica models is possible by the definition of constraints that check models on the basis of their tree based representation (AST). The constraint language OCL and the Java language are utilized for the validation. It becomes clear that many constraints can also be checked in an efficient way. By implementing the flattening of classes with Java, the extension of the constraint definition is reduced and the validation process accelerated. The integration of the validation is possible and can significantly enhance the development of Modelica models, since errors can be immediately displayed to the user.

However, it is pointed out that the interpretation of OCL constraints is time consuming, although the AST access has been enhanced. Therefore OCL should only be used for fast constraints. Since Java is up to 6 times faster when validating all currently available constraints, it is advantageous to implement the remaining constraints using Java in order to keep the performance fast. Another way to gain performance could be achieved by transforming the OCL constraints to Java code [Wilke, 2009].

Beyond validation against the MLSpec, structural constraints can be checked. This allows developers to define restrictions preventing errors that are obvious to library designers but that may lead to problems when done by library users. The approach may also allow the developer to restrict the usage of components that are known not to be compatible but can not be restricted by the modeling language itself.

In future work more constraints will be implemented for the validation against the MLSpec. If all 200 constraints found so far can be implemented in an efficient way, validations can be recognized immediately by the developer. Furthermore, compatibility for various simulators can be established
more easily since violations of the MLSpec that are accepted by some simulators can be easily identified.
Chapter 4

Static Source Code Analysis of Physical Model Semantics using Role Models

The development of models for simulation and assessment has become an important task in the design of new physical systems. Simulations need to be performed in order to assess the suitability of a given design in regards to desired performance and durability. Simulation results are used to demonstrate compliance to industry standards and for the certification of system components. Recurring design cycles demand an optimal structure for various aspects like cost reduction, optimal performance, or safety. In this Chapter, the different kinds of design aspects to be taken into account are expressed by role models. Reenskaug [Reenskaug et al., 1996] states that role models provide a way to describe the collaboration between objects. While objects in object-oriented languages have a certain state and behavior that is analyzed during development, the collaboration of objects is usually not considered. Objects can perform several roles, for example the generator of a wind turbine converts torsion from the drive train into electrical power but it also influences the vibration frequency of attached components like the nacelle and its mass. The collaboration of objects for these different aspects can be modeled with role models. Recent modeling languages that are used for the development of physical models, like Modelica, assist engineers efficiently in creating component-based models through a domain specific syntax. By separating models into components they can easily be distributed and ultimately reused by other engineers. Free libraries (e.g., the Modelica standard library) are available for commonly used components while commercial libraries providing highly detailed models can be
purchased for special purposes (e.g., the power train library [Otter et al., 2000] from DLR).

Aside from textual representation, physical models can also be created graphically by using a Modelica connection editor or specialized graphical modeling languages like ModelicaML.

ModelicaML [Schamai, 2009] is a UML-Modelica profile for the UML2 standard and thus builds on an industry standard that is enriched by stereotypes for a domain specific behavior. ModelicaML also allows generation of simulatable Modelica code from annotated UML models. At this point ModelicaML is used as a second language since it defines physical models in a different format (UML instead of Modelica) but yet represents the models in a similar way by using component-based structures (and ultimately generates identical simulation code). However, the herein presented validation framework can potentially be used with any model that is based on the EMF.

Models of both kinds, Modelica and ModelicaML, are connected in a similar way due to their component-based structure. Components are coupled by linking two connectors to each other. Only two connectors of the same type are permitted to be linked. Frame connectors from the Modelica standard library are used for the mechanical systems shown in the examples. While this is the only restriction for the application, incorrect specifications may result when connecting two components that do not fit together semantically. At best, the user will get an error message about the faulty model from the simulation software. For large models it can be time consuming to detect and fix such connection errors since Modelica models must first be translated into a computable form (usually C code) and the modeling error may occur in an advanced stage of the simulation. The presented framework allows the user to check for such errors during the development of physical models. Figure 4.1 displays error markers and error descriptions for incorrectly connected model components in the Modelica editor.

Outside of the structural requirements on the correctness of physical models, additional design aspects need to be considered during validation. Wind turbine models for example are defined by connecting different kinds of sub-components to each other. The typical wind turbine (see Figure 2.1) consists of a tower that is connected to a nacelle. The nacelle contains a drive train that is connected to the generator which generates electrical power from the torque produced by the rotor. The drive train is connected to a rotor mounting one or more rotor blades (the typical design has three rotor blades).

While the experienced modeler knows the structure of wind turbines and hence may immediately recognize incorrectly connected components when they are visualized in a connection editor, the same user may have problems seeing the errors in a textual editor or when creating highly detailed components. Furthermore, when using model components from libraries, the library designers may want to restrict the way in which the model compo-
Figure 4.1: Semantic errors marked in the Modelica editor.

Components are used in order to prevent the creation of models that could not be constructed in real life.

An example in the area of wind turbine modeling is the existence of components of the same kind but parameterized in a way that makes them incompatible to components with differing parameter sets (this is also called a multi-point constraint in [Assmann et al., 2005]). The rated power of a turbine requires that components like generators and hubs are designed for this specific power rating, e.g., 5 MW or 10 MW. Physical models of generators thus may be structurally identical but different parameters make them specific for a certain rated power class. In Modelica this may be expressed by naming conventions for the models (e.g., Generator5MW and Generator10MW) indicating special parameter sets to the user. This convention becomes problematic when further aspects of a model design must be expressed and automatic validation is desired.

A real world example, where the validation of different parameter sets for structurally equal models is important, are load calculations for the certification of wind turbines based on different simulation set-ups. From the experience of our engineers this can be an error-prone job when simulating more than a thousand load cases for a single wind turbine design. A load case defines which environmental parameters such as wind speed and maximum wave height, are used for a simulation and how the structure should respond to the resulting loads. It can happen quite easily that components are not suitable for a certain load case which will result in redundant simulations that slow down the certification process significantly.
The proposed role model\(^6\) based validation system allows these aspects to be expressed explicitly and models to be validated automatically. Hence, adding more semantic information to models can point modelers to incorrect design decisions, the development process can be accelerated, and the resulting designs are more likely to meet the design criteria.

The concept of role models tackles a problem with instances of language elements: they do not provide information for certain design aspects concerning their collaboration within a program. Role models define design aspects explicitly and language elements can be annotated with the roles that they perform within an aspect. Therefore, the use of language elements can be further restricted for design aspects that apply to the current program code.

### 4.1 Concept of the Role Validation Framework

This section addresses the structure of the role-based validation framework. The technologies currently in use are listed and languages that have been implemented to allow the definition of role models, constraints (Role Model Constraint Language (RMCL)) and mappings (Role Model Mapping Language (RMML)) are explained. Finally, the validation mechanism is described.

#### 4.1.1 Supporting Various Modeling Languages

Eclipse is used as the base technology for its ability to add custom functionality through a plug-in mechanism (see Figure 4.2). The EMF provides an

---

\(^6\)or role-constraint model, since the purpose in the framework is to define constraints on the collaboration of objects
environment for the definition of DSLs and a validation framework that is
designed for use with models created with the DSLs. Language definitions
(i.e., metamodels) define an abstract syntax that is detached from a concrete
syntax. Hence, models created with EMF can be displayed generically with
the provided tree or diagram editors. Furthermore one or more concrete
syntaxes can be defined for an abstract syntax by defining custom icons for
diagram editors or providing a textual representation that allows parsers to
create an AST from a textual document.

OneModelica provides parsers which create EMF based syntax trees from
Modelica code. Hence, Modelica models can be used with the role model
validation framework presented herein. The second use case is UML2 models
which are annotated with domain specific information provided by Modeli-
dicaML. The UML2 language\(^7\) is also defined with EMF, which means that
UML2 models created with this language can also be used within our frame-
work.

4.1.2 The Validation Framework’s Languages

Two languages have been developed for restricting models with constraining
roles:

1. The role model constraint language RMCL (Section 4.2) allows to
define role models and roles for arbitrary context objects. Context
objects trigger validations and need to be mapped to entities of models
that are being analyzed. As a starting point context objects have
ports that link to roles which are queried from the context object .
Additionally, restrictions (rules) are defined with logical expressions
to specify how the roles can be used in context with each other. In the
examples shown, the connection of component based physical models
are restricted so that the connections made are the context objects and
the connected components are queried for the roles that they perform.
The RMCL creates an abstract restriction model which is independent
of a target model.

2. The role model mapping language RMML (Section 4.3) defines which
instances of language elements (e.g., a Modelica ConnectClause or
UML2 Connector inside a class diagram) can be associated with con-
text objects defined through the RMCL. Queries for each port of the
context object need to be defined for the modeled language to obtain
the roles from the validated objects. The roles need to be defined as
strings so that they can be associated with the previously defined roles
by the validation algorithm. This is done through annotations in Modeli-
dica models or through comments in UML2 models (see Section 4.5
for example implementations).

\(^7\)www.eclipse.org/uml2/
By applying these languages it is possible to define roles and constraints for arbitrary EMF-based data structures. For each metamodel (each language being used for the development of physical systems, e.g., Modelica, UML2 or Petri nets) the mapping has to be defined once and can be re-used by other developers. As a last step, models need to be annotated with the roles that they perform.

### 4.1.3 Performing the Validation

Now that the validation framework has been set up, models can be defined and connected to each other. The framework registers an EMF validator that validates all documents of mapped languages present in the workspace, i.e., languages for which mapping to a role model is available. The validation of edited models is automatically triggered when a model is saved or when the workspace is being built. Alternatively a validation can be triggered manually. Roles that are defined in the model elements are obtained and the constraints checked based on the role data. Error markers indicate violations of rules in creating valid models to guide developers.

### 4.1.4 Granularity of Rule Definitions

For the validation of component based models with connectors and connections, two different kinds of role definitions can be distinguished. They differ in the granularity of the defined roles that components can perform.

- The general role describes which two components can be used with each other. As an example, the role Tower for a tower component and the role Rotor for a rotor component cannot interact with each other when targeting structural validity. A nacelle component is missing which is the linking object between the structures.

- In addition to the component-wide roles, fine grained roles for single ports of a validated object are needed. This enables the definition of rules that would allow only the top frame of a Tower to be connected to a Nacelle whereas the bottom frame requires connection to a Substructure or Ground.

Where only general restrictions are possible, the distinction between top and bottom connections could not be expressed. In that case, given rules would allow the connection of the Tower to Nacelle, Substructure, and Ground. In this case, a connection between the bottom frame of a Tower and the frame of a Nacelle would be valid but unrealistic.

Since the user can define which objects in his or her models should be annotated with roles and how the roles can be queried, he or she can define different types of granularity as needed. Also the use of the validation framework is not restricted to component-based models and thus provides a flexible way for the definition of restrictions for arbitrary model types.
4.2 Defining Role Models and Constraints

This section explains how the RMCL can be used to define validation context objects, role models, roles, and constraints. The language has been created with Xtext (Figure 4.3 displays the metamodel) which automatically generates a text editor with syntax highlighting, syntax checks, and name-based reference resolving. This supports the editing process by guiding the user through defining correct role model definitions.

4.2.1 Context Objects, Role Models, and Roles

Role models describe the collaboration of objects of a program for a certain design aspect like the structural validity of physical models or the suitability for specific environmental conditions. Role models define a name and contain roles that can be performed by components. Since each role has a name, qualified names (e.g., "rolemodelname.rolename") can be used to annotate models with the role that they are performing. Rules can be defined for the roles inside a role model to restrict the collaboration between objects. In order to create role models, the first task for the role designer is to define a context object which triggers validations and which will be marked with error markers in case of rule violations. Context objects can have arbitrary ports which will be used for querying roles. For component-based models the context object could be a binary connection with a left and a right port, pointing to the connected components. Listing 4.1 displays an example from a RMCL document defining a context object with two ports.

Now that a context object is defined and available, the second step is to define role models, roles, and rules. Role models contain a collection of roles that represent a certain aspect of validation. The examples in Section 4.5 showcase structural and power design aspects. The restriction through roles is necessary since the languages used do not limit the connection of components sufficiently. For example, it is possible to connect any Modelica components to each other as long as the connectors are of the same type. This can lead to the creation of incorrect models. Listing 4.2 displays an example of a role model with two roles and one rule, targeting the previously defined context object.

---

Listing 4.1: Defining a context object and ports.

```
1 2 3 4 5
context ContextObject {
  port portA
  port portB
}
```

---

8In the case of Modelica, the connection is modeled as a ConnectClause and the validated connection object is created by a parser
Figure 4.3: RMCL metamodel.
In the first line a role model with the name RoleModelName is defined and is used as a qualifier for the roles that are contained within the role model. The next section, starting with the keyword roles, contains all roles that are known to the role model. This results in qualified role definitions (in this example RoleModelName:RoleA and RoleModelName:RoleB). Roles are defined by the keyword role followed by a characteristic name and an optional comment describing the intent of the role to the user.

The relationship between roles could also be defined using the Object Role Model (ORM) diagram notation. In the proposed framework facts are expressed using logical expressions. This makes it possible to automatically validate the rules defined by the logical expressions.

### 4.2.2 Rules

The third step is to define rules for context objects which restrict how the roles can collaborate. Roles can be retrieved from the context object’s port. In Listing 4.2 the rule says that objects performing RoleA can only interact with objects performing RoleB. This is done by a logical statement saying that if a query on the context object for portA retrieves the role RoleA, then a query for the port portB must retrieve the role RoleB. For bijectivity the statement is defined in a second statement and combined with a logical and. In this example no other role will be allowed to interact with RoleA and RoleB. The following logical statements are currently supported for rule definitions by the language (corresponding to propositional logic):

```
rolemodel RoleModelName {
  roles
    role RoleA "Comment for RoleA"
    role RoleB "Comment for RoleB"
  rules for ContextObject {
    (portA=RoleA implies portB=RoleB) and
    (portA=RoleB implies portB=RoleA)
    ERROR "RoleA can only connect to RoleB"
  }
}
```

Listing 4.2: Defining role models, roles and rules.
and Returns TRUE if the left-hand side and the right-hand side evaluate to TRUE. Otherwise it returns FALSE.

or Returns TRUE if the left-hand side or the right-hand side or both sides evaluate to TRUE. Otherwise it returns FALSE.

implies Returns TRUE if the left-hand side of the expression evaluates to FALSE or the left-hand side and the right-hand side evaluate to TRUE. Otherwise it returns FALSE.

not Negates the subsequent expression.

The statements and and implies were used in the example of Listing 4.2. For the support of cardinalities in rules further language elements are available. Rules can start with for all (roleA=RoleA or roleA=RoleB) to select a subset of context objects that are defined inside the same parent object and that are associated with the stated roles. For example in Modelica, all connections that link to a component with a certain role can be selected from a model.

The relational operators (<, <=, >, >=, ==, <> ) are used to compare two cardinalities with each other. Cardinalities can either be Integers or ranges (e.g., 1..3). The function card(rolename − portA,portB) is used to select roles from multiple ports of a context object. The first argument of the function defines the role name that is selected. The list of arguments following the dash defines which ports of the context object are considered in the selection as described above. The number of selected roles can then be compared using relational operators to check if the cardinality is permitted.

Examples of the use of cardinalities in rules can be found in Section 4.5. Every rule must define a severity that corresponds to Eclipse’s resource error scheme (error, warning, or info). Additionally, an error message providing a description of the problem for the user must be defined. Erroneous objects are marked with the provided information to inform the user directly within editors. Additionally, Eclipse provides a separate view for displaying all errors that have occurred.

4.3 Mapping the Role Models to Target Languages

The previous section showed how role models, roles and rules can be defined with the RMCL. Since the RMCL-language elements are not bound to specific models and can be reused, mappings to target languages must be provided.
4.3.1 Relation of Role Models and Modeling Languages

Role models refer to context objects which are the starting points for validation. Context objects need to be mapped to elements of the modeling language. This is a one to one relation. The context objects define ports which are used for the definition of rules. The ports point to roles which have to be queried from instances of the modeling language. Hence, starting with the context object, the user needs to define a query that collects all role definitions from model instances. Queries are defined within role mapping documents and are used to collect role definitions from model instances (see Section 4.3.3).

When a context object is validated, the role definitions are collected and grouped so that all roles belonging to a role model can be accessed during validation. The role models that refer to the context object are then selected and the rules that are defined for a role model are validated with the role definitions that have been found. Figure 4.4 displays the mapping from context objects and roles to language instances.

4.3.2 The Role Model Mapping Language

For each target language, a mapping must be provided. The metamodel of RMML shown in Figure 4.5. Listing 4.3 displays basic statements of an RMML document. In the first line the target language is referenced by its name space URI. The second statement indicates which role model is being mapped. The statements are used to resolve references to elements of
Figure 4.5: RMML metamodel.

1. target "http://www.example.com/lang/ExampleLanguage"
2. maps "platform:/resource/Example/src/Example.role"

Listing 4.3: Defining the target model and the mapped role model.
Context mapping ContextObject = ExampleClass {
  left = LeftQuery,
  right = RightQuery
}

Listing 4.4: Mapping connections and ports to the target language.

the target language and the RMCL to immediately provide the user with information whether referenced objects exist and queries can be evaluated. Additionally, proposals for structural features of the context objects that are being mapped can be calculated by introspection, assisting the user during the mapping definition.

In Listing 4.4 the context object ContextObject from the role model example in Section 4.2 is mapped to a class definition ExampleClass from the referenced language. Instances implementing the class ExampleClass will trigger validation as described earlier. The query definitions of each port from the mapping document are used to receive the role definitions from the context object. In the example, the LeftQuery and RightQuery are used to receive the role definitions for the left and right ports from the ExampleClass instance. The queries ultimately need to retrieve an object of type String that contain one or more role declarations.

4.3.3 Querying Roles from Context Objects

A custom language has been created for the definition of queries. There are several existing languages that could be used for this purpose since they allow querying of EMF-based models [Gogolla & Richters, 1998; Eclipse Modeling Framework Project, 2014; Kolovos et al., 2013], but they have some drawbacks in handling large models as described in Chapter 3. The present language therefore tries to simplify the querying of large models. The user defines how the roles can be received from model instances. This is based on the model’s structure as defined by it’s metamodel. Thereby, the models are analysed regarding their static semantics as defined by the roles. If an error occurs while querying a model, an empty set of roles is returned. Possible errors include a query receiving an object that is not of type String, or a referenced object that does not exist and thus would lead to a potential Null pointer access. Another simplification is to not differentiate between single objects and collections of objects. Queries can be defined in the same way and there is no need for special set operations. However, selections are still possible like those for objects of a certain type.

4.3.4 Query Language Concepts

Listing 4.5 shows the RMML concepts that are currently available to the user. Line 1 displays a simple query. A structural feature is selected from
an object, for example from the context object at the beginning of a query. A structural feature in EMF is an attribute, a reference, or an operation of a modeled class. The type of the object may not be statically known if the object has been retrieved from a reference with an abstract type definition. In Modelica, for example, equations in an equation section can be of type ConnectClause or EquationExpression (see Figure 4.7). Hence, the structural features cannot be statically derived. To avoid this, the user can define a cast to a specific type as demonstrated in line 2 if he knows that only one type will appear.

In line 3 a function is called that needs a return value. Two types of functions exist in the RMML. One type is modeled using EMF and is generated or hand-written in the generator’s target language, e.g., Java. The other type of functions is defined directly with the RMML language and is used to gather recurring operations, like common queries. The currently queried object will then be provided as an input parameter for the functions.

