Modelica Code Generation with Polymorphic Arrays and Records Used in Wind Turbine Modeling

Roland Samlaus[1] Peter Fritzson[2] Adam Zuga[1] Michael Strobel[1] Claudio Hillmann[1]
Fraunhofer Institute for Wind Energy and Energy System Technology IWES[1]
Linköpings universitet, Dept. of Computer and Information Science[2]

Abstract

At Fraunhofer Institute for Wind Energy and Energy System Technology IWES a simulation software for offshore wind farms is being developed, concentrating on the ability to define physical models at different levels of detail. Therefore parameterizable models representing parts of wind turbines are defined that can be transformed for various purposes like simulation with Finite Element Method (FEM) tools or Modelica solvers.

This paper describes the concepts of purely parametric physical models and code generation. It is elucidated how models of different complexity can be transformed into each other by model driven development techniques. Thereby the focus is set on the generation of Modelica code and it is explained how the use of Modelica libraries simplifies the generation of simulatable code.

During the development of generators for Modelica, issues arose regarding type compatibility of arrays with different sizes when using polymorphism. These issues are explained by an example and possible enhancements for the Modelica language are suggested.

Keywords: model transformation; polymorphism; code generation; wind turbine modeling

1 Introduction

At Fraunhofer Institute for Wind Energy and Energy System Technology IWES a simulation software for offshore wind farms is being developed 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 in 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 [5, 3] 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.

Nowadays many tools are involved in the process of wind turbine design like GH Bladed[1] for load calculations or Focus[2] for rotorblade designs, just to name two of them. Additionally Computational Fluid Dynamics (CFD) tools give a more precise view on aerodynamical 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 to provide 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 Engineer Design Data (EDD) [17]. 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. Figure 1 displays the concept of the EDD with transformations and code generation.

Due to the different domains that the simulation environments are aiming at, there can not 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 rotorblades 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

[1]http://www.gl-garradhassan.com/en/GHBladed.php
[2]http://www.wmc.eu/focus6.php

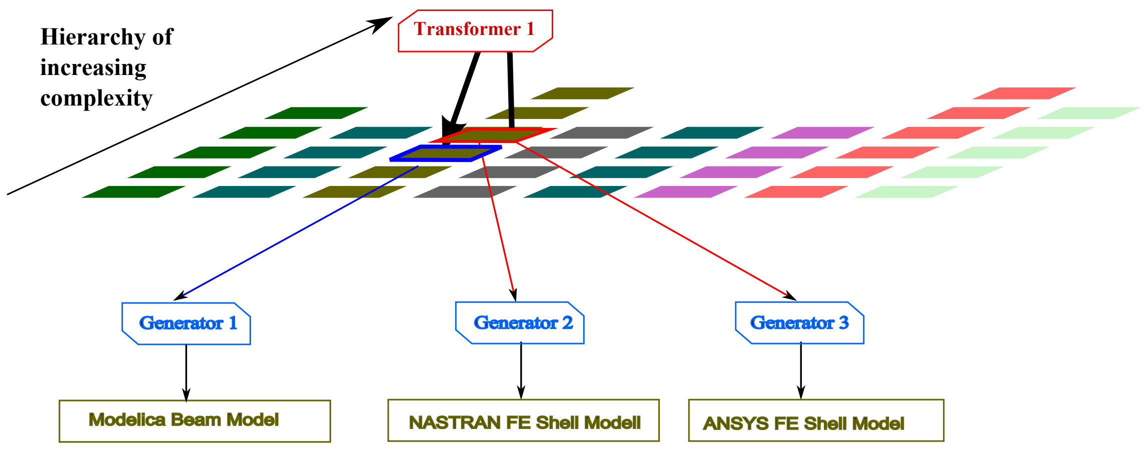


Figure 1: Example of a Transformation Between Two Types of Models and Code Generation for Simulation

composite structure of a rotorblade needs a model with fine grained information about the layer structure to be used for Finite Element Method (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. 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. However, the details of these transformations and their underlying theories are out of scope of this paper. More of interest is the transformation from the EDD representation of wind turbine models to computable Modelica models, as one feature of the OneWind development environment. In the domain of wind turbine modeling with Modelica we can use the EDD 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 behaviour. 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 components of the OneWind Modelica library [18]. The library consists of major components in different complexity levels needed for load calculations of typical offshore wind turbines. 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. 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 4. This has the benefit that developers of Modelica models can re-use the 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.

