Static Validation of Modelica Models for Language Compliance and Structural Integrity

Roland Samlaus Mareike Strach

Turbine Simulation, Software Development and Aerodynamics
Fraunhofer Institute for Wind Energy and Energy System Technology, Germany
{roland.samlaus,mareike.strach}@iwes.fraunhofer.de

Abstract

The increasing importance of the simulation of physical systems models demands enhanced support for developers. Models do not only increase in terms of quantity, but also complexity. Hence, libraries need to be created containing valid models for re-use. It is crucial for library developers to get immediate feedback about errors regarding the language specification. Moreover, users of libraries need to know immediately if existing components are misused.

When using Modelica as the modeling language the models are validated at compilation time by recent development environments. This decreases the development speed as developers recognize errors in their models late and therefore need to recapitalize the design decisions made in order to maintain the intent of the code during error fixing.

In this paper we present two implementations, i.e. Object Constraint Language (OCL) and Java, for Modelica code validation that can be triggered during model editing. Both variants are compared to each other regarding readability of constraints as well as execution performance. Therefore, rules are extracted from the Modelica language specification asserting that the models are correct. Furthermore, custom rules are defined restricting library models such that they can only be used in the intended way.

Keywords Modelica model validation, static source code analysis, constraint languages

1. Introduction

In the past years the open modeling language Modelica has become widely used by engineers for physical model development. The benefit of an open language approach is that the user is not dependent on a single tool vendor and can influence the further development of the language standard. The development of the language was accompanied by tool vendors, providing environments accelerating the development process compared to plain text editors. Furthermore, libraries are being developed, open source as well as proprietary, enabling re-use of components for further model design.

At the Fraunhofer Institute for Wind Energy and Energy Systems Technology (IWES), a development environment is being developed, aimed at extensive support for physical model developers. The application of modern model driven technologies allows one to quickly create an Integrated Development Environment (IDE) with advanced tools support for Modelica users. Utilizing the popular Eclipse Modeling Framework (EMF) [1] is advantageous since additional tools built for the Framework can directly be used. As an example, implementations of Object Constraint Language (OCL) interpreters are available, supporting the interpretation of OCL constraints on any language whose meta model is based on EMF.

The development environment is used by engineers at Fraunhofer IWES to create a Modelica library for wind turbines [11]. The demand for immediate validation[1] arose due to the extent of the library. Changes in a model can affect others, but since full validation is only done during simulation, errors are often detected late and therefore the reason may not be obvious any more. Subsequent fixes can contain further semantic errors leading to more design iterations. The errors discussed here are caused by violation of rules defined by the Modelica language specification. Since models are usually composed of various components and the 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 we aim at being independent from any third party tool, this solution

[1] All validations and constraints described in this paper target the correctness regarding the Modelica language specification or Modelica framework design. It is not intended to validate or to constraint physical models.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

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, e.g. from a library, that physically 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 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 — e.g. because it is structurally singular — or to a phyiscally 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 paper describes an approach for static model validation, enforcing the rules defined by the Modelica language specification. Previous work forming the basis for the validation is shown and important parts of the Modelica meta model definition needed to understand the discussed constraints are described. The rules of the Modelica language specification 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 Modelica language specification is explained. Furthermore, the implementation of the constraints in Java is explained and compared to OCL regarding readability, reusability and performance. Further rules regarding establishing structural constraints are discussed and the implementation of a prototype with Java for wind turbine models is presented. Finally, conclusions are drawn and an outlook to future work is given.

2. Related Work

The Object Management Group (OMG) specifies meta modeling by the Meta Object Facility (MOF) [8] that is implemented as the EMF for Eclipse. MOF defines four layers (see Figure 1) of modeling where layer 0 represents the objects of the real world, e.g. physical systems like a wind turbine. Layer 1 (model) contains the models of real world objects, e.g. instances of Modelica classes representing the behavior of wind turbines. In Layer 2 (meta model), the structural properties of instances are described (i.e. concepts like Modelica classes, extend clauses or equations are described). Layer 3 (meta meta model) provides concepts for the definition of layer 2 elements.