In lines 4-6 a case distinction is used to select the correct query for the current object by type-checking the object. This is also used when a list of objects of different kind is being processed. In the example QUERY_A is selected if the investigated object is of type TypeA. The third entry states that QUERY_C is selected if the current object converted to a String equals the given value.

The last two lines handle lists of objects. While in line 7 all elements of a type are selected, in line 8 a single object at a specific index in the list is selected.

4.4 The Validation Algorithm

It is important to understand in regards to the validation algorithm that the role model validation framework keeps track of all RMCL and RMML documents inside the workspace. As soon as a document is added, changed, or removed, the information concerning the affected resource is updated in the registry. As a result, up-to-date information about all mappings, roles, and context object definitions are available during the validation of model instances.
Figure 4.6 shows the objects that are involved in the validation process. All documents for mapping and role definition which are present in the workspace are taken into account during validation. Since rules are interpreted new documents can be added dynamically. If standard rules and mappings are expected to be provided in the future, they can be added via an extension point. However, this feature is currently not implemented.

Two separate tasks occur during validation: The role model validator uses a query engine to receive role definitions (defined as text) from model instances. The logical validator uses the provided roles to check if the model element under investigation violates rules. The validation result is then passed back to the role model validator which will mark model elements with an error marker if a violation has been detected. The validation algorithm is described in detail below:

1. The EMF validation framework triggers the role model validation framework’s validator (role model validator in Figure 4.6) upon model change, workspace build or manual activation. The validation starts with the root element of a document and continues with all sub-elements until all model elements inside the document have been validated.

2. Initially all previously added error markers of the current element are removed.

3. The validator checks if a mapping from the model element’s class to a context object exists in the registry. If not, no validation will be performed for the object at hand and the algorithm continues with step 1.

4. The role model validator now associates the model element with the context object definition. For each port definition of the context object, the queries that are defined inside the role model mapping documents are performed by the query engine on the current model element. If role definitions can be found by the query engine, they are kept in a list that is associated with the port and passed over to the logical validator.

5. The role model constraint documents contained in the registry provide the rule definitions for the context object of the current model element. For each rule, the roles of the current model element are filtered in order to select only valid role names. For example, in Listing 4.2 the roles with the qualified names RoleModelName.RoleA and RoleModelName.RoleB are selected.

6. Valid roles as well as rules from the role model constraint documents are passed on to the logical validator. The roles are verified by the
logical validator according to the logical statements described in Section 4.2.2. The example shows that Listing 4.2 checks if the role RoleA is defined for portA. In this case, the RoleB must be defined for portB and vice versa.

7. The validation results are handed over to the role model validator. If the validation of a rule returns a negative result, the current model element is marked with an error marker that is provided by the Eclipse framework. The error message from the rule definition is used to describe the type of error.

4.5 Evaluation

Within the domain of wind turbines many design aspects need to be considered when highly detailed models are created (e.g., the need for sealed components for offshore wind turbines). As the ability of collaboration between components grows, it becomes more important to check the compatibility of these systems in order to assist the user in tracking all design aspects.

This section defines role models and constraints for two kinds of modeling languages: Modelica and UML2. The focus lies on two different design aspects. One that targets the overall validity of the model structure and
one validating that composite wind turbine models are designed for the same power rating.

### 4.5.1 Role Models for Structural Validity and Rated Power Design Aspects

As a first step, role models, roles, and rules need to be defined for the design aspects in question. Listing 4.6 displays three constraints enforcing correct connection of tower, nacelle, rotor blade, and hub connectors as well as one constraint that defines the need for a wind turbine to have a connection between a nacelle and a tower. Roles are defined for the top connector of a tower, the connector of a nacelle linking to a tower, a rotor blade connector, a connector of a hub linking to a rotor blade, and a wind turbine. The first two constraints define the following: only models with the role `Nacelle_To_Tower` can be connected to models performing the role `Tower_Top`; there must be exactly one connection between these roles. The third constraint defines that only 1 to 3 rotor blades can be connected to a hub. These restrictions represent typical instances in wind turbine design. It is generally understood, that a connection between a tower and a rotor is unrealistic. However, in large textual Modelica files with many connect equations, this kind of error might be hard to identify. In real life application the use of one, two, or three rotor blades is common for fast rotating wind turbines. Mounting additional rotor blades could lead to structurally unstable behavior.

Model components must fulfill requirements regarding reliability in order to get certified. Guidelines [Germanischer Lloyd, 2010; International Electrotechnical Commission, 2009] define load scenarios that need to be simulated and the specified parameters vary with the power rating of the wind turbine design. During the creation of simulation models it is easily possible to use incorrect components which would result in time consuming, erroneous, and unnecessary simulations being performed repeatedly to produce valid results. Listing 4.7 contains two simple constraints that target the rated power levels of 5 and 10 mega watt wind turbines. For different power levels the models usually only differ in magnitude of parameters. The drive train of a 10 mega watt turbine must bear more torque than the drive train of a 5 mega watt turbine which results in larger dimensions and a higher mass. When components are connected that are annotated with constraints, the power rating of each component can automatically be verified for compatibility. This will prevent a modeled drive train with a rated power of 5 MW to be used in 10 MW wind turbines. In such a scenario the torque generated by the rotor would exceed the maximum capacity of the drive train in strong wind conditions. In real world applications the components could not be connected because the flange of a 10 MW rotor would be too large to be mounted to a 5 MW drive train. In modelling, however, the difference in the components is most likely modeled by different parameter
rolemodel WindTurbineStructure{

roles

role Tower_Top "Tower top connector"
role Nacelle_To_Tower "Nacelle to tower connector"
role Rotorblade "Rotor blade root connector"
role Hub_to_Blade "Hub to rotor blade frames"
role WindTurbine "Wind turbine"

rules for BinaryConnection{

(left=Tower_Top or right=Tower_Top) implies
(left=Nacelle_To_Tower or right=Nacelle_To_Tower)
ERROR "The tower top frame can only be connected
to appropriate nacelle frames"

for all (left=Tower_Top or right=Tower_Top):
  card (Nacelle_To_Tower - left , right) == 1
  ERROR "The tower can be connected to exactly one
  nacelle"

for all (left=Hub_to_Blade or right=Hub_to_Blade):
  card (Rotorblade - left , right) == 1 ... 3
  ERROR "The hub connects exactly 1-3 rotor blades"
}

rules for MandatoryConnection{

mandatory=WindTurbine implies
(left=Nacelle_To_Tower and right=Tower_Top) or
(left=Tower_Top or right=Nacelle_To_Tower)
ERROR "A wind turbine must have a connection
  between a tower and a nacelle"
}

Listing 4.6: Roles for structural validity.
sets. Hence, the models of 5MW and 10MW drive trains are equal except for the parameters, which makes it difficult for non-experts to choose suitable components when creating new wind turbine models.

The combination of role constraints can be useful since structural rules and rated power related rules can be used side by side. This is even more useful when another modeling task is taken into account which concerns the location of the designed wind turbine. If wind turbines are located offshore special components, such as sealed nacelles, have to be used that can withstand the rough environmental conditions at sea. Components can be annotated with roles for offshore or onshore use which would allow the user to check whether a wind turbine model contains only components that are designed for 10 mega watt rated wind power and is suitable for offshore weather conditions. Hence, further rules can assist the user in creating correct models for numerous kinds of design aspects.

### 4.5.2 Mapping the Role Models to Modelica

Now that the roles and constraints are defined using the RMCL, mapping between the elements of the RMCL and the Modelica language needs to be defined with the RMML in order to validate the Modelica wind turbine models. We will show mappings to BinaryConnection roles for Modelica and mappings to MandatoryConnection roles for UML2. The Modelica language is complex and therefore we will discuss a simplified language (see Figure 4.7) at this point. The complete language definition can be found in textual form in Appendix B and C. ModelicaSourceFile represents the
Figure 4.7: Simplified Modelica metamodel.
Chapter 4. Semantic Validation

4.5. Evaluation

```plaintext
1 target "http://www.onewind.de/language/Modelica"
2 maps "platform:/resource/Roles/src/Contexts.role"
3
4 contextmapping BinaryConnection = ConnectClause {
5   left = ConnectClauseLeftQuery,
6   right = ConnectClauseRightQuery
7 }
8 RoleQuery ConnectClauseLeftQuery {
9   case ConnectClause :
10      left.selectAnnotations().text
11 }
12 RoleQuery ConnectClauseRightQuery {
13   case ConnectClause :
14      right.selectAnnotations().text
15 }
16 function Annotation selectAnnotations(Component){
17      return type.elements.all(Annotation)
18 }

Listing 4.8: Mapping for Modelica.
```

root of a Modelica document and is the entry point for the parser creating the parse tree. A Modelica document can contain several classes. Each class can have elements such as import statements, subclasses, and many more, but only EquationSections containing ConnectClauses are of interest in our example and therefore the other types of elements are omitted at this point. A ConnectClause has two references (left and right) to Components that are defined inside classes and which are of type Connector. ConnectClauses are defined as context objects in the RMML document and will trigger validation. The roles are retrieved by following the references left and right to the connected components that are of type Connector. Now the annotations of the components are analyzed for Strings that define roles. Annotations can be quite complex as well, so be aware that the reference text is a shortened expression which actually retrieves the role definitions as a String from the Annotation. Listing 4.8 displays the Mapping for the simplified Modelica language. The RMML document starts with the declaration of the target language via its URI (Uniform Resource Identifier). In the second line the document containing the roles is referenced. Then a contextmapping is defined stating that a BinaryConnection in the RMCL document corresponds to objects of type ConnectClause.

For the ports left and right queries are provided which retrieve role definitions from the context objects. The role queries start with a case statement checking whether the object provided for query is of type ConnectClause. If this is true the left and likewise the right references to a component are queried and the function selectAnnotations is called which obtains the retrieved Component as an input. In this function the elements of the component’s type are taken, the Annotations are selected
and then returned. Then the text reference of each Annotation - which is a String - is returned.

Figure 4.1 from the introduction displays a screenshot of the Modelica editor showing parts of the specification for a wind turbine model. The last connect equation attaching frame number 4 of the hub to the frame of blade number 4 in line 18 is marked with an error since it exceeds the limit of a maximum of 3 connections. The subsequent connect equation violates the constraint which says that only one connection between a tower and a nacelle can exist. Finally the last connect statement violates the restriction that the tower top frame can only be connected to the appropriate nacelle frame.

### 4.5.3 Mapping the Role Models to UML2

The same rules used above for the Modelica models shall now be re-used with wind turbine models created with ModelicaML. Figure 4.8 displays an extract of the UML2 language definition. Listing 4.9 starts with a reference to the UML2 language definition and the role definition file that contains the objects to be mapped. The MandatoryConnection context object from the role definition is mapped to UML2 Class instances. For the definition of roles inside UML2 models comments are used that can be attached to UML2 models. The ClassRoleQuery is defined for the mandatory role which generates the body of the ownedComment from the Class. The left and right queries generate the UML2 connectors, which represent ports of classes that can be connected to other ports from the investigated objects. They take the left and the right end of the connection respectively (which

```
1 target "http://www.eclipse.org/uml2/4.0.0/UML"
2 maps "platform:/resource/Roles/src/Contexts.role"

contextmapping MandatoryConnection = Class {
  mandatory = ClassRoleQuery,
  left = ClassLeftPortQuery,
  right = ClassRightPortQuery
}

RoleQuery ClassRoleQuery {
  case Class : ownedComment.body
}

RoleQuery ClassLeftPortQuery {
  case Class : ownedConnector.end[0]
    .role.ownedComment.body
}

RoleQuery ClassRightPortQuery {
  case Class : ownedConnector.end[1]
    .role.ownedComment.body
}

Listing 4.9: Mapping for UML2.
```
Figure 4.8: Parts of the UML2 metamodel.
corresponds to the first and second entry in the list of `connectorEnds`). The roles are selected from the `connectorEnds` and again the bodies of the comments are queried.

Now that the mapping is defined, UML2 models can be annotated and checked against the rules previously defined. Figure 4.9 displays a wind turbine model in the ModelicaML diagram editor. The faulty connections are marked with an error icon. Additionally, the Eclipse problems view indicates which errors have occurred. Compared to tree and text based editors, the diagram editor has the advantage that faulty connections can easily be recognized visually without the need for additional validation.

However, this is not the case for errors regarding the permitted amount of connections since the user could possibly create arbitrary connections between the hub and rotor blades. The same holds true for power rating constraints where the types of components which are being connected to each other are of importance. Any errors in these instances cannot be recognized.
Chapter 4. Semantic Validation

4.6. Related Work

immediately by using diagram editors and demands for automatic validation.

In addition, rules were defined for the definition of operating conditions with low wind speeds, which are common for in-land turbines, and for operating conditions with high wind speeds, which are common for offshore turbines. Figure 4.10 displays a simple wind turbine model with annotated components in the Modelica editor. The two components gearbox and rotor in the wind turbine model HSWindTurbine can be connected to each other since the connection does not conflict with any constraints. The second connect statement however, is not allowed since the slowgearbox component is designed for in-country use with slow average wind speeds while the rotor is designed for high wind speed environments. The third connect statement violates the role constraints from Listing 4.7.

4.5.4 Results

Once a mapping has been defined roles can be reused with models of the mapped languages. The complexity of the language will determine the effort necessary for mapping. We were able to show that the framework can be used with different kinds of EMF-based models and error markers are created to display the inadequate use of components. The rules defined for structural validity assist our students in learning how to create wind turbine models with Modelica. Further rules need to be defined in order to restrict all important scenarios that exist in wind turbine modelling.

Other kinds of role constraints help to assist the user in selecting the correct components for a certain design scenario. The combination of different role models helps to build a strong validation framework that checks the model against various aspects. Hence, the selection of components from large libraries as well as the development of new models can be assisted especially when many variations of the same component already exist.

4.6 Related Work

The idea of role models that express the collaboration between objects was initially described by Trygve Reenskaug et al. [Reenskaug et al., 1996]. In the proposed Object Oriented Role Analysis and Modeling method, roles are used to describe the collaboration for certain design aspects and views are provided for presenting the objects collaborating for an aspect. Objects can play multiple roles where each role is qualified by aspects.

Dirk Riehle and Thomas Gross demonstrated [Riehle & Gross, 1998] that role models can be used for the design of software frameworks. They added semantics to the framework classes on how they can be used in collaboration by defining rules for the roles the classes play. As an example, they showed how the hierarchy for figures in a visualization framework can be defined by applying roles for parent-child relations to framework classes.
Figure 4.10: Example of a wind turbine model with rated power and wind speed constrained components.
Role models were successfully used for generic refactorings [Reimann et al., 2010]. The framework allows to define refactorings in a generic way e.g., to rename objects or to move extract objects into a new container object. Similar to this work, roles and rules for the refactoring are defined with custom language. The definitions need to be mapped to the meta-model of each target language that can be used with the refactorings. The applicability has been demonstrated with several DSLs.

Section 3.3 presents a first approach for semantic validation of Modelica models with OCL. OCL as a common language for static semantic validation of EMF models allows the definition of constraints that can be automatically validated by the EMF validation framework. Constraints are also defined for context objects and the validation is triggered in the same way as validation with the proposed role model validation framework. One drawback is that constraints require re-definition for each target language with OCL. When one rule needs to be changed, every OCL-constraint requires modification accordingly. Also meta model changes should be reflected in each constraint for the modified language since the constraints are defined directly based on the meta model. These problems have been avoided by the role model validation framework through separation of constraint definition and language mapping. Finally, it was mentioned that the performance of interpreting OCL constraints on large data sets has a negative performance compared to Java code.

When modeling physical systems, certain issues arise for component-based software systems and frameworks [McIlroy, 1968; Szyperski, 2002]. Components provide ports that can be connected to each other in order to interact. From the syntactic point of view, components can be connected in any way as long as the ports match each other (in [Assmann, 2003] arbitrary class members can be defined as ports for composition). [Assmann et al., 2005] more generally describes how framework instantiation languages should be defined to prevent users from misuse.

The use of model driven technologies and applications in simulation modelling is addressed in [Vangheluwe & de Lara, 2003]. Different techniques like model-based engineering, model driven software development and model transformation languages are explained and it is demonstrated how these technologies can be used to create models with DEVS formalism.

4.7 Conclusion and Future Work

The presented validation framework allows to create roles for physical components with the dedicated textual language RMCL. Constraints can be defined for restriction of how the components are connected with each other. By using the framework’s mapping language RMML, arbitrary domain specific languages can be used for the validation of these restrictions. The validation framework was evaluated with two types of modeling languages used at Fraunhofer IWES. One language is UML2 refined with ModelicaML
stereotypes. The other one is Modelica, a general purpose language in the field of physical modeling. Models defined with the two languages were successfully restricted for two design aspects: Some rules for structural validity and suitability for different rated power designs were applied.

Although this Chapter concentrates on the validation of component-oriented models by checking the connections between components, the framework could be used in other contexts. Instead of connections, references from one model element to another could be validated. The use of components can be restricted for object-oriented models, which corresponds to the approach presented in [Riehle & Gross, 1998].

In the future, the RMCL may be extended to support extension of roles by other roles. Thus, general restrictions could affect roles dedicated for special purposes (e.g., a role offshore tower could extend the regular tower role). This would reduce the effort of defining restrictions and the number of roles that need to be defined inside the models. Although the available restriction types are sufficient for the investigated component-oriented models, more expressions may be needed for a generalization.

A future topic of investigation is the change of roles depending on the state of the context object. When designing offshore wind turbines, the water depth plays a crucial role in the selection of the substructure [Jonkman, 2007]. For shallow water (up to about 30 meters depth) regular monopile structures can be used whereas in deep-water environments (60 meter depth and more) only floating structures are applicable.

For now only a small number of role models and constraints have been implemented which can help beginners to create correct wind turbine models. In the future, it would be beneficial to implement more rules for structural validity and to find other collaboration aspects where the framework can also aid experienced developers. It would be helpful for users to generate rules for the MSL. This would enable constraints to support a large number of users since the library is widely used. For large models with many constraints the performance of the role model validation framework needs to be investigated.
An important part of the process of developing physical models is model testing since it greatly increases the probability that the expected behavior is accurately modeled. Within software development it is convenient to create tests and test suites using test frameworks like JUnit\(^9\). The use of these test frameworks ultimately lead to a “test first” development approach, which means that tests are created before actually developing the software parts that must conform to the tests. Also, engineers creating physical models can benefit from a test driven development. Immediate feedback about incorrect model changes can assist in keeping the models consistent during their lifecycle in the modeling process.

By using fine-grained tests for single model components, errors in composed models can be traced back to the erroneous element. This also simplifies the development in teams as one modeler’s changes may have a negative influence on another modeler’s models. However, test-based development as known from software engineering cannot directly be transferred to physical model developed with Modelica. While the algorithmic parts of models could be checked in a similar way, the behavior expressed by equations needs to be simulated since time is used as a parameter. However, by using simulations test-based development can also be used when creating models.

A reasonable solution for testing whether the expected behaviour of models changed due to modifications is the use of regression tests where reference data is used as a base for comparison with results from subsequent simulations. When models are altered, the result can be compared to the reference files to check if the behavior is still as expected. The two commonly used

\(^9\)http://www.junit.org
Modelica simulation tools OpenModelica and Dymola use regression tests for checking if changes made in the compilers still result in valid simulations as well as to allow users to create regression tests for model development.