The remainder of this paper is structured as follows: In section 2, the concept of the EDD is introduced concentrating on a simple wind turbine model. In section 3 the transformation from the EDD to the Modelica representation is explained. Section 4 elucidates the problems that we encountered by transforming user defined parameters to a Modelica array representation. In section 5, we finally come to a conclusion and suggest how polymorphism of the Modelica language could be enhanced by introducing polymorphism in arrays.

2 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. This section introduces how EDD models with different representation types can be defined.

2.1 Abstract Syntax Definition

A meta-model hierarchy called Meta-Object Facility (MOF) [15] for the definition of models is defined by the Object Management Group (OMG) (see Figure 2). The hierarchy is specified as follows: The M0 level describes objects of the real world, as an example it could be an instance of a model of a wind turbine with specific parameters. The model layer (M1) defines, how a real life object can be represented, e.g. by defining a wind turbine model with Modelica. The model consists of components like rotorblades, a tower and a hub that are represented as class or model definitions. In the meta level (M2) the objects that can be used for the development of M1 models are described. For Modelica this implies, that the different language constructs like classes, models, equations, … are defined. Finally the meta-meta level (M3) describes, how the M2 models are defined. We selected Eclipse [2] as the base environment for our products since the Integrated Development Environment (IDE) is open source and can be customized easily by plug-ins that are implemented by software developers. Since the underlying framework is Eclipse, we picked the Eclipse Modeling Framework (EMF) [1] as the meta meta-model for our Modelica language definition as well as the parametric data layer. Using EMF as the M3 layer implementation, language constructs like Modelica classes and equations are defined by EClasses, type references from a component to model declarations can be defined with EReferences and so on. EMF implements a basic version of the MOF, called EMOF. 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. EMF is widely used in the Eclipse community and hence many tools exist that simplify the definition and use of the models. The Framework allows generic processing of model instances making it possible to create functionality for a wide spectrum of diverse models. As an example we implemented a generic editor 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 [13] service in order to provide a convenient way to edit custom data types.

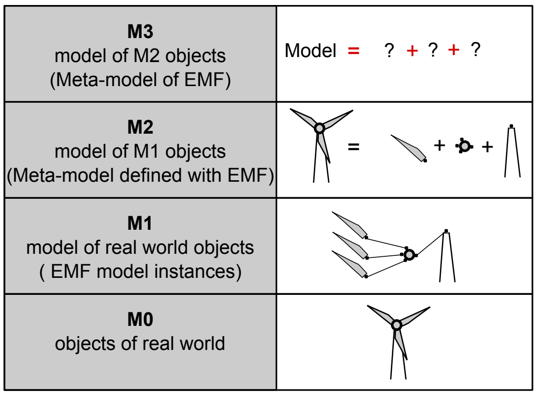


Figure 2: Hierarchy of Meta-models Defined by the OMG

2.2 Concrete Syntax Definition

The abstract syntax of a Domain-Specific Language (DSL)is described by its meta-model, i.e., it 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 Modelica specification [14] 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. In our project the textual DSL Modelica is defined with Xtext [12], allowing us to describe textual representations and to automatically generate EMF-based meta models. Through the grammar definition textual editors that developers can use to define models using the language are automatically generated. The editors recognize syntax errors and the IDE can check additional Well-Formedness Rules (WFR) on the user defined models [16] in order to assist the developer in creating correct code.