When using EMF these concepts can be classes, references or attributes that are used to describe the structure of a Domain-Specific Language (DSL). Constraints are defined based on layer 2 and establish semantic rules, e.g. rules defined by the Modelica language specification. Additionally, rules can be user defined, e.g. to establish code styles or to prevent the combination of incompatible components.

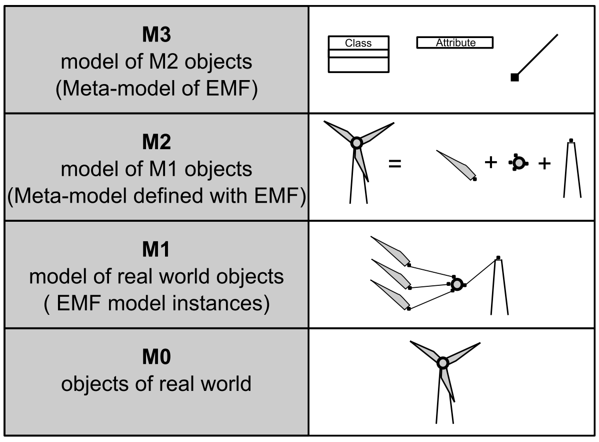


Figure 1. Layers of the Meta Object Facility

OneModelica [9] is a Modelica IDE that is implemented using Model Driven Software Development (MDSD). The validation aims at enriching the IDE and therefore builds on the technologies being provided by the IDE. Since Modelica documents are parsed and represented in a EMF based tree, the constraints can be checked directly on the tree. Thereby, it is possible to either interpret constraints defined in special constraint languages like OCL or to validate instances in the tree directly by general purpose languages like Java. Besides its use as an editor and a validation interface, the IDE provides views to the Modelica developer, helping to understand complex models by viewing information regarding certain aspects in a simplified way. This includes an outline view, displaying the structural content of a document, a documentation view, a hint display for the developer on how to use predefined models, and others. Furthermore, linking is implemented by connecting related documents to each other, e.g. by linking types of components to their respective class declarations. Linking is essential for the validation of Modelica models since many constraints define restrictions on the basis of inherited classes.

RestrictED [10] 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 [4] — 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 [3] 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 [6] 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 [2] proposes a graph-based methodology that provides users with information about under- and overconstraint equation systems. However, the general validity regarding the Modelica language specification is not aimed at.

3. Modelica Meta Model

The Modelica meta model that is the basis for the constraint definitions is described in this section. However, Xtext [5], which is used as the framework for the meta model definition, will not be explained for the sake of brevity. A short introduction describing the Modelica IDE [9] can be found in related work and on the project's web site[2]. The Modelica meta model structure will not be explained in detail, since it is quite complex. However, a simplified representation is displayed in Figure 2, giving an overview of the most important elements needed for the constraint definitions in Section 4.

The root element (`AstModelicaSourceFile`) of the language definition represents a stored Modelica program that can contain arbitrary class declarations and a statement (`within`) declaring in which package these class declarations are contained. Classes can again contain class declarations and other elements like `components`, `extends clauses` and `algorithm sections`.

Classes extending other classes inherit the declared 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*.

`Components` have a type defined by referencing a class declaration. Again, no restrictions are made by the syntax definition of the language. But this needs to be restricted since type compatibility must be enforced, especially when using equations. Here it is important to check whether the types fit in order to be able to do calculations. Furthermore, connect statements connect `components`