Although these kind of tests already aid the development process of physical models, regression tests may not be sufficient to find all problems that may appear during the development and the tools provide a non-uniform way of defining test cases. The test framework MoUnit aims at facilitating test case and test suite definition for single model components as well as groups of models. Besides, the tests should be executable with various simulation tools as well as in different environments, for example directly inside the Modelica IDE OneModelica, or in continuous build environments like Hudson\textsuperscript{10}, which is currently in use for OneModelica and OpenModelica development. Moreover, users should be able to extend MoUnit in order to reuse their already implemented test functionality as well as making it possible to create more sophisticated tests than just regression tests.

MoUnit allows the user to create test cases for automatic testing. The user can use arbitrary simulation tools and compare the results to reference data and to the results of a second experiment. In the current implementation result files need to have the same step sizes and the results for a variable can be compared using a user defined acceptable deviation. Thereby, it is possible to compare simulation tools to each other or investigate the impact of different solvers on the simulation results without the need to create additional reference files. Moreover, the extensibility of the test framework is demonstrated, for example through the implementation of a test component for regression test of equidistant result files. As a second test component the calculation of basic statistics has been implemented. This allows us to validate stochastic properties of our Modelica irregular water wave models by comparing to results from the commercial software ASAS WAVE\textsuperscript{11}. Here it is advantageous that several input formats are supported in order to use data from tools that are not Modelica-based. Thus, calculations from analytic formulæ producing numerical results can also be used for testing in the framework.

### 5.1 Testing Modelica Models Within OneModelica

Testing is a crucial part of model development. The following aspects of a model should be checked: whether it (a) can be simulated without errors, (b) delivers the expected results, and (c) represents the reality appropriately. Furthermore, when component models are developed and/or modified it also has to be checked whether the composed models deliver the expected results. That is, the robustness of a component model in various situations

\textsuperscript{10}http://hudson-ci.org/
\textsuperscript{11}http://www.ansys.com/Products/Other+Products/ANSYS+ASAS
has to be ensured as — depending on the complexity of the model — even a small change might lead to unexpected results. Testing a model in different situations can be a tedious process as it involves a lot of “playing around with parameters” and waiting for simulations to finish. All these points are reasons for automated testing during model development.

The testing strategy which is presented herein and from which the requirements for the development of MoUnit arose, is closely tied to the Modelica IDE OneModelica. In OneModelica, models are organized in model projects. This approach allows to have two model projects for each component model that has to be implemented. The first one contains all the physical system equations and model logic. However, the models are not yet simulatable because the parameters do not have values yet — as they should be modified by the user when the model is parameterized for various application scenarios.

The second one contains the actual test models. The test models contain instances of the models taken from the original model project. The modifications of the original models are done with parameter settings that correspond to defined test case scenarios. This could also be a special set of parameters that has caused problems in the past and it should be ensured for future simulations that it will not occur again. This strategy is somewhat borrowed from software development: JUnit test plug-ins will be separated from the actual Java plug-ins to separate program logic from the corresponding test. The use of instances of the models to be tested rather than extending it, is based on the fact that test models can also be seen as “best practice” examples: a look at the test models shows how the model to be tested should be used. However, if this aspect is not important, it is also possible to use a test model that extends the model to be tested within MoUnit. This would avoid possible inconsistencies between the physics and test model code that is duplicated, for example parameter units. By using the Modelica model examples for regular water waves, this is illustrated in Figure 5.1: The test model project has the same name as the original model project, but with an appended .test. This makes it clearly distinct as a test project for all the developers, who work on the models. As can also be seen in Figure 5.1, the test model project contains experiments that simulate the test models.

The structure of the model to be tested and the testing model is shown in Listing 5.1: the model RegularWave has the parameter instances and equations, while the model TestRegularWave has an instance of type RegularWave and modifies the corresponding parameters.

The experiments that are used to simulate the test models are set up manually. They can also be simulated manually during the model development. However, once the number of models and functions has exceeded some threshold, manual execution of all the experiments becomes too cumbersome. This is the point where MoUnit should be used.

The necessary files for automated execution of the experiments are stored
Figure 5.1: Model projects for both model logic and test models for regular water waves
model RegularWave
  "kinematics of a regular linear wave"
import SI = Modelica.SIunits;
import NonSI = Modelica.SIunits.Conversions.NonSIunits;

parameter SI.Height H "wave height";
parameter SI.Period T "wave period";
parameter NonSI.Angle_deg epsilon "phase";
// rest of parameters omitted

equation
  // model logic is placed here
end RegularWave;

model TestRegularWave
  "test model for 'RegularWave'"
import SI = Modelica.SIunits;
import NonSI = Modelica.SIunits.Conversions.NonSIunits;

countant SI.Height H = 5;
countant SI.Period T = 11;
countant NonSI.Angle_deg epsilon = 12;

RegularWave regularWave(H = H, T = T,
  epsilon = epsilon
  // rest of modifications omitted
) "instance to be tested";

equation
  // test logic like assert statements
  // is placed here
end TestRegularWave;

Listing 5.1: Example for a model and a corresponding test model
in the test model project as shown in Figure 5.1. These files enable checking that all the models that have been implemented in the original model project are fulfilling the three properties mentioned above: it can be easily seen if a simulation crashes and if the implementation of the model works correctly by comparing it to reference data that can either come from other simulation tools, measurements, or analytical solutions.

5.2 Implementation of the Test Framework

For the development of valid physical models with Modelica it is essential to define test cases to check that model changes do not negatively affect simulation results. A common way to assure this is using regression tests. This is used by the OpenModelica development group using their framework to test compiler changes by comparing results of simulations to reference data. In the same way Dymola's model management library allows performing regression tests with Modelica models.

The goal of MoUnit is to provide test-driven physical model development with Modelica. A special test language has been defined that can be parameterized with custom test components and test result listeners which are responsible for processing test results appropriately.

In the following, the developed test language is described. The language has been developed using the Xtext [Köhlein & Efftinge, 2014] system. It allows defining test cases and test suites for automatic execution. Additionally, the extension mechanism is explained to show how custom test components can be registered allowing users to write test cases for special purposes — for example, basic statistics analysis. Subsequently, the test engine that performs the test execution is described and the registration of listeners is demonstrated. The listener mechanism allows to use MoUnit in different environments, for example by displaying the test results visually inside OneModelica or by generating text files for the integration of tests in continuous build environments like Hudson. An ANT task has been implemented that is executed by Hudson. The task adds a listener to the test engine creating a XML file for the documentation of the test results. The result file can then be used by Hudson for error reports.

5.2.1 Test Definition Language

As described previously, Xtext is a tool for textual language definition and development (see Section 2.2.3). By providing a grammar definition of a language, editors and views can automatically be generated with helpful functionality like syntax highlighting and syntax checking. This is also applicable to the test definition language. Since mandatory components for automatic test execution in our IDE are defined as Ecore-models, those components can easily be reused for MoUnit.
Two components can be defined with the test language that are comparable to concepts used by the test framework JUnit for Java code: Test cases and test suites. Test cases (see Listing 5.2) define the basic elements that are needed for a test to run while test suites (see Listing 5.3) combine test cases and other test suites. Hence, a set of test cases can be collected to test the correct behavior of interacting Modelica models.

Test cases define names that make them referenceable and thus usable in test suites. The `experiment` keyword states that a Modelica experiment is used in the test. Modelica experiments in OneModelica can use several tools for simulation, including OpenModelica and Dymola. For regression tests a path to a reference file is provided following the keyword `file`. Several checks were implemented to display problems to the user, for example when the provided file cannot be found or the experiment is set to leave the Dymola window open after simulation for post processing (see Figure 5.2). This would cause the test execution to pause until the user closes the window manually. It is also possible to directly compare the test results of two experiments by using a second experiment instead of the reference file. This can help to ensure that the results of simulations with OpenModelica are the same as those obtained from simulations with Dymola. Another possibility is to check if the results are equal for different kinds of solvers.

Furthermore, two result files can be compared to each other, for example when it is desired to compare results from other modeling tools and languages. For two result files that contain results with the same step size the built-in tester can compare the results step by step with a user defined

```plaintext
suite AllTests:
    TestSuite1,
    TestCase1
```

Listing 5.3: Example of a test suite definition
deviation. The result files can be in MAT format as provided by OpenModelica and Dymola (textual or binary format) as well as in comma separated values format. Moreover, the user can also create results based on some analytical formulae and save them in one of the supported formats, if it is not desired to implement the analytical result as a test component and register it to the test framework.

Simulation result files can be referenced by absolute or relative paths. Using relative paths, files can be put into a project inside OneModelica’s workspace and can be synchronized by multiple users with source code management tools like SVN. Since simulation result files might be large, it may be necessary to use network drives for storage. This can be done by using absolute paths that can be prepended with the prefix $$. The prefix will be replaced by a user defined path. This way tests are re-usable by other users, and the prefix only has to be configured once per workspace.

Now that the data for comparison is available, the user needs to provide the qualified names of model elements that shall be compared, following the compare keyword. If no element is defined then all time series from the results are compared to each other. The user can use the asterisk character * to select groups of elements. In Listing 5.2 all values found with a qualified name starting with “qualifiedname.” are compared to each other. The definition \texttt{test.qn -> reference.qn} states that the value \texttt{test.qn} from the experiment result shall be compared to the value \texttt{reference.qn} from the provided file.
Next, test components are referenced that perform the comparison of results. The components are registered via an Eclipse extension point [Clayberg & Rubel, 2009; McAffer et al., 2010]. The test components must implement a predefined interface and need to be modeled using Ecore. Attributes can be defined which must be parameterized by the user. In the provided example a tolerance for the result comparison is defined which will allow a small difference in the result data of the compared time series.

Possible values for the parameterization are Doubles, Integers and arrays of Doubles and Integers. The attributes are looked up by introspection [Eckel, 2006] which is used to validate whether the user provides correct input. When the validation is performed a new instance of the test component is created and the parameters are set. Finally the user defines an error message that will be displayed when a test run fails.

Test suites start with the keyword `suite` followed by a name. They contain a list of test cases and additional test suites. All test cases contained in a test suite will be performed when the suite is executed.

### 5.2.2 Test Execution and Result Processing

Figure 5.3 displays how the MoUnit components interact with each other. The test definitions are processed by a Modelica test engine. The engine is implemented without dependencies to user interface components to allow usage in build environments that do not have graphical user interface installed (e.g., Linux operating systems on dedicated build servers). The test engine executes the experiments used in the test cases and compares the results to the referenced experiments or files. Listeners can be registered to the test engine to be informed about the executed tests and test results. The duration of each test is measured for performance analysis. The test components must return a status object that provides information about whether the test succeeded or failed. When a test failed, a message must be provided and an optional index (e.g., the time step) can be used to tell the user about the erroneous location in the result file.

Besides the basic information described above, a test component can return arbitrary objects with additional information about an error. This can, for example, be an SWT\(^{12}\) composite object for visualization inside OneModelica or a file path to a serialized plot which can be used for a test report. The test engine listener is responsible for checking whether it can handle the object or not.

For the visualization of test results in OneModelica a view (see Section 5.3) that is comparable to JUnit’s test view has been implemented. Test suites and test cases are displayed as a tree and the nodes are marked according to successful or erroneous execution. While JUnit test cases are based on assert statements that check whether particular parts of code run correctly, the result files need to be compared in order to check whether

\(^{12}\text{http://www.eclipse.org/swt/}\)
Figure 5.3: Structure of the Modelica test framework
the desired behavior is matched. This results in a different kind of test status visualization where the qualified names of erroneous models is provided and optional graphs can visualize deviations between expected and present values. A JUnit like behavior could be implemented for algorithmic code, possibly by interpreting the algorithms [Höger, 2013], allowing to check code very fast. However, this is currently not yet supported by MoUnit.

For the integration with automatic build environments an Eclipse ANT task\(^\text{13}\) has been developed. The task is performed on an Eclipse workspace that contains the Modelica projects as described in Section 5.1 with experiments and test definitions. The user provides information about which test cases and test suites shall be executed. For test logging a test engine listener has been implemented that creates XML files according to the format used by the JUnit ANT task. Hence, failing Modelica tests can be handled in the same way as JUnit tests, for example by sending out notification mails to the responsible model developers. The ANT task enables the use of the same test definitions in continuous build environments as well as for manually triggered testing inside OneModelica.

5.3 Test Examples

In the following, the example of linear water wave models is used with MoUnit. These models were developed at Fraunhofer IWES to display both regular and irregular waves according to Airy wave theory with corresponding stretching methods. The underlying theory for these models can be found in standard offshore engineering text books like [Faltinsen, 1993] or [Chakrabarti, 2005].

For evaluation purposes, the variable \(\eta\) corresponding to the wave elevation, which represents the instantaneous wave height at a given position, is used. For a regular linear wave, \(\eta\) is deterministic as it can be displayed by a cosine function. That is, it can be compared to reference data through a regression test. Accordingly, this is used as a first test case denoted as TestRegularWave.

The second test case is called TestIrregularWave and tests the variable \(\eta\) for an irregular linear wave. In this case, \(\eta\) is stochastic, which means that a direct comparison of time series to reference data is meaningless unless they are generated with the exact same input. However, this is not always possible, for example when the simulated time series are compared to reference data from a different tool or measurements. Instead, the mean value and the standard deviation of the time series are compared to those of reference data.

For the test case TestRegularWave (see Listing 5.4), the test component EquidistantTSValidator has been implemented. It can be parameterized with a tolerance that represents the acceptable absolute
difference between the two compared time series. The input files for the component must have the same length as well as equidistant values. If the input does not match the reference data, such that the absolute difference between input and reference data exceeds the value of tolerance, an error will be displayed. The resulting view for our example is shown in Figure 5.4: the simulation time plus the time used for validating the values is displayed as a sum in the test case tree view. When the user selects the erroneous test, basic information about the location of the first assert failure is provided together with a plot of the two compared curves.

For the test case TestIrregularWave (see Listing 5.5) a test component named BasicStatistics has been implemented to compare mean value and standard deviation as described above. The user can provide the parameters startindex and endindex to define the range of values that shall be checked for the provided input values.

Additionally, the user can define the allowed variance of the calculated mean values and standard deviations. Figure 5.5 shows the resulting view displaying information about the assertions that were violated. While the mean value of the wave elevation matches the reference data, the standard deviation is not acceptable. As for the test case TestRegularWave, a plot is provided to visualize the compared values to the user.

5.4 Conclusion and Future Work

It has been demonstrated that the test framework MoUnit can effectively be used for testing Modelica models by unit testing. It was shown that the framework is flexible as it supports multiple simulation tools as well as input files from other tools for result comparison.
Figure 5.4: Result for test case TestRegularWave

Figure 5.5: Result for test case TestIrregularWave
This allows to check the different behavior of models when using different simulation tools as well as different solvers. It was also illustrated that it is possible to extend MoUnit with custom test components that provide functionality different from regression testing. Using the example of simulating water waves, simulation results of Modelica models can be compared to simulation results of the reference tool ASAS WAVE. The extensibility also allows to add test components that use results from analytical calculations for result validation.

For error reporting, a flexible interface has been defined. A plot has been implemented for wave model tests displaying the time series of the data that is being compared together with the corresponding mean values and standard deviations. Thereby, the user can directly investigate the cause for a problem visually. Furthermore, an ANT task has been implemented that executes test suites and registers a listener that serializes a test report in an XML file compatible to JUnit. Hence, the tests can be integrated into a build system for continuous testing.

In the future it is planned to create additional test functionality that allows to test single values of result files. The implementation of test components for analytical solutions are planned for the models of wind turbine components — as far as they are available. For fast testing of algorithms, it is furthermore desirable and planned to interpret the algorithmic code, such as Modelica functions, to allow direct calls and comparison to expected result values.
Chapter 6

Abstracting Physical Models for the Use with Arbitrary Simulation Environments

Models for offshore wind farms are being developed at Fraunhofer IWES under the project name OneWind. The goal is to provide a tool that allows wind turbine designers and manufacturers to rapidly develop models of wind turbines at different levels of detail. It shall also be possible to use different types of models and to transform them into each other in order to check the models against the users expectations with the best suitable simulation technique. Furthermore, simulations of different load cases according to the respective wind turbine standards and guidelines [Energie, 2005; International Electrotechnical Commission, 2009] will be possible. A key purpose of the OneWind project is to implement the load calculation as a coupled aero-servo-hydro-elastic simulation in Modelica, to get a better estimation of the turbine performance, to analyze the system response and to optimize the component and control system design.

6.1 Parameterizeable Models and Modelica Code Generation

Nowadays many tools are involved in the process of wind turbine design like GH Bladed\textsuperscript{14} for load calculations or Focus\textsuperscript{15} for rotorblade designs, just to name two of them. Additionally, Computational Fluid Dynamics

\textsuperscript{14}http://www.gl-garradhassan.com/en/GHBladed.php
\textsuperscript{15}http://www.wmc.eu/focus6.php
(CFD) tools give a more precise view on aero-dynamical influences from 3-D flow effects on rotating blades. All of these tools define their own data and model representations and hence provide only limited interoperability. The OneWind project aims at providing consistency in the highly iterative design process between different models for various purposes at design time. Therefore it facilitates the usability of the tools in one integrated development process by introducing a purely parametric data layer called EDD [Strach et al., 2012]. Hence data from external tools must be imported into the parametric representation and reverse transformations to the tools data model must be performed in order to generate compatible data as input for simulations with the external tools.

Due to the different domains that the simulation environments are aiming at, there cannot be just one model in the parametric level that represents all kinds of physical properties of a wind turbine component. As an example, structural models of rotor blades must be fairly simple with only few degrees of freedom (e.g. a modal description) in order to be able to execute load calculation for thousands of loadcases in a reasonable time. In contrast, the detailed design of the composite structure of a rotor blade needs a model with fine grained information about the layer structure to be used for FEM simulations. During the design process changes in the fine-grained models need to be transferred in each iteration step to the simpler models of the load calculation (see Figure 6.1). These transformations from fine to coarse-grained models can often be done automatically. The opposite transformation direction is called design transformation and needs additional user inputs and engineering know how in order to be performed. The transformation is explained for rotor blades in Section 6.1.2. A detailed laminate model is transformed into a low-detailed beam model. The transformation from the EDD representation of wind turbine models to computable Mod-
elica models, as another feature of the OneWind development environment is described in Section 6.1.2.

In the domain of wind turbine modeling with Modelica the EDD can be used to reduce the complexity of model parameterization for the user. Instead of editing the potentially complex source code directly, the user only sees the model parameters that are crucial for the model’s behavior. Thus, the user does not need to understand the syntax of Modelica. Instead of transforming the complete model to a Modelica representation, only the user-defined parameters are transformed to Modelica records, that belong to model components of the OneWind Modelica library [Strobel et al., 2011]. The library consists of major components at different complexity levels needed for load calculations of typical offshore wind turbines. Model components for the structure and aerodynamics of rotor blades are provided as well as a hub, nacelle with drivetrain and generator, tower, substructure and operating control procedures. Additionally, the library includes models for the simulation of external conditions, (wind, soil and waves) and their influence on the wind turbine’s structure (for more details see Section 2.1.3).

The library is constructed in a way, that the assembly of model classes with related parameter classes can be manipulated by redeclaration statements and inheritance from the base library classes. An example is shown in Section 6.2. This has the benefit that developers of Modelica models can reuse the model components of the library, individually change parameters of the model classes and easily enhance it. Furthermore a wind turbine model with a desired complexity level can be constructed using a custom combination of library models.