A textual representation of DSLs is one way of model representation. A graphical notation may also be defined consisting of icons for structural features. A popular example is the graphical notation language Unified Modeling Language (UML) [6]. It defines icons like rectangular boxes for classes or lines between classes representing associations. When using EMF for the definition of meta-models, it is not necessary to define a graphical representation. The framework provides generic tree-based editors for editing model instances. The standard serialization is based on the XMI file format. However, tools like the Graphical Modeling Framework (GMF) [8] allow the definition of graphical notations similar to the previously mentioned UML for custom meta-models defined with EMF. Hence we have three types of representations for DSLs:

1. Abstract syntax without graphical notation
2. Abstract syntax with textual concrete syntax
3. Abstract syntax with icon based concrete syntax

As we have seen, there are multiple ways of representing 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 we use the first form of DSLs without graphical notation for the parameterizable components of our wind turbine, i.e., for EDD models. Besides the tree based editor that is provided by EMF we implemented a more convenient and extensible editor based on SWT composites as mentioned above. The textual DSLs currently supported are Modelica, the data format ANSYS Parametric Design Language (APDL) [3] and a definition language for airfoils. Xtext grammars were defined for these formats resulting in generated editors, parsers and serializers (also known as unparsers). Since Xtext uses AntLR for the parser generation, support for some formats like NASTRAN [4] bulk data format are hard to implement.

Pure data formats are often structured by terminal symbols like white spaces or line breaks that complicate or even make it impossible to define a LL(k)-grammar [11]. Hence, a custom parser and serializer/unparser could be implemented in the future that would create an EMF compatible Abstract Syntax Tree (AST) from the text files and write the tree representation back to a file. Currently we have no icon based DSL, which is best viewed using an icon-based editor. However, we implemented a connection editor allowing us to connect wind turbine components represented in the purely parametric representation like it is also done in many tools for Modelica models [4, 7].

[3]http://www.apdl.de/

[4]http://www.mscsoftware.com/products/cae-tools/msc-nastran.aspx

In the next section the transformation between different kinds of models is discussed focusing on the generation of Modelica code from wind turbine models.

3 Model Transformation

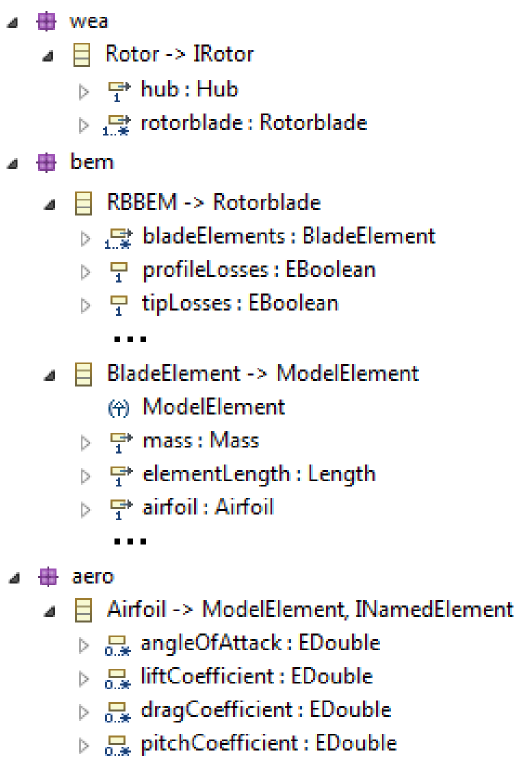
Since model driven software development is increasingly accepted and used by software engineers, transformations of the developed models are becoming important. Various techniques for the transformations have been developed, of which some are described in this section. The transformations to Modelica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

3.1 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, i.e., in both directions, automatic synchronization between two models can be realized. This kind of transformation is called a Triple Graph Grammar (TGG) [10].