[2] http://www.onewind.de/OneModelica.html

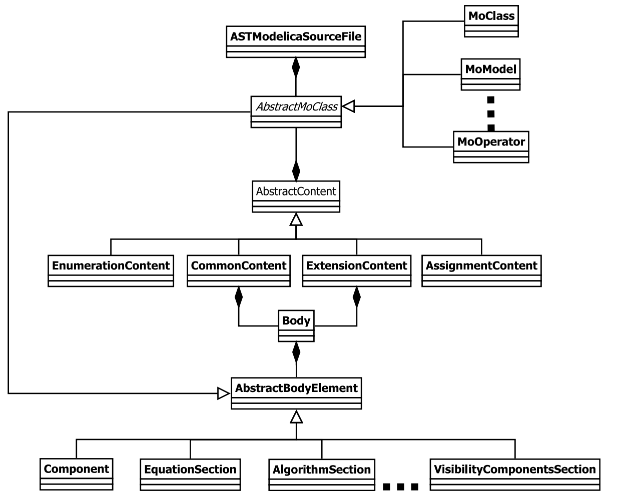


Figure 2. Simplified Modelica meta model

to each other that must be of type `connector` [7]. Components and subclasses are stored in a class `body` that is only present in classes where the content type is `CommonContent` or `ExtensionContent`. Classes with an `EnumerationContent` contain only enumerations while an `AssignmentContent` assigns a present class declaration to a new declaration. This is commonly used when new types are defined in order to make the code easier to understand (the `Modelica.SIunits` package defines new types like `Length` extending `Real` and setting the quantity to *Length* and the unit to *meter*).

The class concept in the grammar is not restrictive. Hence, every class type (class, package, function, record, block, ...) can contain other classes, algorithms and so on. The restrictions are defined in the Modelica language specification 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 Abstract Syntax Tree (AST), it is possible to check the semantic constraints on that representation.

Since the parsed 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 with Java. Xtext provides the possibility of adding methods to elements of the meta model with the language Xtend. The methods that were added mainly implement flattening of classes, i.e. collecting all components defined in a class including its extended 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 4.

4. Validating Modelica Language Specification Compliance

This section describes the static Modelica model validation regarding correctness as defined by the Modelica language specification. Two methods of constraint definitions are compared and implemented with 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 then 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 specification. Type checking is not performed at the moment since the specification is not clear in any points and the effort for the implementation is high as first attempts showed. Nevertheless, type checking should be possible and will be addressed in future work.

4.1 OCL Constraints

OCL was initially designed by the OMG to constrain Unified Modeling Language (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 language specification [7] (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 (42)
- A function can have at most one algorithm section (135)
- A stream connector must have exactly one scalar variable with the flow prefix (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 conditions are met that need to be fulfilled when a conditional aspect is met.

```
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 1. OCL constraint restricting the use of operators

The first constraint (Listing 1) analyzes whether an operator is located in a permitted enclosing class. The language specification 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 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`.

```
context AbstractMoClass
 def: parentIsOperatorRecord(): Boolean =
  not getAbstractMoClass().oclIsUndefined()
  and getAbstractMoClass().oclIsKindOf(MoRecord)
  and getAbstractMoClass()
           .oclAsType(MoRecord).operator
```

Listing 2. OCL helper method for checking whether a class is a operator record

The method `getAbstractMoclass()` is implemented with Java, as described in Section 3, and returns the enclosing class of an AST element. In the case that the operator is defined in 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 2).

The second constraint, as shown in Listing 3, analyses whether a declaration of a `MoFunction` contains at most one `AlgorithmSection`. The syntax definition in the Modelica language specification makes no distinction between the different class concepts. Hence, the definition of several algorithm sections inside a function syntactically conforms to the Modelica specification and thus is not marked as an error by the parser. However, this is semantically incorrect and must be prevented.

```
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. OCL constraint restricting functions to have at most one algorithm section

For validation, the body of the context object is selected. Bodies are only available in type `CommonContent` and `ExtensionContent` (see Section 3). If no body is available (checked by `b.oclIsUndefined()`), the constraint can not 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.

```
context MoConnector
 inv stream_connector_exactly_one_flow:
 let components: Collection(Component)
       = getAllComponents()
 in
   components->exists(cIsStream()) implies
   (components->forAll(cIsFlow() implies
    componentnames->size() = 1)
  and
   components->select(cIsFlow())->size() = 1)
```

Listing 4. OCL constraint checking whether a stream connector has exactly one scalar variable with the flow prefix

The third OCL constraint (Listing 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 5.

```
context Component
def: cIsFlow(): Boolean =
 (not connectorprefix.oclIsUndefined()
  and connectorprefix.isFlow()
def: cIsStream():Boolean =
 (not connectorprefix.oclIsUndefined()
  and connectorprefix.isStream()
context ConnectorPrefix
def: isFlow(): Boolean =
 (not value.oclIsUndefined()) and
  value = 'flow'
def: isStream(): Boolean =
 (not value.oclIsUndefined()) and
  value = 'stream'
```