6.1.1 Engineer Design Data

The concept of a purely parametric data layer is used in all products of the OneWind project. It represents the idea to ensure the consistency of models in different levels of detail for all purposes needed during the design process of a wind turbine. In this layer the user can manipulate models which are imported or newly created in a unified way, regardless of the software used for further processing or simulation. The models can then be transformed to a computable form and simulated by external tools. When the simulation results are obtained, the user can analyze and assess the results and start with a new design iteration in order to enhance the physical models.

Hence the parametric layer used in OneWind is build by various meta-model definitions defined with EMF. EDD models specified in the meta-models implement a common interface. One strength of the meta-model approach is that one can define automatic transformations between two models of the same level, if one can define relations between features on the meta level.
6.1.2 Model Transformation

Since model driven software development is increasingly accepted and used by software engineers, transformations of the developed models are becoming important. Various techniques for the transformations have been developed, of which some are described in this section. The transformations to Modelica models which are used in our project are presented in the subsequent section.

About Model Transformations

Model transformations can be done in different ways. The most appropriate one is the direct transformation between models based on rules defined for elements of two meta-models. These rules can automatically be applied to convert one model to another. If the transformation rules are bijective, which means in both directions, automatic synchronization between two models can be realized. This kind of transformation is called a Triple Graph Grammar (TGG) [Königs, 2005].

TGGs can only be defined for a small set of models. The first requirement is that the two models to be transformed into each other must be semantically similar. For example, models of towers cannot be transformed to rotor blade models. Secondly the information content must be comparable. Modal blade models may not be translated into more detailed models that can be used for FEM simulations. The information needed for the physical properties in a FEM model cannot be automatically derived from the kind of parameters available in a modal blade model. This observation does not only hold in the context of TGGs. Generally speaking, a transformation from one model to another can only be done if the “structural information” content of the initial model is greater or equal to the “structural information” of the target model.

In general models of physical components at different levels of detail for the use with different theories do not meet the above mentioned requirement for TGGs of being bijective. Often physical theories are needed to transform detailed models into coarser models and the opposite design transformation is always based on assumptions and engineering know how, which is to be obtained from the user in terms of parameters of the transformation. Transformation mechanisms are needed to implement the complex algorithms in order to perform the transformations. Nevertheless the transformation of a detailed model into a coarse-grained model is highly automatable and can be reused when parameters change in the detailed model during the design process. For the transformation of EMF based models several tools are available, like QVT or ATL\textsuperscript{16}. These languages provide functional language style syntax for the definition of automatic model transformation rules. In the future these languages might be used in the OneWind project where applicable. However, at the moment only Java-based transformation modules are being developed.

\textsuperscript{16}http://www.eclipse.org/m2m/
Chapter 6. EDD
6.1. Parameterizeable Models

Figure 6.2: Structure of laminate blade models

Figure 6.3: A simple beam model

Transforming a laminate model into a beam model

During the OneWind project, the transformation of rotor blade models has been implemented at Fraunhofer IWES. More precisely, the transformation from laminate models to beam models was considered. This section presents the results from [Strobel et al., 2012]. Laminate models are the most detailed models of rotor blades that have been implemented for the OneWind project. They allow to define the inner and outer geometry and the used materials (see Figure 6.2) with high detail. Laminate models are used for the detailed rotor blade design, i.e. when the layup of the rotor blade laminates is investigated.

The beam model on the other hand is a simple model that is build up by beam elements (finite elements). While they are not as accurate as laminate models in terms of material and geometric properties, they are of vital importance for loads analysis. Using a laminate model in a loads analysis would increase the simulation time too much, which is why a beam model is used. Figure 6.3 displays an example of a beam model. For the transformation from laminate to beam models, the classical laminate theory (CLT) is used. The following steps are performed:
Chapter 6. EDD

6.1. Parameterizeable Models

Figure 6.4: An idealized blade cross section

- Idealization of the laminate layers as discs and calculation of their stiffness matrices,

- Transformation of the stiffness matrices of each layer from the layer’s coordinate system, considering the fibre angles, into the laminate’s coordinate system,

- Combination of the transformed stiffness matrices for each layer into one single laminate stiffness,

- Calculation of the laminate constants and the laminate thickness.

Performing the steps above leads to a model with homogeneous laminate properties. The material of a blade profile only varies at the external circumference at this point and not in thickness direction.

The geometry needs to be calculated as accurate as possible in order to retrieve the properties of the cross sections. Rectangles are calculated by the interpolation of immediate neighboring polygons. The width of the rectangles represents the thickness of the laminates. A simplified resulting cross section width 10 rectangles is displayed in Figure 6.4. It was found that 500 rectangles are usually needed and sufficient for detailed calculation.

The material area can be calculated as the sum of all rectangles (problems caused by overlapping rectangles as occurring at the trailing edge of the blade are not discussed here). The average material density is calculated as the sum of the densities of the rectangles weighted by the rectangle’s size. The same holds for the coordinates of the area and mass centers. Based on the properties of the rectangles the following properties for cross sections can be calculated:

- Overall stiffness,
- Bending stiffness,
- Torsional stiffness,
- Coordinates of ideal center of gravity.

An arbitrary number of nodes can be chosen for the beam model which is not dependent on the number of cross sections of the laminate model.
Figure 6.5: Calculation of the beam model

Figure 6.5 displays the calculation of the beam properties based on the cross section model. Black lines denote blade sections and red lines denote the centers of the beam elements. The material property of each blade section is constant. For the calculation of the properties of the green hatched beam element the overlapped blade section is completely used for the beam properties. The other 2 sections are partially considered by interpolation.

Another example for a transformation was shown in [Strach et al., 2012]: the transformation from high-level offshore substructure models for detailed design to low-level models, that can be used in loads analysis.

**Transforming EDD to Modelica Code**

As a result of defining the Modelica language as a Xtext grammar, serialization/unparsing of Modelica code from an AST representation to textual Modelica source code is automatically available. Transforming EMF based wind turbine models to Modelica AST representation may be possible by using transformation languages as mentioned above. However, since the abstract syntax of Modelica is rather complex, our initial approach is to write such transformers in Java.

The generated Modelica source files are used along with the OneWind library (see Figure 6.7) for the highly coupled aero-servo-hydro-elastic simulation of wind turbines. The EDD model for a rotor (see Figure 6.6) is explained and serves as an example of the generation of Modelica code. A rotor consists of a hub and multiple rotor blades. Usually three rotor blades are used in modern horizontal axis wind turbines. The rotor blades consist of blade elements that define structural properties like masses, stiffnesses or lengths. Additionally, each blade element defines an airfoil that describes the aerodynamic properties of that part of the blade. The user can edit the properties mentioned above, in order to design a rotor. The generator
Figure 6.6: EDD-model of a rotor, rotor blade, blade element and airfoil
then generates Modelica records containing the user defined parameters. A Modelica rotor stub that is defined in the OneWind library is parameterized by the generated data. Finally the model consisting of the library components and the generated part can be simulated using Modelica simulation environments. The parameters which are customizable by the user are separated in Modelica records. Hence for each model that is being transformed, for example a blade, a data record is created that contains the parameters. In the blade example the data record contains single parameters for unary properties and arrays for multiple properties like blade elements. The array size is equal to the number of elements in the list. Moreover, the user can choose between different kinds of model components to change the structural properties of the model. One can, for example, decide whether a rigid, modal or FEM blade model shall be used for the simulation. When the user selects a blade model that differs from the default one used in the library, the blade model is changed by the generation of a redeclaration. Thereby it is possible to customize the wind turbine model. The approach described above allows us to provide the wind turbine designer with an abstract view of the main properties of a wind turbine model. Variants of wind turbine models can be created quickly and compared to each other.
record BladeData

//array length 3
Profile_Cylinder1 cylinder1;

//array length 142
Profile_DU21 du21;

//array length 127
Profile_NACA64 naca64;

parameter Integer nBladeElements = 3;

replaceable parameter Profile
profile[5] =
{cylinder1,du40,du35,du25,naca64};

end BladeData;

Listing 6.1: Generated blade data record

In the next section some problems that occurred during the implementa-
tion of the transformation modules including the use of polymorphic data
types are described.

6.2  Polymorphism in Modelica

The transformation strategy described in the previous section provides a
generic way of converting models from a purely parametric representation
to simulatable Modelica models. For this approach to work the two types of
models must be structurally equivalent. Hence the library must be designed
in a way that provides suitable class stubs for the generation of Modelica
code and the parametric model must meet the structure of the model com-
ponents defined by the library. During the development, we recognized that
the conditions can be met with reasonable effort. Nevertheless problems
arise in cases where lists of items of complex types are transformed to Mod-
elica code. One example is rotor blades containing blade elements that have
a length and an airfoil property. In the example, the *NREL 5 MW reference
baseline wind turbine model* [Jonkman et al., 2009] is used. Listing 6.1
displays the resulting data record of the transformation.

A load element contains a parameter profile that holds a list of aerody-
namic profiles (Listing 6.2) of type Profile. The class Profile displayed
in Listing 6.3 is a kind of template record. It contains arrays of profile spe-
cific data: the angle of attack $\alpha$[deg], lift coefficient $c_a$($\alpha$)[-],
drag coefficient $c_w$($\alpha$)[-] and $c_m$($\alpha$)[−] which is the pitching
moment coefficient. The array size is variable since the number of properties
varies between different profile types.

Concrete profile records like Profile_NACA64 (see Listing 6.4) define
the profile specific value quantity and assign the concrete values to the array.
This structure provides a similar behavior as generic array lists in Java.
model LoadElement
  parameter Profile profile;
end LoadElement;

Listing 6.2: Load element containing the airfoil descriptions

record Profile
  import Modelica.SIunits.Conversions.
  NonSIunits.Angle_deg;
  parameter Angle_deg alpha[:];
  parameter Real ca0fAlpha[:];
  parameter Real cw0fAlpha[:];
  parameter Real cm0fAlpha[:];
end Profile;

Listing 6.3: Generic type profile

whereby the generic type in this case is defined by the Modelica record Profile.

For simplification reasons the listing shows only a reduced set of aerodynamical coefficients of the NACA64 airfoil. Typically, it consists of many support positions for the complete range (-180 to +180 deg) of the attack angle alpha with variable equidistant steps. For frequently used regions of attack angles usually small step sizes are used. Thereby a better linearization between these points and approximation of the measured coefficients during a simulation can be achieved for the used airfoil. The profile data can be used in a unified way as defined by the Profile record and therefore the class using the profile data does not need to know the concrete profile type.

Finally a blade model is generated that assigns the aerodynamic profiles to the load element of the blade (Listing 6.5).

A problem that occurs using polymorphic arrays as explained above is that the created list of instances of class Profile consists of types with different array size, like the size of all arrays from profile[1] = 3 and from profile[2] = 142. Unifying the records leads to ragged arrays that are not defined by the MLSpec. Hence the behavior during simulation is unpredictable or the simulation tool does not even compile the code.

Listing 6.6 shows a workaround for this issue. Instead of creating objects from a list of types with variable array size, the Profile class for each load element object is directly declared with a modification statement of the desired profile class. The profile data of each blade element object is filled by array concatenation which corresponds to a normal parameter modification statement. This circumvents getting objects with variable sized array types.

In this case a list of instances of the class LoadElement is defined, where each load element object has a different airfoil type and the size is specified
//For clarity reduced profile set
//of NACA64 (15 instead of 127 values)
record Profile_NACA64
  extends Profile
  {
    alpha = {-180.00,-90.00,-30.00,
    -10.00,-5.00,-3.00,-1.00,
    0,1.00,3.00,5.00,10.00,
    30.00,90.00,180.00},
    caOfAlpha = {0,-0.067,-0.829,
    -0.711,-0.151,0.088,
    0.328,0.442,0.556,
    0.784,1.011,1.382,
    0.926,0.053,0},
    cwOfAlpha = {0.0198,1.3587,0.4295,
    0.0111,0.0079,0.0064,
    0.0052,0.0052,0.0052,
    0.0053,0.0058,0.015,
    0.4294,1.4565,0.0198},
    cmOfAlpha = {0.00,0.3636,0.1563,
    -0.0734,-0.0841,-0.0912,
    -0.0971,-0.1014,-0.1076,
    -0.1157,-0.124,-0.1149,
    -0.1668,-0.3858,0.00};
  };
end Profile_NACA64;

Listing 6.4: Concrete profile record Profile_NACA64
by the respective modification. Thus the array sizes are known at this point as the profile data is not assigned in a generic way and the compiler does not fail to unify the record types.

The drawback of this approach is that the code generation is not realized as described in section 6.1.2 and therefore it may not be obvious where the generated data comes from. Additionally, it prevents one from creating an automatic transformation mechanism as custom adaptions to the code generators must be implemented. The advantage of an automatic transformation algorithm is that it reduces the implementation effort, creates code that is easier to test and it enhances the readability of the generated code.

The model from Listing 6.6 can now replace the default blade model from the OneWind library by using Modelica replaceable object types. This also holds true for the class LoadElement. The physical algorithms (e.g. calculating loads for the blade from wind inflow) are reused, only the calculation parameters are modified. This approach is used for all main model components of the library (rotor, nacelle, tower, substructure, operating control, environment etc.) in order to create a custom model of a wind turbine.

The OneWind library contains a default wind turbine model HorizontalAxis.OffshoreWindTurbine. All main components in this model are replaceable objects and can thereby be redeclared by parameterized classes of the concrete NREL5M model. Listing 6.7 shows the main class of the generated concrete wind turbine model of the NREL5M reference baseline offshore wind turbine, which inherits from the default model. It can be simulated with a Modelica compiler in combination with the generated model classes and the OneWind library components.

### 6.3 Conclusion and Future Work

Based on the experience gained during the development of the simulation environment, it can be seen that it is possible to create a common data basis for different tools dealing with the design and simulation of wind turbines. Transformations between different kinds of models enable the reuse
1 model NREL5MBlade
2   extends RigidBlade(
3     redeclare LoadElement loadElement(
4       // old:
5       // profile = bladeData.profile
6       // new:
7       profile = {
8         bladeData.cylinder1,
9         bladeData.cylinder1,
10        bladeData.cylinder2,
11        bladeData.du40,
12        bladeData.du35,
13       // ...
14        bladeData.naca64,
15        bladeData.naca64}
16     )
17   );
18 end NREL5MBlade;

Listing 6.6: Redeclaration of blade element data

1 model NREL5MOffshore
2   extends HorizontalAxis.OffshoreWindTurbine
3   (  
4     redeclare NREL5MOperatingControl
5     operatingControl,
6     redeclare NREL5MRotor rotor,
7     redeclare NREL5MNacelle nacelle,
8     redeclare NREL5MTower tower,
9     redeclare NREL5MSubstructure
10     substructure,
11     redeclare NREL5MSoil soil,
12     redeclare NREL5MWind wind,
13     redeclare NREL5MWater water
14   )
15 end NREL5MOffshore;

Listing 6.7: Main wind turbine class with redeclared components

106
for different purposes.

As described for the generation of Modelica models, the use of the EDD approach allows one to parameterize models and to create simulatable representations like Modelica source code. Furthermore, models can be structurally customized to create several versions of physical models in the end. In the future, automatic transformation with languages as described in section 6.1.2 may be introduced for the transformation between EDD models as well as between EDD and simulator specific models.

For the Modelica code generation it is desirable to use transformation languages, since changes in the MLSpec are easier reflected by adapting a few transformation rules than by modifying Java classes.

Increased polymorphism, as discussed in section 6.2, would enhance the generation as the generated code would be easier to understand and the generation could be encapsulated for each model component. This simplifies the code generation and reduces the dependencies between model components and their contained declarations.

In order to enable the use of more polymorphism, the MLSpec would have to be enhanced, in this case allow polymorphic ragged arrays (see Fritzson et al. [2011] for the implementation of polymorphic ragged arrays in MetaModelica). As most of the Modelica simulators compile Modelica to plain C code, the polymorphism would have to be adapted as C does not support polymorphism. However, this could enhance the parameterization of Modelica code as the implementation would be independent from the concrete model components that are used.

To enhance the transformation process the use of transformation languages will be investigated as the next step. This will provide a more generic way of code generation and enhance the maintainability since changes in the meta-model of Modelica can be applied easier.

Furthermore, investigation is needed whether transformation rules can be derived that allow transformation of arbitrary types of models. Hence, it would not be necessary to create transformation rules for each particular EDD model, but universally applicable rules would further simplify the transformations. To realize this goal more generic data structures as described in section 6.2 would be desirable.
Chapter 7

Developing a Modelica IDE using Model-Driven Software Development

This chapter describes the development of the Modelica IDE OneModelica. OneModelica builds upon the Eclipse Framework [Vogel, 2013; Clayberg & Rubel, 2008; Gamma & Beck, 2003] which is freely available and highly extensible through its plug-in mechanism. The basic Eclipse Framework provides the OSGi-Framework implementation Equinox [McAffer et al., 2010]. Equinox allows the extension of the framework by means of a plug-in mechanism. Examples of plug-in contributions are the Java development tools (JDT)\(^\text{17}\) and the C/C++ Development Tooling (CDT)\(^\text{18}\) which are used for the development of Java and C code. Various configurations of Eclipse can be downloaded from the project’s web site. Each of the product is set up to support the user in the development with a particular toolset. Since OneModelica is developed using model-driven software development, the Eclipse Modeling Tools\(^\text{19}\) version of Eclipse is used as the base framework configuration and extended by plug-ins that support the development of Modelica models. The models that are created for code generation are defined with the EMF which is the central component of the Eclipse Modeling Tools. Figure 7.1 displays the overall structure of OneModelica. Plug-ins can define new extension points using a XML extension point scheme. OneModelica provides extension points for custom model validators or model linking (name lookup) implementations.

UI-dependent and UI-independent code is usually divided into two distinct plug-ins. The separation allows to run and test programs on sys-

\(^{17}\)http://www.eclipse.org/jdt/
\(^{18}\)http://www.eclipse.org/ctd/
\(^{19}\)http://www.eclipse.org/modeling/
tems that do not provide a graphical user interface. This is necessary if code is used on servers that does not offer any user interface, for example for performing automatic tests on build servers. The plug-ins developed for OneModelica adhere to this convention which made it possible to create a continuous integration process including automatic tests with Hudson [Burns & Prakash, 2013].

This section introduces the plug-ins developed for basic features of OneModelica, which implement support for Modelica model development. The features include a custom editor and views that display detailed information about the Modelica models.

7.1 The Modelica Editor

The Xtext framework allows the generation of code for DSLs, including editors and basic view implementations, based on grammar definitions. Hence, the main component of OneModelica is the Xtext grammar for the language Modelica. The Java code that is generated from the grammar file includes a tokenizer and parser that are used to read Modelica documents. The parser is also used by the generated Modelica editor (Figure 7.2) to check for syntax precision and syntax highlighting of opened documents. Engineers are assisted in creating models with Modelica by the editor. The supporting functions assisting the Modelica developer in creating models are described in the following sections.
Chapter 7. OneModelica

7.1. The Modelica Editor

7.1.1 Syntax Highlighting

Syntax highlighting is a feature that defines how textual documents are presented to the user in text editors. DSLs are structured by keywords, terminals, and non-terminals. Examples of keywords in Modelica are `model` and `package` which denote the beginning of a model or package declaration. The keywords are defined in the MLSpec [Modelica Association, 2012] and are reserved, so they cannot be used for other purposes such as component names. Based on the grammar definition, Modelica documents can be analyzed and represented (parsed) as syntax trees.