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 can not 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



```
wea
  Rotor -> IRotor
    hub : Hub
    rotorblade : Rotorblade
bem
  RBBEM -> Rotorblade
    bladeElements : BladeElement
    profileLosses : EBoolean
    tipLosses : EBoolean
    ...
  BladeElement -> ModelElement
    ModelElement
    mass : Mass
    elementLength : Length
    airfoil : Airfoil
    ...
aero
  Airfoil -> ModelElement, INamedElement
    angleOfAttack : EDouble
    liftCoefficient : EDouble
    dragCoefficient : EDouble
    pitchCoefficient : EDouble
```

Figure 3: EDD-model of a Rotor, Rotor Blade, Blade Element and Airfoil

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 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[5]. 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.

3.2 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 4) mentioned in section 1 for the highly coupled aero-servo-hydro-elastic simulation of wind turbines. The EDD model for a rotor (see Figure 3) 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 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, e.g. 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 components to change the structural properties of the model. One can, for example, decide whether a rigid, modal

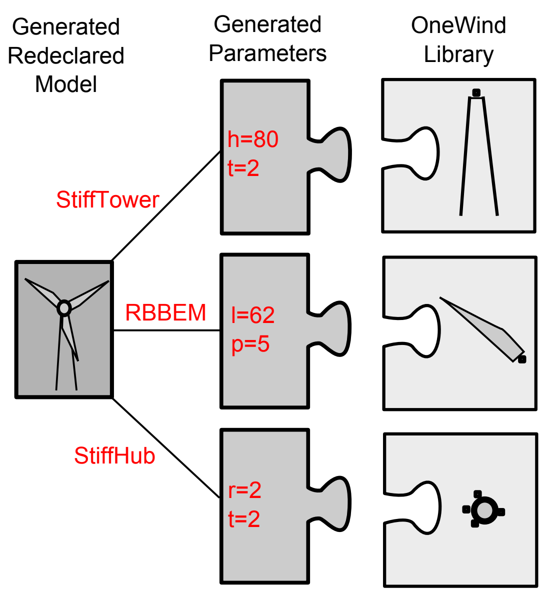


Figure 4: The OneWind Library and Generated Modelica Code

[5] http://www.eclipse.org/m2m/

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.

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

4 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 components 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 Modelica 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* [9] is used. Listing 1 displays the resulting data record of the transformation. A load element contains a parameter profile that holds a list of aerodynamic

Listing 1: Generated Blade Data Record

```
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;
```

profiles (Listing 2) of type `Profile`. The class `Profile` displayed in Listing 3 is a kind of template record. It contains arrays of profile specific data: the angle of attack `alpha[deg]`, lift coefficient `ca(alpha)[-]`, drag coefficient `cw(alpha)[-]` and pitching moment coefficient `cm(alpha)[-]`. The array size is variable as the number of properties varies between different profile types.

Listing 2: Load Element Containing the Airfoil Descriptions

```
model LoadElement
  parameter Profile profile;
end LoadElement;
```

Concrete profile records like `Profile_NACA64` (see Listing 4) define the profile specific value quantity and assign the concrete values to the array. This structure provides a similar behaviour as generic array lists in Java whereby the generic type in this case is defined by the Modelica record `Profile`.

Listing 3: Generic Type Profile

```
record Profile

  import Modelica.SIunits.Conversions.
  NonSIunits.Angle_deg;
  parameter Angle_deg alpha[:];
  parameter Real caOfAlpha[:];
  parameter Real cwOfAlpha[:];
  parameter Real cmOfAlpha[:];

end Profile;
```

For simplification reasons the listing shows only a reduced set of aerodynamical coefficients of the `NACA64` airfoil. Typically, it consist 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 linearisation 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 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, e.g. 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 Modelica specification. Hence the behaviour during simulation is unpredictable or the simulation tool does not even compile the code.