Listing 5. OCL helper methods checking whether a component is a flow or stream variable

4.2 Java Constraints

To compare the performance of the interpreted language OCL to a general purpose language, the constraints are also defined with Java. The structure of the constraints is somehow comparable. However, it is possible to optimize the performance, since the use of 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 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;
      }
      if (enclosingClass instanceof MoPackage) {
        enclosingClass = enclosingClass.
                  getAbstractMoClass();
        if (enclosingClass == null) {
          return false;
        }
        if (enclosingClass instanceof MoRecord
            && ((MoRecord) enclosingClass).
                isOperator()) {
          return true;
        }
      }
      return false;
    }
    return false;
  }
```

Listing 6. Java constraint checking whether a stream connector has exactly one scalar variable with the flow prefix

In contrast, Listing 7 displays a constraint that benefits from the additional language constructs of Java.

```
  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 7. Optimized Java constraint checking the stream connector restriction

The iteration over components of a class can be interrupted when a violation is detected since the number of scalar components can be checked 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 is `stream` and 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.

4.3 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 constraints targeting the same context object in one document.

On the other hand, Java as the 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.

4.4 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[3], which contains models for various fields of physical modeling. The library contains all sorts of language constructs defined by the language specification 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 Modelica standard library 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

This includes the definitions of base types like `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 done in about 14000 ms. This may be enhanced in the future since the linking mechanism is not optimal regarding performance at the moment. Table 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 we can see 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 3.

The differences in performance originates in the higher efficiency of Java when handling collections. This is particularly clear 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, OCL performs sufficiently. The constraint `function_no_multiple_algorithms` that validates the 1432 functions of the Modelica standard library (defined in Section 4) 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 native 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 Modelica language specification, it is obvious that the performance will not be sufficient for automatic validation during editing with the recent interpreter implementation for OCL included in the Eclipse MDT project. Even for

[3] https://modelica.org/libraries/Modelica

Constraint	OCL Calls	Time (ms)	JAVA Calls	Time (ms)
unique_element_names_comp	5039	9896	5039	140
protected_variables_dot_reference	225240	3842	225240	731
prefixes_structured_component_flow	22186	97	22186	3
function_no_multiple_algorithms	**1432**	**77**	**1432**	**15**
flow_subtype_of_real	22186	76	22186	47
stream_only_in_connector	22186	28	22186	15
stream_connector_exactly_one_flow	**108**	**2**	**108**	**0**
function_no_equations	1446	1	1446	0
nested_when_equations	35	0	35	0
operator_only_in_record_or_package	**3**	**0**	**3**	**0**
...				
	94677	**20063**	946774	3330

Table 1. OCL and Java validation performance of selected constraints

Java, if the user does not want to be disturbed by long lasting validations taking place while editing Modelica models, focus on high performance is needed during further implementations of the missing constraints.

5. 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 [11] 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, e.g. by providing OCLconstraints 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.

5.1 Possible Model Structures for a Horizontal-Axis Wind Turbine

Using the OneWind Modelica libary, 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 3.

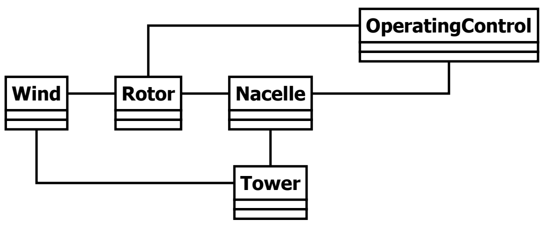


Figure 3. Model structure of a horizontal axis wind turbine using OneWindModelica library components