The Modelica syntax is highlighted in the Modelica editor in order to make it easier to develop and understand Modelica code. Keywords are automatically highlighted by the Xtext framework with a magenta colored and bold font. Xtext provides a predefined terminal definition for Strings, which are then automatically colored blue. For Modelica the definition of Strings has been altered since text sequences in single quotes (’) which are often used as Strings in DSLs, represent identifiers in Modelica. Figure 7.3 displays the Modelica editor with the highlighted keywords `within`, `model`, `extends`, `equation`, `annotation` and `end` and two blue colored Strings. The way how keywords and terminals are displayed in the editor can be configured, allowing the user to define custom visualization profiles.

7.1.2 Syntax Validation

Based on the grammar definition, the syntax of Modelica documents can be checked automatically. The generated parser analyzes documents based
on the rules defined in the grammar. The parser tries to create an AST representation of Modelica documents. An error message will be displayed if the input does not match expected rules. If the document does not conform to the grammar definition, for example if the user accidentally wrote equations instead of the keyword equation which denotes the beginning of an equation section, an error message is displayed. With the information provided by the CST, the parser can inform about the location of the error and the input that is expected at this location (see Figure 7.4). This can help the user to fix the incorrect part of his or her document.

By checking the syntax it can be ensured that the models are at least syntactically correct when they are used for simulation with a Modelica simulation tool. This accelerates the development process compared to using general text editors that do not provide any information about the syntax correctness of Modelica documents.

7.1.3 Number of Variables and Equations

For correct Modelica models it is important to define an equal number of unknown variables and equations. Otherwise the resulting equation system is not balanced and cannot be solved. The Modelica editor in OneModelica has been extended to provide this information to the user. The number of variables and equations of the currently edited document is displayed in the information panel of the editor. Furthermore, the numbers are also provided for the flattened model, which is the resulting model after all variables and equations of extended classes are considered. Hence, the user can immediately recognize if the developed model is balanced.

7.1.4 Indexing and Linking Modelica Models

Since Modelica is an object oriented language, classes can be created that implement certain functionality like the behavior of a tower or a rotor blade in a wind turbine. Instances of classes are created as components of a model
and can be parameterized (by setting parameters or by re-declaration of inner components).

The hierarchical structure elements for name spaces in Modelica are (potentially nested) packages. Classes and components are defined inside packages and can be identified by their qualified names inside the name spaces. Classes can also be defined directly in the base package, which does not define a name. This is the case for Real or Integer declarations. Within statements are used to define the name space of class and component declarations inside a Modelica document. Hence, the within statement declares in which package the contained declarations are defined. According to [Modelica Association, 2012], the package structure should also be represented by folders in the file system.

In order to resolve references from one document to the corresponding declaration in the same or another document, name space information must be extracted. Therefore, the Modelica files need to be parsed and indexed. The references are then resolved using the namespace information in the index data. Listing 7.1 displays a simple Modelica document.

The within statement declares that all class declarations inside the file are defined inside the namespace WindTurbine. The first class declaration is a package with the name Components that contains the models BladeElement and RotorBlade. The blade element declares an instance variable called length with the type Modelica.SIunits.Length. The fully qualified name is used to reference the type Length which is defined within the MSL. The type reference of the bladeElements components inside the model RotorBlade does not define a fully qualified name. In the example this means that the class declaration of RotorBlade must be defined in
the same name space (\texttt{WindTurbine.Components}) in order to be visible in the context of the referencing model.

More rules exist for resolving references. Components defined inside extended classes, are visible without the need of using qualified names. Hence, the content of the extended class must be known during linking. Moreover, single components and classes as well as sub-elements can be made visible without the need of using fully qualified names by using \texttt{import} statements.

\section*{Indexing Modelica Documents}

For fast reference resolving, an efficient algorithm is needed. The default algorithm provided by Xtext is inefficient for Modelica models since it creates scopes for every named element that provides information about any objects that are visible for the element. For Modelica this approach would lead to large sets of scopes. Additionally, the use of \texttt{redeclare} and \texttt{replaceable} is problematic since visible components can change in different model scopes. Instead of calculating all visible elements beforehand and storing them in scopes, the linking mechanism has been altered. References can be searched based on the algorithm defined in [Modelica Association, 2012, Chapter 5] and [Fritzson, 2014, Chapter 10.3.2.2] allowing to end the search early when the correct element has been found.

The procedure of indexing all documents in the workspace is the following:

- all Modelica documents inside the workspace are parsed
- an index is created for each project in the workspace
- the parsed documents are analyzed and all package, class, and component declarations are recorded in the index files.

If a document is being edited later on, the existing information about the file is deleted from the index. Afterwards the document is analyzed again and the index is updated according to the content of the document.
Linking Modelica Documents

After indexing, linking of models in the workspace is automatically triggered by the Xtext framework. As an example, the search order for a component reference in an equation is the following:

- search for the component declaration inside the current class
- search for the component declaration inside extended classes
- search for an imported constant component declaration packages

The index is especially helpful when imported elements are investigated. The Modelica files which define elements in the imported name space are not opened and parsed, which can be time consuming, unless the index contains information that the searched element is defined inside the current file. Projects can references other projects and libraries. For the linking algorithm this implies that the index files of the referenced resources are also considered when searching for declarations.

Index Data Structure

For OneModelica a custom data structure has been implemented with EMF which better reflects the special needs for linking (Figure 7.5). The class AbstractModelResources is the base class containing information about known Modelica files and other referenced resources. Each Modelica project has such an object that stores information about source folders inside the project, references to other Modelica projects and Modelica libraries (which contain a special implementation of AbstractModelResources).

Each AbstractModelResources can have two types of indices: A linking index which is used to resolve references in Modelica documents, and a package index which provides information about the package structure of the documents known by the AbstractModelResources (e.g. of a project). The indices use nested HashMap objects which allow for fast access. The main difference between both kinds of indices is that the linking index uses the name of classes and components as the first map’s key while the package index represent the hierarchical structure of the name spaces and hence stores simple package names as keys in the first map. For the example code in Listing 7.1 the key sequence of the BladeElement model inside the linking index is:

BladeElement ↦ WindTurbine ↦ Components

While the same class is represented as follows in the package index:

WindTurbine ↦ Components ↦ BladeElement
The package index is not used during the linking process but for other purposes such as displaying the content of a Modelica library or searching for model declarations in the entire workspace (Section 7.2).

Each index has two maps. The `nameToInfoMap` contains the information about classes and components as described above. The `fileToInfoMap` stores information about the content of each file. If a file is being altered (edited, removed or overwritten) the qualified names of the file are removed from the indices. This is necessary since those changes need to be reflected in the index files.

For the mapping of file names to qualified names a single `HashMap` instance is used. Note that due to the use of the key attribute inside `FileToInfoMapEntry` and by setting the instance class name to the type `java.util.Map$Entry` and the value reference to `FileEntryList` a `HashMap` is automatically generated by EMF [Steinberg et al., 2009, p.269]. The information in `nameToInfoMap` is stored inside instances of the type `IndexMapEntry` which is again modeled as a map. The first entries contain the class and component names for the linking index and the base packages for the package index (e.g. Modelica for the Modelica standard library). Information about sub-elements is stored in inner maps of the `ClassIndexElement`. Each `ClassIndexElement` has a `ClassInfo` that stores the URI to the containing file. This allows the defining document to be loaded and the examined classes and components to be found inside the file.

### 7.1.5 Code Completion

A benefit of specialized editors in contrast to simple text editors is that the syntax of the edited code is known. With the grammar file for Modelica at
hand, code completion for keywords can automatically be derived. The generated Modelica editor automatically provides code completion for keywords. If the user triggers the code completion at a certain position in Modelica code, the corresponding AST node for the position is obtained. The node represents a data rule of the grammar and hence the permitted keyword for that rule can be suggested. If the user wants to complete a keyword and already typed in some letters, the proposals are filtered accordingly.

Class and component references can be completed in much the same way (see Figure 7.6). Xtext automatically uses the scopes generated during linking for attaining valid declarations (see Section 7.1.4). Since the default scoping mechanism turned out to be inefficient for Modelica, the completion of code has been adjusted to match these special needs.

The completion of code for references is similar to resolving references in that the scopes of visible objects are searched at the current text position. Different search algorithms are used depending on the current position and entered text:

**Import statements** Proposals for import statements can be straightforward to find. The indices which are accessible from the current project are used to find suitable package and class names for the text entered. If the user asks for proposals directly after the import keyword all base packages (e.g. Modelica from the MSL or OneWind from the OneWind library) will be proposed. If the text entered contains a period, the string preceding the last period is analyzed to find the imported package. The succeeding string is used to filter the proposed sub-packages, classes and constant parameters. For the text `Modelica.M` the package proposals `Modelica.Magnetic`, `Modelica.Math`, `Modelica.Mechanics` and `Modelica.Media` are provided for the MSL.

**Extends clauses** For extends clauses the search scope depends on the package in which the current class is defined and the classes that are
visible due to import clauses. Hence, all visible classes are collected and then filtered by the text typed in by the user. For fully qualified extends clauses the proposal computation is identical to what is performed for import clauses except for the proposal of constant parameters.

Components and Variables The declarations inside the class and visible elements (public) of extended classes are collected and filtered when components and variables are searched, as during the definition of equations and algorithms. If the entered text contains a period, the sub-components of the component preceding the period are collected. Once again the visibility of the component declaration needs to be taken into account in order to provide only valid proposals.

7.1.6 Code Folding

Code folding allows to hide parts of source code inside the editor. Thus, the user can concentrate on the important parts during development. Code folding is automatically provided by the Xtext editor and needs to be configured in order to fold only suitable parts of the language. In OneModelica the following sections of models can be folded:

- Annotation
- Class
- Connect clause
- Equation section
- Algorithm section
- Component declaration list

Annotations are automatically folded in the Modelica editor. The generated editor has been modified and all annotations are collapsed when the editor is opened. This is useful since annotations usually contain a lot of graphical information and HTML-based documentation that is hard to understand for the user as plain text. Hence, hiding this information can help to keep track of the important parts of code containing the structure and behavior of the models. Additional information can be displayed by specialized views like a connection editor or a web browser. Figure 7.7 shows an example of code folding. Regions that can be folded but are currently expanded are marked with a “minus” icon as can be seen for the model CodeFolding. The annotation inside the model is already folded and can be expanded by selecting the “plus” icon. When moving the mouse above the icon, the folded text is displayed in a pop-up window. This functionality is provided by the Eclipse framework and has been configured in OneModelica to suit the Modelica language.
Refactoring [Fowler, 1999] is an important task when developing large scale software and models. Existing code needs to be adjusted over time since requirements change during development. Basic name refactoring for classes and component names is automatically available through the Xtext framework. When a name is changed, all references to the declaration must be changed accordingly in order to maintain valid code. The name refactoring can be achieved because the linking mechanism described in Section 7.1.4 keeps track of all name declarations and references to the named elements.

Additionally, Modelica specific refactorings have been implemented for OneModelica. Figure 7.8 displays the model ModalTower. The within clause in the first line indicates that the model should be defined inside the sub-package Tower. However, the resource is located inside the folders OneWind and Components. The MLSpec [Modelica Association, 2012] defines that the package structure of models should be represented by the folder structure in the file system. Hence, a sub-folder Tower is missing for the example model. This problem is indicated by a warning in OneModelica. When the user clicks on the warning, two refactorings are provided:

- The within statement can be adjusted to match the folder structure. In the current example this would result in the package declaration OneWind.Components
- The document can be moved to a folder that represents the package declaration. In this example a sub-folder with the name Tower would be created inside the folder Components and the document would be moved into the new folder.
Another refactoring that has been implemented is the renaming and moving of packages. If the user renames a package in the Project Explorer, the within statements of all Modelica documents contained within (also in sub-folders) is automatically adjusted. Afterwards the folder is renamed and moved to the appropriate location in the file system. If the package is moved into another package that does not yet exist, the missing folder structure and package.mo files will be created.

7.2 Modelica Views

The main elements in the Eclipse framework are editors. However, it is important to provide additional information about the edited documents to the user. Since the content of documents can be complex it can also be necessary to only provide information about a certain aspect, like the structure of the document. This task is accomplished by views. In older Eclipse versions, one editor area was surrounded by a number of views. This strict layout scheme was omitted with Eclipse 4. Now it is possible to freely move editors and views or to embed editors inside views in order to manipulate data. This section presents the views that were implemented for OneModelica.

7.2.1 Modelica Perspective

A perspective in Eclipse is a predefined layout of editors and views for a certain development purpose. OneModelica provides a perspective for the development of Modelica models. The perspective opens and arranges the Modelica editor and the most important views (Figure 7.9) by default. The main part is the Modelica editor that is arranged in the center of the IDE. On the left-hand side the project explorer views the content of the current workspace. The workspace is a user defined folder in the file system.
Projects can be created inside the workspace. The projects can contain files and folders. A “problems” view, located below the editor, displays errors, warnings and information markers that are attached to resources in the workspace. Syntax errors, unresolved references and semantic errors are visible in this view for Modelica models and the user can jump to the error location by double-clicking the error messages. A Modelica documentation view is located below the problems view. It displays the HTML-based documentation that is defined inside class annotations. The outline view is located on the right-hand side of the editor. It displays the structure of the currently opened Modelica document. This makes navigation in large Modelica documents easier. Outside of the “outline” view a SVN repository view is opened since it is important for the development of Modelica models in teams to connect projects to a source code management system like SVN.

### 7.2.2 Project Explorer

The Project Explorer is a view that is provided by the Eclipse framework. The main purpose is file management allowing the user to view, create and manipulate files and folders in the workspace, which is the root element of the Project Explorer. All files and folders are kept inside projects which are sub-folders inside the workspace with a special configuration file named “.project” (it is also possible to link external resources to the workspace to make them available to the user). Configuration files starting with a period are hidden by default by the Project Explorer. All files, folders, and projects inside the workspace and the workspace itself are managed by the Eclipse resources plug-in. The plug-in provides methods for manipulation and information retrieval. The Project Explorer provides extension points
Figure 7.10: Project explorer with Modelica specific extensions

that allow to adjust the representation of resources in the workspace. For OneModelica, content providers were implemented that provide additional information for Modelica resources like the contained class definitions of Modelica documents.

Modelica Project Folders

Figure 7.10 demonstrates how the Project Explorer displays the projects defined inside a workspace to the user. All projects in the example workspace are configured with a “Model Nature” (see Section 7.4.1 for details about Eclipse natures and the implementation in OneModelica). This is indicated by the wind turbine overlay icon in the upper right corner of the project icon. When a model nature is set, the Modelica content provider adds the virtual entry Plug-in Dependencies as the first child of the project. The package and class hierarchy of referenced projects and li-
Modelica projects provide some folders by default that contain only data for special purposes. The experiments folder is a folder which is indicated by the green overlay icon. The folder is automatically created with a new Modelica project. Experiments define properties for simulation such as the selected simulation tool (e.g. OpenModelica or Dymola), simulation start and stop time or output formats. Simulation results are stored inside the experiments folder and can be browsed and viewed in an XY-plot view (for details see Section 7.4.3). Experiments simulate physical models which are defined inside the models folder. This special folder is also created automatically and marked with an overlay icon. Prior to the experiment definition, a physical model must be created with the physical models wizard (see Section 7.3.3) which can then be selected in the experiment wizard (Section 7.3.4). Physical models can be browsed to inspect the instance tree of the model (Figure 7.11 displays the instance tree of a model that extends the DoublePendulum model from the MSL).

The last folder in Figure 7.10 is the source folder that contains the Modelica resources. Modelica source folders are marked with an overlay icon as shown in the Figure. Additional source folders can be created by the user.

### 7.2.3 Outline View

The outline view, provided by the Eclipse framework, displays the structure of documents and also makes documents navigable. The view needs to be configured for each language. Xtext automatically provides a generic view configuration that uses the grammar definition of the language. The generic
configuration can be customized by dependency injection. By default all data rules in the grammar definition that do not call other data rules but keywords and terminals, are represented as leafs. Data rules that call other data rules are represented as nodes in the document’s tree structure inside the outline view. The nodes are collapsed by default and can be expanded by the user to investigate the document’s content.

For Modelica documents the default behavior provided by Xtext would lead to large trees and thus would not provide a plain overview of the document’s structure. Therefore the view has been adapted to show only the structure elements that are important during the development of Modelica models. Figure 7.12 displays the resulting Outline View for the model RobotR3 from the MSL. Following Modelica specific data rules are displayed as leafs in the outline view:

- Extends Clause
- Import Clause
- Component
- Equation
- Algorithm

Data rules that are displayed as expandable nodes in the tree are:

- Class, Model, Package, ...
- Equation Section
- Algorithm Section

Any other data rules in the grammar definitions are neglected as elements of the outline view. Outside of the content which is displayed in the outline view, the icons and labels that are used for the nodes were customized. Keywords are displayed with the same font as defined for the Modelica editor (e.g. extends and import keywords as displayed in Figure 7.12). Icons for classes and class types of components are taken from the annotation definition. If no icon has been defined with an annotation in a class, a generic icon is used (e.g. the “T”-icon for types).

The labels of components are displayed with two different background colors: A red background symbolizes that the component is defined with the visibility protected. Public components are labeled by a green background. Hence, the developer can figure out fast if a component can be accessed outside of the defining class.

A filter has been implemented for the outline view which can be triggered by the very left icon in the view’s toolbar. By default all inherited features of a class are displayed, including components, equations, and algorithms
that are defined inside extended classes. Hence, the content of classes is displayed in a “flattened” way. Inherited components are labeled with an italic font to make them easily distinguishable from components defined in the current class. If the filter is applied, only features of the current class without inherited elements are displayed.

Two further settings are accessible via the toolbar, which are shipped with the Eclipse framework: When activating the very right icon (two yellow arrows), the outline view is linked with the active editor. Therefore, when selecting an element inside the outline view, the current cursor position is updated accordingly inside the editor and the element is selected. The icon in the middle (“a-z” icon) sorts the elements inside the view. When selected, the elements are sorted alphabetically, otherwise the order is the same as the order in which the elements are defined inside the document.

### 7.2.4 Modelica Documentation View

Two different kinds of documentation are used in Modelica: Comments that are defined as Strings in quotations for a brief description of the purpose of a class or component declaration. This kind of comment is used to provide developers a quick overview. In OneModelica these comments can be dis-
Figure 7.13: HTML documentation of the DoublePendulum model

played in a pop-up when the user moves the mouse over references to the element (e.g. components used in an equation).

The second type of documentation is defined as HTML-conforming text embedded in annotations and is used to describe the properties and behaviors of models in detail. The documentation can also contain images and references to other models and components if this is necessary for explaining the behavior and usage of the documented element. The view in Figure 7.13 displays the documentation defined inside an annotation of the DoublePendulum model in the MSL. According to [Modelica Association, 2012] the HTML-based documentation must be defined as demonstrated in Listing 7.2. The annotation contains a reference to the class `Documentation`. The HTML documentation is then set as a String to the `info` parameter. The documentation is defined inside the HTML-tags.

In OneModelica the HTML-code is rendered and displayed in a separate view. If the user clicks inside a class or on a class reference, the HTML-code is retrieved and displayed in the view. The rendering is performed by an embedded web browser. The standard web browser of the operating system is automatically provided by the Eclipse framework.

```plaintext
annotation (Documentation(info="<html></html>");
```

Listing 7.2: Modelica documentation annotation
Chapter 7. OneModelica

7.3 Wizards

Wizards are specialized dialogs of the Eclipse framework helping users in creating new objects. The user is guided through the creation process step-by-step. The properties of the new object need to be provided by the user in each consecutive step. The user can re-do previous steps during the creation process in order to revert or change properties. Furthermore, the user is assisted by errors and warnings and their accompanying messages during the object creation. Context sensitive help can be added to the wizard that provides further information about the properties that need to be set. The creation of objects by wizards can only be finished if no errors are present.