Listing 4: Concrete Profile Record Profile_NACA64

```
//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 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 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 3.2 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.

Listing 5: Generated Blade Model

```
model NREL5MBlade

  BladeData bladeData;
  LoadElement loadElement
     [bladeData.nBladeElements](
     profile = bladeData.profile
  );

end NREL5MBlade;
```

The model from Listing 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 balde from wind inflow) are reused, only the calculation parameters are modified. This approach is used for all main components of the library (rotor, nacelle, tower, substructure, operating control, environment etc.) in order to create a custom model of a wind turbine.

Listing 6: Redeclaration of Blade Element Data

```
model NREL5MBlade
  extends RigidBlade(
    redeclare LoadElement loadElement(
    // old:
    // profile = bladeData.profile
    // new:
      profile = {
        bladeData.cylinder1,
        bladeData.cylinder1,
        bladeData.cylinder2,
        bladeData.du40,
        bladeData.du35,
        // ...
        bladeData.naca64,
        bladeData.naca64}
        )
    );
end NREL5MBlade;
```

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 parameterised classes of the concrete NREL5M model. Listing 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.

Listing 7: Main wind turbine class with redeclared components

```
model NREL5MOffshore
  extends
    HorizontalAxis.OffshoreWindTurbine
  (
    redeclare NREL5MOperatingControl
      operatingControl,
    redeclare NREL5MRotor rotor,
    redeclare NREL5MNacelle nacelle,
    redeclare NREL5MTower tower,
    redeclare NREL5MSubstructure
      substructure,
    redeclare NREL5MSoil soil,
    redeclare NREL5MWind wind,
    redeclare NREL5MWater water
  );
end NREL5MOffshore;
```

5 Conclusion and Outlook

Based on the experience gained during the development of our simulation environment, we can see 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 re-use 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 3 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 Modelica language specification are easier reflected by adapting a few transformation rules than by modifying Java classes.

Increased polymorphism, as discussed in section 4, would enhance the generation as the generated code would be easier to understand and the generation could be encapsulated for each component. This simplifies the code generation and reduces the dependencies between components and their contained declarations.

In order to enable the use of more polymorphism, the Modelica language specification would have to be enhanced, in this case allow polymorphic ragged arrays. 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 components that are used.

To enhance the transformation process we will investigate the use of transformation languages 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 4 would be desirable.

Acknowledgment

This work is 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.

References

[1] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[2] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison Wesley Professional, 3rd. edition, 2009.

[3] International Electrotechnical Commission. Wind turbines - part 3: Design requirements for offshore wind turbines, 2009.

[4] Dynasim AB. *Dymola User Manual: Version 6*. Dassault Systemes, Lund (Sweden), 2009.

[5] Germanischer Lloyd Wind Energie. Guideline for the certification of offshore wind turbines, 2005.

[6] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*.

Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[7] Peter Fritzson, Adrian Pop, Martin Sjölund, Per Östlund, and et. al. Openmodelica useres guide: Version 2012-01-30 for openmodelica 1.8.1. Linköping (Sweden), January 2012.

[8] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[9] J Jonkman, S Butterfield, W Musial, and G Scott. Definition of a 5-mw reference wind turbine for offshore system development. *Contract*, 324(February):75, 2009.

[10] Alexander Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice, Satellite Workshop of Models 2005, Montego*, 2005.

[11] Donald E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971.

[12] Jan Köhnlein and Sven Efftinge. Xtext 2.1 documentation, October 31, 2011.

[13] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley Professional, 1st edition, 2010.

[14] Modelica Association. The Modelica Language Specification version 3.2, 2010.

[15] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[16] Roland Samlaus, Claudio Hillmann, Birgit Demuth, and Martin Krebs. Towards a model driven modelica ide. In *8th International Modelica Conference*, 2011.

[17] M. Strach, F. Vorpahl, C. Hillmann, M. Strobel, and M. Brommundt. Modeling offshore wind turbine substructures using engineer design data — a newly developed approach. In *Proceedings of the Twenty-second (2012) International Offshore and Polar Engineering Conference*, Rhodes, June 2012. International Society of Offshore and Polar Engineers (ISOPE). Accepted.

[18] M. Strobel, R. Vorpahl, C. Hillmann, X. Gu, A. Zuga, and U Wihlfahrt. The onwind modelica library for offshore wind turbines - implementation and first results. In *Proceedings of the Modelica Conference*, 2011.