OneModelica has several wizards that assist the user in creating new Modelica specific objects like Modelica projects and experiments. The wizards are explained in detail in the next sections.

7.3.1 Modelica Projects

The Modelica project wizard assists the user in creating new projects for the development of Modelica models. Figure 7.14 shows the wizard. The user must define a name for the project that he or she wants to create and the name has to be unique, which means that there cannot be a project with the same name in the workspace. Otherwise an error message is displayed. Besides, the user can provide a folder name for the source folder that is created in the Modelica project. The name src is set as default.

When selecting the question mark icon in the lower left of the wizard, a context sensitive help is opened on the right side. Basic information about the wizard is provided inside the opened help page. An example of a newly created Modelica project is displayed in Figure 7.15. Besides the source folder src folders for experiments and physical models are created auto-
Projects and libraries can be referenced and browsed inside the Plug-in Dependencies. Since the MSL is always added by default, it can already be browsed in the example project (the MSL is abbreviated with MSL).

### 7.3.2 Model Elements

Modelica is an object-oriented language, thus it supports the encapsulation of model behavior in objects. Classes are templates of objects that define behavior (equations and algorithms) and states (components). Modelica distinguishes between different types of classes (e.g., class, package and model). The MLSpec [Modelica Association, 2012] defines that classes should be located inside packages and that the folders in a file system that represent a package should contain a document with the name “package.mo” in which a package is declared with the same name as the folder.

A wizard has been implemented for OneModelica that assists the user in creating new classes of any allowed type. Package folders including the “package.mo” file are created automatically when needed, for example if a class is created within a package that is currently not available. Figure 7.16 displays the wizard. The user must provide a path to the source folder of the new document in the first text field. He or she can select the browse button next to the text field which will open a dialog displaying all source folders in the currently selected Modelica project. If a source folder (or a sub-folder of a source folder) was selected in the Project Explorer while the user opened the wizard, the entry is filled with the selected package. If the user selected a sub-folder, the wizard’s second text field is filled with a corresponding package declaration. Again, the user can browse the file system to select a package where the new model shall be located. The dialog is displayed on the right side of Figure 7.16. If the user defines a package that does not already exist, the folder structure is created automatically when the finish button is pressed.

In the third text field a name for the new model element must be defined. The user can select the kind of model that shall be created in the drop-down
list. The last text field lists the classes that will be extended by the new model element. Classes can be searched by selecting the “add” button. The model type search dialog described in Section 7.4.6 is reused for this purpose. Previously selected classes for extension can be removed from the text field.

When the finish button is pressed and a new Modelica document is created. The file name equals the model’s name with the file extension “.mo”. Inside the document a model element is declared with the package, model name and extends clauses as defined in the wizard.

### 7.3.3 Physical Models

The physical models described in this section correspond to the definition used in the OneWind framework. Physical models are an abstract class of models that can be simulated but do not refer to a single modeling language. The physical models are used for arbitrary model types but by the abstraction can be used similarly by the products of the OneWind product family. Physical models contain a reference to a concrete model instance, in the case of OneModelica the referenced models are always defined with the Modelica modeling language. In OneModelica this concept has been retained unchanged for re-usability in the OneWind framework although simulatable Modelica models may not represent a real world physical object (control algorithms an state machines can, for example, be modeled and simulated with Modelica).

All Modelica classes of type class and model that are not declared to be partial (i.e., the class is incomplete and needs to be refined by a sub-class in order to be used) can potentially be used for simulation. In
Chapter 7. OneModelica

7.3. Wizards

Figure 7.17: Physical model wizard

OneModelica the user needs to select a class by creating a physical model in order to perform a simulation. The physical model can then be used with experiments that configure and start the simulations. The instance tree of physical models can be browsed inside the Project Explorer (see Section 7.2.2). By declaration of a physical model it becomes obvious which model can be used as the root model for simulation. This can help keeping an overview, for example when load cases need to be simulated for certain components regarding certification.

Figure 7.17 displays the physical model wizard. In the first two dropdown boxes the user can select the Modelica project and the model folder in which the physical model will be created. This is automatically selected when a resource inside a Modelica project is selected. The third entry displays the chosen Modelica model with its full qualified name. The dialog on the right-hand side of the Figure is opened by selecting the button “Select Model”. A tree view displays the package hierarchy with all classes and models inside the selected project. The items of the tree can be filtered to search for models inside large package hierarchies.

The physical model name must be defined in the last text field of the wizard. A name is proposed when a model is selected. However, the user can change it to a custom name. When the user selects the finish button, a new resource is created that serializes the settings of the physical model descriptor, including a reference to the simulatable model (the Modelica model), as a data structure that has been modeled with EMF.

7.3.4 Experiments

An experiment always references a physical model that is being simulated when the experiment is executed. The simulation tool that is used for the translation of the Modelica code to executable code, such as C or Java, and
for the execution of the simulation is specified in the experiment descriptor. The user provides simulation parameters, e.g. simulation start and stop time, selected solver, during the creation of an experiment in the experiment wizard.

Figure 7.18 displays the first page of the experiment wizard. The Modelica project and the experiment folder, which is the folder where the experiment resources will be located in, need to be indicated. If a workspace resource has previously been selected, suggestions are provided. The abstract physical model dropdown list contains all physical models in the selected project that were previously defined with the physical model wizard (Section 7.3.3). A name for the experiment is automatically suggested, based on the name of the selected physical model, which can be changed by the user.

The next page of the wizard displays the registered simulation tools that the user can select for simulating Modelica models (Figure 7.19). Currently the following tools are supported:

- Dymola
- Dymola MOS-controlled (The user defines a custom MOS-script file that will control the simulation. Only the simulation file preparation, such as merging all Modelica documents from referenced projects into one folder, is performed by OneModelica)
- JModelica.org
- OpenModelica
- OpenModelica MOS-controlled (Similar to Dymola MOS-controlled.)

Based on the selection of the simulation tool, the settings page in the last step of the wizard changes accordingly (Figure 7.20). Typical simulation parameters of simulation tools are start and stop time or the used integration method. In case of a MOS-controlled simulation tool selection, the user only needs to select the script file, that will control the simulation.

When the finish button is selected, the settings are saved inside the experiments folder and can later be used for the execution of simulations.

7.3.5 Modal Reduction and Turbulent Wind Field Generation

In wind turbine simulations, flexible multibody dynamics is widely used to describe the large-scale motion and flexible deformation of blades. A flexible multibody system is based on the floating Frame of reference formulation. In a standard flexible multibody method, a finite element discretization is often combined with the multibody formulation to describe the beam flexibility.

In the finite element method, the flexible beam is discretized into several beam elements. Each element is described with local shape functions
Chapter 7. OneModelica

7.3. Wizards

Figure 7.18: Experiment wizard location page

Figure 7.19: Experiment wizard simulation tool selection page

Figure 7.20: Experiment wizard simulation tool settings page
weighted by nodal displacements. The numerical accuracy of the finite element method is dependent on the number of elements. A large number of elements would require quite a lot of computation, which would lead to unreasonably long computation times that are usually not practical for wind turbine simulation. A modal reduction approach can dramatically reduce the number of DOF (degrees of freedom) of the beam by transforming nodal coordinates to modal coordinates [Thomas et al., 2014].

The transformation of structure components into modal reduced representations has been implemented in Java. This was necessary due to problems with large matrices when calculating the eigenfrequencies directly in Modelica.

In OneModelica, input models for the transformation are automatically recognized by analysis if the contained model extends a predefined template class from the OneWind library. In this case a marker is added to the model files. Input models need to be of type record and must extend the record ModalInput which is included in the OneWind Modelica Library. By right-clicking on the model file, the user can open the transformation dialog and provide a path and file name for the generated modal representation (Figure 7.21).

The record ModalInput defines distributed structural properties. At the user defined record level, parameters have to be redefined and the needed information for the FEM structure model have to be provided. In this way, the usage of the same data for modal and finite element model is ensured. The parameters are then extracted from the Modelica file and used by the transformation code developed in Java. The generated Modelica record with modal data contains the modal reduced mass and the stiffness matrix as well as the eigenforms and can be used with modal components.
Changes in the structure model can easily be transferred to the modal component by the regeneration of the modal record. The same workflow is used for the generation of turbulent wind data. OneModelica provides functionality for the generation of turbulent wind (according to Kaimal [Kaimal et al., 1972]). Input files for Modelica models are generated in binary or ASCII format. A template record must be extended and parameterized. The generated wind file can then be used by a Modelica wind field generator that is included in the OneWind Modelica Library. The user can reuse the generator and provide a path to the generated wind file.

7.4 Dialogs

While wizards aim at creating new objects, Dialogs allow to view and change properties of already existing elements in the workspace. A typical example is a search dialog that helps to find text in documents or files by name in a folder structure. The following sections explain the dialogs that were implemented for OneModelica.

7.4.1 Model Nature

Eclipse uses the concept of natures to configure projects for the use with languages, such as Java or C++. OneModelica provides a model nature for the use of Modelica. If the user creates a Modelica project with the wizard described in Section 7.3.1 the model nature is automatically configured for the project.

A model nature can be added in order to enable Modelica-specific behavior to already existing projects. The wizard is displayed in Figure 7.22. For the OneWind project to different kinds of resources are managed in projects: Modelica resources as used in OneModelica and EDD resources which can be transformed into simulatable models. The type of resources
can be selected in the dropdown box of the wizard. In OneModelica only Modelica resources can be selected. A source folder needs to be selected in the text box that may already contain Modelica documents. Additional source folders can be added later.

When the model nature is configured for a project, the Project Explorer enables the nature specific functionality as described in Section 7.2.2.

7.4.2 Experiments Configuration Dialog

Once an experiment has been created with the experiment wizard (Section 7.3.4), it can be used for simulation. A dialog is opened before the simulation is executed, allowing the user to re-configure the experiment settings. Figure 7.23 displays the dialog where any setting previously defined for experiments can be changed. The three pages of the experiment wizard are displayed inside tabs. The user can change the name and location of the experiment in the tab labeled “Experiment”, change the selected simulation tool in the tab named “Compiler Selection” and configure the simulation tool specific settings in the tab with the label “Properties Page”. When the user selects the button “Run” the simulation is executed.

7.4.3 Result File Parsers and Plots

If an experiment has successfully been executed, the results are saved inside a file. For assessment of the results plots are needed that visualize the results to the user. Two components are necessary for post-processing of
simulation results: result file parsers and plot dialogs. The file parsers have been implemented for various output formats of the simulation tools. The supported formats are:

- Comma separated values (*.csv)
- Matlab output format - binary and ASCII format (*.mat)
- Plot format (*.plt)

Experiment results can be displayed inside the project explorer using the file parsers. The results are presented in a hierarchical way when the user expands the experiment. The qualified names of the components are used to restore the component hierarchy of the simulated model in the way that, for example, \texttt{revolutel.frame.a} in the MSL’s example model \texttt{DoublePendulum} is a sub-component of \texttt{revolutel}.

The user can select components for plotting by right-clicking on an element of the result hierarchy tree and selecting the menu entry that opens the create plot dialog (Figure 7.24). A title can be given for the plot and the settings are saved in a file inside the workspace, when the save check-box is selected. This allows the user to re-open and to modify the plot later on. The user can select the project and folder where the plot will be saved to.

The element that the user already selected for plotting is automatically added to the value section that lists all plot elements that will be displayed. The user can add arbitrary additional elements whereby he or she has to specify which two values are chosen for the x and y axis. For time series plots one of the selected values needs to be the time, while the other value is taken from a component of the simulated model. It is also possible to select two values from the model, for example to plot resulting displacement from...
incoming forces. Another feature of the plot mechanism is that results from different files can be plotted together. This makes it possible to compare results from different simulation tools that processed the same model, or to compare the behavior of a model regarding the selected solver.

If the user selects the check-box “Multiple Y-Axes”, the plot lines are not scaled to equally fit the y-axis of the plot diagram, but multiple scales will be displayed for each line instead.

Figure 7.25 shows the plot view that displays a plot element with two lines with the same y-axis. Two forces are plotted over time with the previously defined colors. The user can zoom the plot to investigate results for a certain time step. The icons in the plot view window trigger the following actions:

- create a FFT plot of the displayed values in a new plot window
- export the plot as a PNG image
- save the plot element
- export the plot values to a CSV file
- show the properties of the plot element (this opens the properties view allowing the user to change line colors etc.)

The JFreeChart framework\textsuperscript{20}, that has been used for implementing the plot view, additionally allows to print, zoom and range the displayed plot by right-clicking onto the plot view.

\textsuperscript{20}http://www.jfree.org/jfreechart/
7.4.4 Simulation Tool Location

OneModelica is flexible in integrating new simulation tools. The tool needs to be registered as an OSGi-service and can then be selected for simulation during experiment creation. On the Linux operating systems the path to the simulation tool executable is usually registered during installation. However, on the Windows operating systems this is not the case. Several installations of different versions of a simulation tool may also exist on the system where the user wants to select which version he or she wants to use. Hence, the user may need to provide information about the installation location of the simulation tools.

The installation location is configured using an Eclipse properties page. An abstract class has been implemented for this purpose, that has been customized for the simulation tools that are currently supported by default in OneModelica. Figure 7.26 shows the Eclipse preferences view with the compiler location entries. If no location is provided by the user, the default installation locations of the simulation tools are searched for valid executables. In this case it is not guaranteed that the compiler can be found, which will result in an error message, or that the selected compiler is the newest version installed on the system. By selecting the “Restore Defaults” button, the user can delete the previously selected path.

7.4.5 Standard Library Configuration

Models can be provided for reuse as libraries [Tummescheit, 2002]. The models are usually designed in a way that they can easily be extended and adapted to the user’s requirements. The MLSpec [Modelica Association, 2012] defines that every Modelica IDE must provide the MSL. Other free and commercial libraries are available that provide models for specific design aspects, like the OneWind library for wind turbines.
Chapter 7. OneModelica

7.4. Dialogs

OneModelica automatically provides 4 libraries:

- Modelica Standard Library
- Modelica Synchronous Library
- Modelica StateGraph2 Library
- Modelica LinearSystems2 Library

When a new Modelica project is created, the MSL is automatically referenced to enable the use of the therein defined models. This behavior can be customized in OneModelica. Figure 7.27 displays the configuration dialog. New libraries can be added by selecting the button “Add Library”. The libraries must be in a special format that is described in Section 7.5. All libraries that are checked in the list will automatically be referenced by new Modelica projects and the contained models will thereby be available to the user.

7.4.6 Open Model Dialog

When many models exist in the workspace and large libraries are used, it can be hard to find the location of a model. The Open Model dialog helps searching in all models that are available to the user. Figure 7.28 displays the dialog. The user can search for a model by typing in the name into the input field. The fully qualified name of the model is considered. In the example the asterisk sign is used to display all models whose name starts with “Frame” and that are defined inside a package with a name that ends with “Interfaces”. The dialog in the Figure lists four matching model declarations from the Modelica standard library and two models defined
Chapter 7. OneModelica  7.5. Project Management and Libraries

Figure 7.28: Open model dialog

in the OneWind library for wind turbine development. The origin of the model is indicated in the info box below the list. The fully qualified name of the model is displayed followed by the library name “MSL” in square brackets. If the user selects the “OK” button, the model is opened inside a new Modelica editor instance.

7.5 Project Management and Libraries

For the sake of reuse, Modelica documents should be structured in projects. When defining a wind energy plant, every component like tower, nacelle or rotor blade should be encapsulated in its own unit. If the components are divided into different projects, they can be interchanged easily. This helps the developer to survey the structure of the designed physical systems. Furthermore the reuse of functionality from libraries like the Modelica Standard Library is essential.

Basically referencing objects in separate projects or libraries means resolving links from one document to another. In order to be able to find references quickly, meta data must be provided, that holds additional information about the class structure and location of Modelica files (see Section 7.1.4). The central data in this structure is called ModelResource. A ModelResource can contain source folders. All Modelica source files contained in these source folders belong to the same ModelResource.

At first sight the proposed project structure looks quite similar to the one Eclipse uses for plug-in projects. In fact many concepts are reused but also
altered to meet the requirements of a Modelica IDE (see Figure 7.29). Using ModelResources instead of projects as management units allows to have several name spaces inside one project. This is important when simulations are performed. The source files of several simulations can be kept in the same project and allow to keep track of source code changes between different simulations. In the project based approach it would be impossible to keep multiple class definitions with the same qualified name.

To enable linking, each experiment gets its own ModelResource that knows the files used for simulation. The ModelResource mechanism is also used to enable referencing other projects in the workspace. Projects can be exported as compressed and possibly encrypted libraries. The meta data is kept inside the archive, thus no further analysis of the contained source files is needed when libraries are reused. The creation of libraries serves two purposes. First it allows to assemble specific functionality into one library that can be shared among users or maybe even sold to customers. Additionally, using libraries speeds up the development process, because handling a huge amount of source files can lead to a slow development environment and increasing reaction time of the systems on user interaction.

Using EMF turns out to be very helpful, because the data defined in EMF automatically has a persistence model, which means it is automatically saved to persist between sessions. Also references between EMF-based files are resolved automatically. Furthermore it provides a system for change notification that allows to react on any changes of the meta data.

### 7.5.1 Creating Libraries from Projects

Modelica projects can be exported to libraries in order to distribute models to other developers. Figure 7.30 displays the export dialog that can be

![Figure 7.29: Data structure of the Modelica IDE](image)
opened by right-clicking on the project that shall be exported. All interdependencies of projects beginning with the selected project are resolved and displayed in the list labeled with “Referenced Resources”. All projects that are required can be added to the library file. The location for the resulting library file can be provided. All selected projects are copied into a folder together with their index files when the user clicks on the “Save” button. A library-specific ModelResource is created in the folder that references the selected projects inside the folder. Finally, the folder is compressed and can then be distributed and reused by other developers.
Chapter 7. OneModelica

7.6 Related Work

Various tools, commercial and non-commercial, are available for the development and simulation of Modelica models. A selection of commonly used tools is described below:

OpenModelica [Fritzson et al., 2005] is a collection of open source tools. The central tool component in this set of tools is the Modelica model compiler [Sjölund et al., 2014] that is responsible for parsing/un-parsing, compilation, interpreting, and validation of Modelica documents. The connection editor OMEdit [Asghar, 2010] allows graphical editing of Modelica models. Connections between components can easily be created by drawing a line between the connectors. MDT [Pop, 2008], an Eclipse Plug-in for textual manipulation and debugging of Modelica models, can be downloaded separately. Communication between the tools, especially with the compiler, is implemented via a CORBA [Group, 2006] interface. Furthermore, tools for shell scripting (OMShell), Python based scripting (OMPython [Gane- son, 2012]), interactive notebook-like editing (OMNotebook), model design optimization (OMoptim [Thieriot et al., 2011]), dynamic optimization of models with goal functions etc. specified within the model [Bachmann et al., 2012], graphical modeling and dynamic verification of requirements versus usage scenarios of dynamic system behavior (ModelicaML [Pop, 2008; Schamai, 2013]), FMI import or export of models in the Functional Mockup Unit format [Chen et al., 2011], and debugging of both equation-based models and algorithmic Modelica models [Pop et al., 2012] are provided.

JModelica.org is an open source tool for the simulation and optimization [Åkesson, 2007] of Modelica models. Modelica code is compiled to C for simulation. JModelica.org provides interfaces for extension by users. Simulations and optimizations can be performed with Python scripts. The FMILibrary [Andersson et al., 2011] can be used for model exchange and co-simulation with other simulation environments. A language extension (Optimica) is provided that helps to dynamically optimize Modelica code.

Dymola [Dassault Systemes, 2014] is a commercial environment for the development of Modelica models. It integrates textual and graphical editors. During compilation optimizations are performed resulting in fast simulations. Models with 3D annotations, such as models from the Modelica multibody library, can be visualized in a 3D-view during and after simulation. Commercial libraries can be encrypted and used with Dymola making it possible for model developers to sell their models without revealing the source code. Additional features can be added which need to be purchased separately, like the model management feature which includes regression tests of models.


7.7 Model Documentation and Icons

As described before, Modelica models can be annotated with detailed documentation using HTML. Usually the theory and usage of the implemented models is described to guide the user in using the models. The documentation should be extracted to allow easy access for developers who are using the defined models. In libraries the documentation can also help surveying the provided functionality.

Xtext implements the concept of generators that are automatically executed when a source file has been changed. This can, for example, be used to generate executable code from DSLs. For Modelica this mechanism is used for the generation of HTML-files. As soon as a Modelica file has been changed, the documentation is extracted and stored in a separate file. The documentation is stored inside a folder with the name “help” in the project’s root.

Besides documentation, additional information about models is stored in the same document that helps developers in using the model correctly. Figure 7.31 displays the generated files in an example project and the generated documentation of a gear model. The generated icon is displayed in the document followed by the short description defined inside the model. The HTML-based documentation is then followed by information about extended models. Finally, a table describes the provided parameters including their default values.

Besides documentation, icon information is stored in annotations. This is used for the visualization of model instances in connection editors. Icons are defined in a Modelica-specific format by graphical primitives like polygons, ellipses and text. The icon information is also extracted by the generator. The Modelica-specific information is transformed to a Java AWT representation and can thus directly be displayed in OneModelica. Additionally, the icon is saved as a PNG image and can be used, for example, inside the model documentation.

7.8 Conclusion and Future Work

With the help of model-driven software development it was possible to implement a Modelica IDE in a rather short period of time. Using generated code ensures high quality products and reduces the implementation time by automatically creating frequently used components like data structure and serialization mechanisms. Since all generated code is based on the same data description (Ecore), interoperability is guaranteed. Therefore augmenting the generated editor with additional functionality became easy.

Utilizing the introduced IDE Modelica developers are able to create models easier and faster. The validation and referencing mechanisms ensure correctness throughout the development process. The views on the Modelica data structure help surveying large projects. Encapsulating source code
Chapter 7. OneModelica 7.8. Conclusion and Future Work

Figure 7.31: Documentation and icon generation

Motor inertia and gearbox model for r3 joints 1, 2, 3

Information

Models the gearbox used in the first three joints with all its effects, like elasticity and friction. Coulomb friction is approximated by a friction element acting at the “motor”-side. In reality, bearing friction should be also incorporated at the driven side of the gearbox. However, this would require considerable more effort for the measurement of the friction parameters.

Extends from

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>(i)</td>
<td>-105</td>
<td>gear ratio</td>
</tr>
<tr>
<td>Real</td>
<td>(c)</td>
<td>43</td>
<td>Spring constant</td>
</tr>
<tr>
<td>Real</td>
<td>(d)</td>
<td>0.005</td>
<td>Dampner constant</td>
</tr>
<tr>
<td>Torque</td>
<td>(Rv)</td>
<td>0.4</td>
<td>Viscous friction torque at zero velocity</td>
</tr>
<tr>
<td>Real</td>
<td>(Rv1)</td>
<td>(0.13/160)</td>
<td>Viscous friction coefficient (R = Rv0 + Rv1 \cdot \text{abs}(q))</td>
</tr>
<tr>
<td>Real</td>
<td>peak</td>
<td>1</td>
<td>Maximum static friction torque is peak*Rv0 (peak (\geq 1))</td>
</tr>
</tbody>
</table>
into libraries speeds up the user interaction and encourages the developer to create models in a component-based way. This helps creating frameworks that can be shipped and reused as libraries.

A nice feature to have is a graphical editor that allows the developer to graphically compose models from model components, like the graphical editors of OpenModelica and Dymola do. Since Ecore is used as basis for the data, Graphical Modeling Project\(^\text{21}\) and Graphiti\(^\text{22}\) emerge as suitable frameworks for the implementation of such a connection editor. During development it became clear that the Graphical Modeling Project is not suitable since it requires a simple meta model to work with, which Modelica is not. An implementation with Graphiti is already in an advanced stage (see Figure 7.32), however technical issues appeared during development. The icons of components must often be rotated in order to visualize the connection diagram correctly, however rotation is not implemented in the Graphiti framework, yet. Currently a custom rotation mechanism is being implemented.

\(^{21}\)http://www.eclipse.org/modeling/emf/
\(^{22}\)http://www.eclipse.org/graphiti/
Chapter 8

Conclusions and Future Work

An important goal of this dissertation work is the development of a Modelica IDE with model-driven software development that integrates into the OneWind framework and supports the idea of generating simulatable code from an abstract data layer. This Chapter outlines the contributions made. Furthermore, possible future work is described that could further enhance the IDE and support the iterative model development workflow described in Section 1.1.

8.1 Contributions

Five research questions have been defined in Section 1.5. The contributions made to answer the questions are described in the following.

8.1.1 Use of DSL Technologies for the Development of a Modelica IDE

The presented IDE OneModelica has been developed with the DSL tools, EMF and Xtext which are available for the Eclipse framework. Hence, a large part of the code is based on model-driven software development technologies and has been generated from models. By using Xtext, a Modelica editor and accompanying views could be generated automatically based on a grammar definition. Although extensive adaptions need to be performed for complex DSLs like Modelica, the flexible and extensible implementation of the model-driven software development tools were successfully used.

Using model-driven software development allowed the implementation of OneModelica in a generic way such that it can be easily extended, for example, by providing new simulation tools that execute simulations or new
file parsers. Transferring best practices from general purpose languages to Modelica turned out to be beneficial. The separation of model components in projects reduces the necessary code that a developer has to survey and helps concentrate on the main tasks of his development. One benefit of the model-driven software development approach is that the models are represented in a uniform way and large data structures can be processed with available tools. Helper methods have been implemented which simplify the analysis of large and interconnected parse trees of Modelica documents. This has been used for various purposes like model validation or displaying the structure of models and model instances in views.

Moreover, models have been defined for project management, Modelica library support and experiment definition. This allowed separation of Modelica code in projects and libraries in a way that is not supported by other Modelica tools that are available nowadays. Hence, domain experts can concentrate on the models they need to edit and are not distracted by code that they do not necessarily need to be present in the workspace. By using EMF-based models, it was also possible to reuse existing model instances, e.g., experiment definitions that are referenced within test cases of the unit test framework that has been integrated into OneModelica.

Validation of code and the provided automatic refactorings, for example package adjustment that facilitate the modification of existing code, are important features for a mature IDE. These features could be implemented since the used frameworks already provide basic functionality that could be extended for domain specific issues.

8.1.2 Model Validation for Language Compliance

Language compliance validation is important since this ensures correctness and that Modelica models can be simulated. A constraint language approach with OCL has been compared to a Java implementation regarding readability and maintenance of constraints as well as performance issues. Type and unit checks ensure that expressions are valid and units are not mixed in an unintended way which can lead to severe misinterpretation of simulation results. Immediate checks during model development has the advantage of the developer recognizing errors while he or she is writing code. If this feedback is delayed until a simulation is started, the development process can be considerably slowed down. Code style rules support the collaboration of multiple users since code needs to be sufficiently documented.

8.1.3 Domain Specific Model Semantics

Model semantics validation helps in restricting how model components can be used and connected to each other. This is especially helpful for the development of libraries and can prevent misuse and instruct the user in how the model components are meant to be used. The implemented validation framework uses roles to annotate models with additional semantics. The
framework not only supports the use of Modelica but also potentially supports any language that has been modeled with EMF. It has been demonstrated that roles for manifold design aspects, e.g. structural validity and rated power design of wind turbines, can be defined and checked interactively with the proposed framework. The definition of additional semantic rules are beneficial if many variants of models exist. In case of the rated power, models can be ensured that no erroneous model configurations are made which would result in the need for re-calculation. When calculating load cases for the certification of wind turbines, this could otherwise lead to a significant slow-down of the certification process.

8.1.4 Testing the Behaviour of Physical Models

Model development is a recurring process. It must be guaranteed that model changes do not negatively influence other models. Unit tests are a feasible way to check for expected model behavior, for example, by regression testing. The test framework integrated in OneModelica allows to automatically test Modelica components by regression tests and test result analyses. Custom test components can be added by users which allows to extend the framework for the user's needs. Flexible visualization of the test results helps in using the test in different environments, for example, directly in the Modelica IDE or with build environments like Hudson.

8.1.5 Code Generation for Iterative Model Development

Another goal of this work was the integration into the OneWind framework that defines models of wind turbines in an abstract way. For simulation, the abstract models need to be transferred into a language or tool specific representations that can be simulated. It was shown that the transformation to Modelica code can be performed. Therefore, a Modelica library has been developed with OneModelica. Parameters of the abstract representation are stored in a Modelica record which is used for the parameterization of the library models. The generated un-parser is then used to create a model instance that can be simulated, e.g. in OneModelica. Code for modal reduction has been reused in OneModelica. This allows the creation of fast low-detailed models from high-detailed models which are commonly used for tower and rotor blade models in wind turbine simulations.

8.2 Future Work

Further improvements are planned for future work. Although the performance is sufficiently high for the development of large libraries as tested with the OneWind library, it reaches its limits when the MSL itself is edited. The linking can be further improved to tackle this problem. However, it is
limited since the Xtext framework itself reduces performance since it keeps track of all references and referenceable objects in the workspace. It may also be possible to accelerate the constraint checking by further caching. However, this would lead to an increased memory demand.

One important topic that is already being taken care of is the development of a connection editor. This allows the easy connection of model components in large models by providing a topologic view. The prototype implementation with Graphiti is promising. However, problems with the rotation of graphical elements must either be solved inside the framework itself or by a custom implementation.

Regarding the validation for language compliance, further constraints need to be implemented. Although many important constraints like type safety and valid name declaration are already checked, there are many additional constraints that to some extent are not defined clearly in the Modelica Language Specification (MLSpec). A good strategy would be to create a catalogue of constraints together with the Modelica community in order to list and discuss all constraints that exist for the language.

Role-based validation is a viable way of defining semantic constraints for the use of model components. A goal is to annotate the OneWind library for wind turbines sufficiently to prevent the user from composing models that violate constraints from a selection of design aspects, such as rated wind power, certification specifications or the location of assembly. Furthermore, it is desired to apply the role validation to the MSL. Since the library is frequently used, it could be helpful to guide the user, especially if he or she is inexperienced with Modelica, in correctly using the pre-defined model components.

Modelica unit testing is recently used for automatic testing by the engineers at Fraunhofer IWES, and most of all, by regression tests. Further test components should be implemented that help to check if models behave as expected. This can especially be helpful for non-deterministic models like waves and wind. Furthermore, automatic test case creation for standard tests, like creating a regression test as soon as a simulation has been executed, is desirable. This would support the developer and ensure that the test coverage is sufficiently high.
Appendix A

Modelica Syntax

```plaintext
grammar on.modelica.xtext.Modelica
    hidden(WS,ML_COMMENT,SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
generate modelica 'http://www.onewind.de/language/Modelica'

ASTModelicaSourceFile:
    {ASTModelicaSourceFile}
    ("within" importedNamespace=QN? ";")?
    (elements+=AbstractMoClass ";")*;

AbstractMoClass:
    MoClass | MoModel | MoRecord | MoBlock | MoPackage |
    MoConnector | MoType | MoFunction | MoOperator;

MoConnector:
    {redeclare?='redeclare'}? {final?='final'}?
    {inner?='inner'}? {outer?='outer'}?
    {encapsulated?='encapsulated'}? {partial?='partial'}?
    {expandable?='expandable'}?
    'connector' content=AbstractContent;

MoType:
    {redeclare?='redeclare'}? {final?='final'}?
    {inner?='inner'}? {outer?='outer'}?
    {encapsulated?='encapsulated'}? {partial?='partial'}?
    'type' content=AbstractContent;

MoFunction:
    {redeclare?='redeclare'}? {final?='final'}?
    {inner?='inner'}? {outer?='outer'}?
    {encapsulated?='encapsulated'}? {partial?='partial'}?
    {pure?='pure' | impure?='impure'}?
    {operator?='operator'}?
    'function' content=AbstractContent;

MoOperator:
    {redeclare?='redeclare'}? {final?='final'}?
```
Appendix A. Modelica Syntax

```
{inner?=‘inner’}? {outer?=‘outer’}? 
{encapsulated?=‘encapsulated’}? {partial?=‘partial’}? 
‘operator’ content=AbstractContent;

MoClass: 
{redeclare?=‘redeclare’}? {final?=‘final’}? 
{inner?=‘inner’}? {outer?=‘outer’}? 
{encapsulated?=‘encapsulated’}? {partial?=‘partial’}? 
‘class’ content=AbstractContent;

MoPackage: 
{redeclare?=‘redeclare’}? {final?=‘final’}? 
{inner?=‘inner’}? {outer?=‘outer’}? 
{encapsulated?=‘encapsulated’}? {partial?=‘partial’}? 
‘package’ content=AbstractContent;

MoModel: 
{redeclare?=‘redeclare’}? {final?=‘final’}? 
{inner?=‘inner’}? {outer?=‘outer’}? 
{encapsulated?=‘encapsulated’}? {partial?=‘partial’}? 
‘model’ content=AbstractContent;

MoRecord: 
{redeclare?=‘redeclare’}? {final?=‘final’}? 
{inner?=‘inner’}? {outer?=‘outer’}? 
{encapsulated?=‘encapsulated’}? {partial?=‘partial’}? 
{operator?=‘operator’}? 
‘record’ content=AbstractContent;

MoBlock: 
{redeclare?=‘redeclare’}? {final?=‘final’}? 
{inner?=‘inner’}? {outer?=‘outer’}? 
{encapsulated?=‘encapsulated’}? {partial?=‘partial’}? 
‘block’ content=AbstractContent;

AbstractContent: 
CommonContent | AssignmentContent | EnumerationContent 
| ExtensionContent;

CommonContent: 
name=ID stringcomment=StringComment? body=Body ‘end’ 
classnamereference=AbstractContentReference;

AssignmentContent: 
name=ID ‘=’ timevariabilitytype=TimeVariabilityType? 
inputoroutputprefix=InputOrOutputPrefix? 
reference=AbstractContentReference 
{arraysubscripts=ArraySubscript}? 
{modification=Modification}? comment=Comment?;

EnumerationContent: 
name=ID "=" "enumeration" "(" 
{|enumlist=EnumerationList)? | ":") ")" 
comment=Comment?;

ExtensionContent: 
‘extends’ reference=AbstractContentReference
```
Appendix A. Modelica Syntax

\( (\text{statements}+=\text{AbstractStatement } "\);")\)*
'end' 'for' comment=Comment?;

WhileStatement:
'while' expression=AbstractExpression 'loop'
(\text{statements}+=\text{AbstractStatement } "\);")\)*
'end' 'while' comment=Comment?;

WhenStatement:
'when' whencondition=AbstractExpression 'then'
(whenstatements+=\text{AbstractStatement } "\);")\)*
elsewhenstatements+=ElseWhenStatement 'end' 'when'
comment=Comment?;

ElseWhenStatement:
'elsewhen' elsewhencondition=AbstractExpression 'then'
(elsewhenstatements+=\text{AbstractStatement } "\);")\)*

ExpressionListAssignment:
(" expressionlist=ExpressionList "\);") "::=
componentreference=AbstractContentReference "("
functioncallarguments=Argument? ")" comment=Comment?;

AbstractComponent:
(ImportClause | ExtendsClause | Component | Annotation
 | ExternalFunction | ReplaceableClass | AbstractMoClass
) ";";

ExternalFunction:
'external' language=STRING?
(componentreference=Reference "=")? name=ID
(" (expressionlist=ExpressionList)? ")"
annotation=Annotation?;

ExtendsClause:
'extends' reference=AbstractContentReference
(modification=Modification)? (annotation=Annotation)?;

ImportClause:
'import' (name=SimpleComponentName "=")?
importreference=Reference
wildcard="$.*"? comment=Comment?;

Component:
(redeclare?="redeclare")? (inner?="inner")?
(outer?="outer")? (final?="final")?
(replaceable?="replaceable")?
connectorprefix=ConnectorPrefix?
timevariabilitytype=TimeVariabilityType?
componentinputoutput=InputOrOutputPrefix?
type=AbstractContentReference (typearray=ArraySubscript)?
componentnames+=ComponentName
("," componentnames+=ComponentName)*
("if" ifexpression=Expression)?
comment=Comment?
('constrainedby' constrainvalue=QN
 (constraintmodification=Modification)?
...
ComponentDeclarationModification:
{ComponentDeclarationModification}
(declarationarray=ArraySubscript)?
declarationmodification=AbstractModification?
("=" (assignmentreference+=QN
assignmentmodification+=Modification
| assignmentexpressions+=Expression
))
);

ComponentName:
  name=ID
  componentDeclarationModifications=
  ComponentDeclarationModification?;

SimpleComponentName returns ComponentName:
  name=ID;

EquationExpression:
  leftexpression=Expression '='
  rightexpression=AbstractExpression
  (comment=Comment)?

FunctionCall:
  (prefix=AddOp)? reference=AbstractContentReference
  '(' argument=Argument? ')' (functioncallcomment=Comment)?

Argument:
  NamedArguments | FunctionArgument;

FunctionArgument:
  ('function' reference=Reference
   "(" arguments=NamedArguments? ")"
   | expression=AbstractExpression
  ) "" moreArguments=Argument | 'for' for_indices=ForIndices)?

ConnectClause:
  'connect' "(" componentreference1=Reference ","
  componentreference2=Reference ")" comment=Comment?;

Reference:
  (prefix=AddOp)? qn=QualifiedName? (reference=[Component|ID]
  | reference=[AbstractContent|ID]
)
(arraysubscripts=ArraySubscript)?
("." componentreference=Reference)?;

AbstractContentReference returns Reference:
  qn=QualifiedName? reference=[AbstractContent|ID];

ComponentReference returns Reference:
  qn=QualifiedName? reference=[Component|ID]
(arraysubscripts=ArraySubscript)?
("." componentreference=Reference)?;
Appendix A. Modelica Syntax

IfEquation:
"if" condition=Expression "then"
(ifequations+=AbstractEquation ";")*
elseifequations+=ElseIfEquation* ("else"
elseifequations+=AbstractEquation ";")*
"end" "if" (comment=Comment) ?

ElseIfEquation:
"elseif" elseifcondition=Expression
"then" elseifthenequations+=AbstractEquation ";")*

ForEquation:
"for" forindices=ForIndices "loop"
(equations+=AbstractEquation ";")*
"end" "for" (comment=Comment) ?

ForIndices:
indices+=ForIndex (";" indices+=ForIndex) *

ForIndex:
localComponent=ComponentName
(’in’ expression=AbstractExpression) ?

WhenEquation:
"when" whencondition=Expression "then"
(whenequations+=AbstractEquation ";")*
elsewhenequations+=ElseWhenEquation* 
"end" "when" (comment=Comment) ?

ElseWhenEquation:
"elsewhen" elsewhencondition=Expression "then"
(elsewhenthenequations+=AbstractEquation ";")*

ExpressionList:
expressions+=AbstractExpression
(";" expressions+=AbstractExpression) *

RedeclareClass:
’redeclare’ (each?='each')? (final?='final')? classdefinition=AbstractMoClass;

RedeclareComponent returns Component:
’redeclare’ (each?='each')? (final?='final')? 
connectorprefix=ConnectorPrefix?
timevariabilitytype=TimeVariabilityType? 
componentinputoutput=InputOrOutputPrefix?
type=AbstractContentReference 
name=ID (arraysubscripts=ArraySubscript)? 
(modification=AbstractModification)? comment=Comment ?

RedeclareReplaceable:
’redeclare’ (each?='each')? (final?='final')? 
elementreplaceable=AbstractReplaceable;

AbstractReplaceable:
ReplaceableClass | Component;
Modification:
"(" arguments+=AbstractArgument
"," arguments+=AbstractArgument)* ")"
"=" equalsexpression=AbstractExpression)?;

AbstractArgument:
ComponentModification | ReplaceableClass | RedeclareClass
| RedeclareComponent | RedeclareReplaceable | Component;

ComponentModification:
(each?='each')? (final?='final')?
reference=ComponentReference
(modification=AbstractModification)?
(stringcomment=StringComment)?;

ReplaceableClass:
(redeclare?='redeclare')? (inner?='inner')?
(outter?='outer')? (each?=‘each’)? (final?=‘final’)?
'replaceable'
modelicaclassdefinition=AbstractMoClass
('constrainedby'
reference=AbstractContentReference
(modification=Modification)?
)? comment=Comment?;

AbstractModification:
Modification | EqualsExpression;

EqualsExpression:
’=’ expression=AbstractExpression;

Annotation:
‘annotation’ modification=Modification;

EnumerationList:
enumerationliterals+=EnumerationLiteral
"," enumerationliterals+=EnumerationLiteral)*;

EnumerationLiteral:
componentname=SimpleComponentName comment=Comment?;

Comment:
{Comment}
stringcomment=StringComment? annotation=Annotation?;

ArraySubscript:
{ArraySubscript}
"[" subscripts+=Subscript ("," subscripts+=Subscript)* "]";

Subscript:
Colonn | AbstractExpression;

Colon:
colon=':';

AbstractExpression:
Appendix A. Modelica Syntax

Expression | IfExpression;

IfExpression:
   'if' condition=AbstractExpression 'then'
   then=AbstractExpression
   ('elseif' elseifConditions+=AbstractExpression 'then'
    elseifs+=AbstractExpression)*
   'else' else=AbstractExpression;

Expression:
   LogicalExpression
   ({Expression.left=current} operator=':'
    value=LogicalExpression
  )*;

LogicalExpression returns Expression:
   LogicalTerm
   ({Expression.left=current} operator='or'
    value=LogicalTerm
  )*;

LogicalTerm returns Expression:
   LogicalFactor
   ({Expression.left=current} operator='and'
    value=LogicalFactor
  )*;

LogicalFactor returns Expression:
   (not?'not')? left=Relation;

Relation returns Expression:
   ArithmeticExpression
   ({Expression.left=current} operator=RelOp
    value=ArithmeticExpression
  )*;

ArithmeticExpression returns Expression:
   Term
   ({Expression.left=current} operator=AddOp value=Term)*;

Term returns Expression:
   Factor
   ({Expression.left=current} operator=MulOp value=Factor)*;

Factor returns Expression:
   BasicValue
   ({Expression.left=current} operator=PowerOp
    value=BasicValue
  )*;

PowerOp:
   "^" | ".*";

MulOp:
   "*" | "/" | "." | "./";

AddOp:
Appendix A. Modelica Syntax

```
"+" | "-" | ".+" | ".-";
```

RelOp:
```
"<" | "<=" | ">" | ">=" | ">=" | ">>=";
```

BasicValue:
```
    IntegerLiteral | RealLiteral | NegativeValue | Boolean
    | StringLiteral | NewInstance | FunctionCall | Reference
    | BracketExpressionList | SquareBracketExpressionList
    | MultipleVector | ExpressionVector
    | ExpressionsWithForIndices | NamedArguments | EndLiteral
    | InitialLiteral;
```

NewInstance:
```
    'new' reference=Reference '([' arguments=Argument? '])';
```

Boolean:
```
    bool=BooleanLiteral;
```

InitialLiteral:
```
    string="initial()";
```

EndLiteral:
```
    string="end";
```

StringLiteral:
```
    string=STRING;
```

BracketExpressionList returns ExpressionList:
```
    (prefix=AddOp)? "(" expressions+=AbstractExpression
    ("," expressions+=AbstractExpression?)* ")";
```

SquareBracketExpressionList:
```
    (prefix=AddOp)? "[" expressionlist+=ExpressionList
    (";"); expressionlist+=ExpressionList)* "]";
```

Vector:
```
    MultipleVector | ExpressionVector;
```

ExpressionVector:
```
    (prefix=AddOp)? '([' list+=AbstractExpression
    ("," list+=AbstractExpression)* '])';
```

MultipleVector:
```
    (prefix=AddOp)? '([' vectors+=Vector (',}); vectors+=ExpressionVector)* ']')';
```

ExpressionsWithForIndices:
```
    (prefix=AddOp)? '([' expressions+=AbstractExpression
    ('for' forindice=ForIndices)? ']')';
```

NamedArguments:
```
    arguments+=NamedArgument (','); arguments+=NamedArgument*;
```

NamedArgument:
Appendix A. Modelica Syntax

reference=[ComponentName] "=" expression=AbstractExpression;

QualifiedName:
   ID '.' (QualifiedName)?;

QN:
   ID (WS? '.' WS? ID)*;

IntegerLiteral:
   plus?='+'? numberValue=INT;

RealLiteral:
   plus?='+'? numberValue=REAL;

NegativeValue:
   ('−' value=IntegerLiteral | '−' value=RealLiteral);

StringComment:
   STRING ('+' StringComment)*;

denum VisibilityType:
   PUBLIC='public' | PROTECTED='protected';

ConnectorPrefix:
   value='flow' | value='stream';

TimeVariabilityType:
   value='discrete' | value='parameter' | value='constant';

InputOrOutputPrefix:
   value='input' | value='output';

denum BooleanLiteral:
   FALSE="false" | TRUE="true";

terminal INT returns ecore::EInt:
   ('0'..'9')*;

terminal REAL:
   {'"" INT} |
   (INT {'"" (INT)?}?)? (("e" | "E") ("+" | "−")? INT)?

terminal ID:
   {'a'..'z' | 'A'..'Z' | '−'}
   {'a'..'z' | 'A'..'Z' | '−' | '0'..'9'}*
   | /*"*/ {'\' | \'} | 't' | 'n' | 'f' | 'r' | 'r'
   | '"" | '""'} |

terminal STRING:
   /*"*/ {'\' | \'} | 't' | 'n' | 'f' | 'r' | 'r'
   | '"" | '""'} | '*"";

terminal ML_COMMENT:
   '/∗'->'∗'/;

terminal SL_COMMENT:
   '///' !(\n' | '\r') ('\r'? '\n')?;
Appendix A. Modelica Syntax

```modelica
terminal WS:
    (' ' | '\t' | '\r' | '\n')+;

terminal ANY_OTHER:
    .;
```
Appendix B

Role Mapping Syntax

```plaintext
grammar on.rolemapping.Rolemapping
    with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.onewind.de/language/RoleModel" as rolemodel
generate rolemapping
    "http://www.onewind.de/language/RoleMapping"

Rolemapping:
    target=ReferencedMetamodel contents+=Element*;

Element:
    ConnectionMapping | RoleQuery | Mapping | Function;

Mapping:
    'maps' importURI=STRING;

ReferencedMetamodel:
    'target' importURI=[ecore::EPackage|STRING]
        ('as' alias=ID)?;

ConnectionMapping:
    'mapping'
    connection=[rolemodel::Connection] '=' type=[ecore::EClass]
        "extends" extendedMappings+=[ConnectionMapping]
        (',' extendedMappings+=[ConnectionMapping])?
        '{' ports+=PortMapping? (',' ports+=PortMapping)* '}'

PortMapping:
    port=[rolemodel::PortDefinition] '=' queries+=[RoleQuery]?
        (',' queries+=[RoleQuery])*;

RoleQuery:
    'RoleQuery' name=ID '{' variables+=Variable?
        (',' variables+=Variable)*
        cases+=Case (',' cases+=Case)* '}'

Variable:
```

163
Appendix B. Role Mapping Syntax

'var' name=ID type=[ecore::EObject] (selector=Selector)? ('.' query=AbstractQuery)?;

Function:
'function' returntype=[ecore::EClass] name=ID
'(' inputtype=[ecore::EClass] ')' (' variables+=Variable? 'return' returnvalue=FunctionReturn ')';

FunctionReturn:
VariableReference | AbstractQueryReference;

VariableReference:
'var' ref=[Variable];

AbstractQueryReference:
query=AbstractQuery;

Case:
'case' type=[ecore::EClass] ':' query=AbstractQuery;

AbstractQuery:
Query | CastQuery | FunctionCall;

FunctionCall:
operation=[ecore::EObject] '(' ')' (selector=Selector)? ('.' next=AbstractQuery)?;

CastQuery:
'as' '(' type=[ecore::EClass] ')' (selector=Selector)? ('.' next=AbstractQuery)?;

Query:
feature=[ecore::EObject] (selector=Selector)? ('.' next=AbstractQuery)?;

Selector:
ListSelectType | Position | InCase | SwitchType;

SwitchType:
'.' 'switch' '(' statements+=SwitchStatement ('.' statements+=SwitchStatement)* ')';

SwitchStatement:
type=[ecore::EClass] ':' switchQuery=AbstractQuery;

InCase:
'.' 'incase' '(' (check=EClassReference | check=BuiltInFunction) ')';

EClassReference:
reference=[ecore::EObject];

BuiltInFunction:
NodeValueCheck;

164
Appendix B. Role Mapping Syntax

NodeValueCheck:
    query=Query? 'equals' value=STRING;

ListSelectType:
    '.' 'all' '(' type=[ecore::EClass] ')' (selector=Selector)?;

Position:
    '[' index=INT ']' (selector=Selector)?
Appendix C

Role Model Syntax

grammar on.RoleModel with org.eclipse.xtext.common.Terminals
generate roleModel "http://www.onewind.de/language/RoleModel"

DefinitionUnit:
  definitions+=Definition*;

Definition:
  RoleModel | Connection;

Connection:
  'Restriction' name=ID '{' ports+=PortDefinition* '}'

PortDefinition:
  'port' name=ID;

RoleModel:
  'RoleModel' name=ID '{' ('roles' roles+=Role*)?
  ruleSections+=RuleSection* '}'

RuleSection:
  'rules' 'for' connection=[Connection] '{' rules+=Rule* '}'

Role:
  'Role' name=ID comment=STRING?;

Rule:
  selection=ForAll?
  expression=ImpliesExpression
  severity=Severity message=STRING;

enum Operand:
  AND='and' | OR='or' | IMPLIES='implies';

RulePart:
  port=[PortDefinition] '=' //not?='not'
  (multiplicity=Multiplicity distinct?='distinct'?)
  role=[Role];
Cardinality:
```
'card' '(' role+=[Role] (',' role+=[Role])* '->' ports+=[PortDefinition] (',' ports+=[PortDefinition])* ')'
op=RelOp mult=Multiplicity;
```

ForAll:
```
{ForAll}
'for' 'all' expression=ImpliesExpression? ':';
```

enum RelOp:
```
LT="<" | LTE="<=" | GT=">" | GTE=">=" | EQ="==" | UE="<>";
```

ImpliesExpression returns Expression:
```
OrExpression
{ImpliesExpression.left=current}
'implies' right=OrExpression)*;
```

OrExpression returns Expression:
```
AndExpression
{OrExpression.left=current} 'or' right=AndExpression)*;
```

AndExpression returns Expression:
```
LogicalFactor
{AndExpression.left=current} 'and' right=LogicalFactor)*;
```

LogicalFactor returns Expression:
```
Primary | Negation;
```

Negation:
```
'not' primary=Primary;
```

enum Severity:
```
ERROR='ERROR' | WARNING='WARNING' | INFO='INFO';
```

Primary returns Expression:
```
Cardinality | RulePart | '(' ImpliesExpression ')';
```

Multiplicity:
```
IntegerLiteral | AbstractRange;
```

IntegerLiteral:
```
value=INT;
```

AbstractRange:
```
Range | UpperBoundRange;
```

UpperBoundRange:
```
from=INT '...' 's';
```

Range:
```
from=INT '...' to=INT;
```

168


Pop, Adrian. 2008 (June). Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages. Doctoral thesis No 1183, Department of Computer and Information Science, Linköping University, Sweden.


Index

AST, 21, 25, 31–33, 43, 55, 99, 112, 117

Code Completion, 20, 116

Eclipse Modeling Framework (EMF), 18, 19, 21, 23, 31, 55, 95, 96, 99, 109, 115, 116, 141
Ecore, 23, 25, 84, 87, 144, 146
Experiment, 80, 87, 127, 138
  Configuration, 135
  Creation, 123, 130

Index, 22, 112, 114–116, 142

Library, 16, 105, 138, 141

MDSD, 13, 18, 109, 144
Modelica Project, 81, 89, 115, 123, 130, 134
  Creation, 127
  Export, 141

OneModelica, 13, 18, 25, 29, 41, 80, 87, 109

Parser, 20, 21, 55, 110, 111, 135
Physical Model, 17, 19, 79, 84, 93, 95, 123
  Creation, 129
  Reference, 130

Role Model, 51

Simulation Result, 17, 79, 80, 85, 87, 95, 123, 135, 136
Syntax Highlighting, 111

Validation

Code Style, 46
Modelica Language Specification, 29, 32
Modelica Types, 42, 43
Modelica Units, 44, 45
Semantics, 29, 41, 51, 54
Syntax, 111
Types, 28
Dissertations

Linköping Studies in Science and Technology

Linköping Studies in Arts and Science

Linköping Studies in Statistics

Linköpings Studies in Information Science

Linköping Studies in Science and Technology


<table>
<thead>
<tr>
<th>No</th>
<th>Author / Title</th>
<th>ISBN</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>375</td>
<td>Ulf Söderman: Conceptual Modelling of Mode Switching Physical Systems, 1995</td>
<td>91-7871-516-4</td>
<td>95</td>
</tr>
<tr>
<td>383</td>
<td>Andreas Kågedal: Exploiting Groundness in Logic Programs, 1995</td>
<td>91-7871-538-5</td>
<td>95</td>
</tr>
<tr>
<td>396</td>
<td>George Fodor: Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995</td>
<td>91-7871-603-9</td>
<td>95</td>
</tr>
<tr>
<td>413</td>
<td>Mikael Pettersson: Compiling Natural Semantics, 1995</td>
<td>91-7871-641-1</td>
<td>95</td>
</tr>
<tr>
<td>414</td>
<td>Xinli Gu: RT Level Testability Improvement by Testability Analysis and Transformations, 1996</td>
<td>91-7871-654-3</td>
<td>96</td>
</tr>
<tr>
<td>416</td>
<td>Hua Shu: Distributed Default Reasoning, 1996</td>
<td>91-7871-665-9</td>
<td>96</td>
</tr>
<tr>
<td>429</td>
<td>Jaime Villegas: Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996</td>
<td>91-7871-700-0</td>
<td>96</td>
</tr>
<tr>
<td>437</td>
<td>Johan Boye: Directional Types in Logic Programming, 1996</td>
<td>91-7871-725-6</td>
<td>96</td>
</tr>
<tr>
<td>439</td>
<td>Cecilia Sjöberg: Activities, Voices and Arenas: Participatory Design in Practice, 1996</td>
<td>91-7871-728-0</td>
<td>96</td>
</tr>
<tr>
<td>448</td>
<td>Patrick Lambrix: Part-Whole Reasoning in Description Logics, 1996</td>
<td>91-7871-820-1</td>
<td>96</td>
</tr>
<tr>
<td>452</td>
<td>Kjell Orsborn: On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996</td>
<td>91-7871-827-9</td>
<td>96</td>
</tr>
<tr>
<td>461</td>
<td>Lena Strömbläck: User-Defined Constructions in Unification-Based Formalisms, 1997</td>
<td>91-7871-857-0</td>
<td>97</td>
</tr>
<tr>
<td>475</td>
<td>Fredrik Nilsson: Strategi och ekonomisk styrning - En studie av hur ekonomiska styrssystem utformas och används efter företagsförvärv, 1997</td>
<td>91-7871-914-3</td>
<td>97</td>
</tr>
<tr>
<td>480</td>
<td>Mikael Lindvall: An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997</td>
<td>91-7871-927-5</td>
<td>97</td>
</tr>
<tr>
<td>485</td>
<td>Göran Forslund: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997</td>
<td>91-7871-938-0</td>
<td>97</td>
</tr>
<tr>
<td>494</td>
<td>Martin Sköld: Active Database Management Systems for Monitoring and Control, 1997</td>
<td>91-7871-002-7</td>
<td>97</td>
</tr>
<tr>
<td>495</td>
<td>Hans Olssén: Automatic Verification of Petri Nets in a CLP framework, 1997</td>
<td>91-7219-011-6</td>
<td>97</td>
</tr>
<tr>
<td>503</td>
<td>Johan Ringström: Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997</td>
<td>91-7219-045-0</td>
<td>97</td>
</tr>
<tr>
<td>512</td>
<td>Anna Moberg: Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997</td>
<td>91-7219-119-8</td>
<td>97</td>
</tr>
<tr>
<td>530</td>
<td>Henrik Nilsson: Declarative Debugging for Lazy Functional Languages, 1998</td>
<td>91-7219-197-x</td>
<td>98</td>
</tr>
<tr>
<td>561</td>
<td>Ling Lin: Management of I-D Sequence Data - From Discrete to Continuous, 1999</td>
<td>91-7219-402-2</td>
<td>99</td>
</tr>
<tr>
<td>582</td>
<td>Vanja Josifovski: Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999</td>
<td>91-7219-482-0</td>
<td>99</td>
</tr>
<tr>
<td>592</td>
<td>Mikael Ericsson: Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999</td>
<td>91-7219-532-0</td>
<td>99</td>
</tr>
<tr>
<td>593</td>
<td>Lars Karlsson: Actions, Interactions and Narratives, 1999</td>
<td>91-7219-534-7</td>
<td>99</td>
</tr>
<tr>
<td>598</td>
<td>Johan Jenvald: Methods and Tools in Computer-Supported Taskforce Training, 1999</td>
<td>91-7219-547-9</td>
<td>99</td>
</tr>
<tr>
<td>607</td>
<td>Magnus Merkel: Understanding and enhancing translation by parallel text processing, 1999</td>
<td>91-7219-614-9</td>
<td>99</td>
</tr>
<tr>
<td>611</td>
<td>Silvia Coradeschi: Anchoring symbols to sensory data, 1999</td>
<td>91-7219-623-8</td>
<td>99</td>
</tr>
<tr>
<td>618</td>
<td>Jimmy Tjäder: Systemsimplementering i praktiken - En studie av logiker i fyra projekt, 1999</td>
<td>91-7219-657-2</td>
<td>99</td>
</tr>
</tbody>
</table>


**Linköping Studies in Arts and Science**


**Linköping Studies in Statistics**


**Linköping Studies in Information Science**


No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.


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.