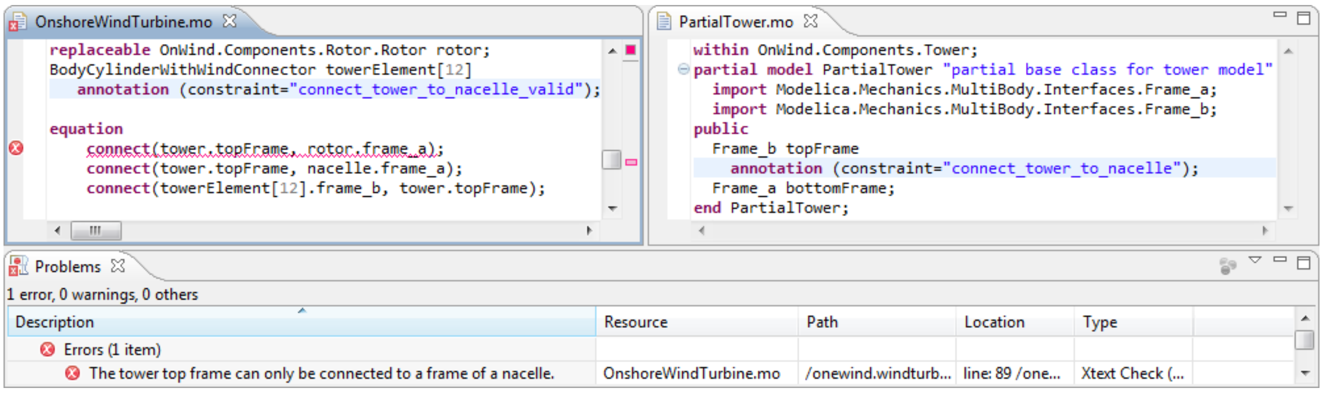
As can be seen in Figure 3, the `Rotor` object is connected to the `Nacelle` object, which is then connected to the `Tower` object. However, a user of the library could connect the `Rotor` object to the `Tower` object directly. In both objects `connector` instances are used which makes this connection correct according to the Modelica language specification. From an engineering perspective, however, this is not reasonable. Furthermore, the components of the OneWind Modelica libary 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 on the true source of the error.

5.2 Introduction of Structural Constraints

To support the library user during the model development, a constraint that gives a warning can be defined to avoid the direct connection of `Rotor` and `Tower` object. This is realized utilizing 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 GUI of OneModelica is shown in Figure 4. 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 4. If the user tries to connect the `Tower` object to a `Rotor` object as it is shown in the upper left side of Figure 4, an error message will appear (cf. bottom part of Figure 4). The connection of the `Tower` object to the `Nacelle` object does not result in this error.



```
replaceable OnWind.Components.Rotor.Rotor rotor;
BodyCylinderWithWindConnector towerElement[12]
   annotation (constraint="connect_tower_to_nacelle_valid");

equation
    connect(tower.topFrame, rotor.frame_a);
    connect(tower.topFrame, nacelle.frame_a);
    connect(towerElement[12].frame_b, tower.topFrame);
```

```
within OnWind.Components.Tower;
partial model PartialTower "partial base class for tower model"
  import Modelica.Mechanics.MultiBody.Interfaces.Frame_a;
  import Modelica.Mechanics.MultiBody.Interfaces.Frame_b;
public
  Frame_b topFrame
    annotation (constraint="connect_tower_to_nacelle");
  Frame_a bottomFrame;
end PartialTower;
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
The tower top frame can only be connected to a frame of a nacelle.	OnshoreWindTurbine.mo	/onewind.windturb...	line: 89 /one...	Xtext Check (...

Figure 4. Structural constraint for connection of `Tower` and `Nacelle` object

However, if the user decides that he would like to connect an object to the `Tower` object that is not of type `Nacelle`, he can avoid the error message by defining another annotation. This is shown also in the upper left part of Figure 4. In this way, the user is not prohibited from using the library components to his wishes. But the constraint helps especially unexperienced users to avoid mistakes which is of high importance.

For this kind of constraints following steps needed to be performed. For the context type `ConnectClause` a Java validator has been registered. The validator then checks whether one of the connected elements defines a constraints inside its annotation (*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 (*constraints="constraintname_valid"*). If the constraint does not exist, an error marker is created.

6. Conclusion and Future Work

This work shows that 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 Java 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 enhance the development of Modelica models heavily, since errors can be immediately displayed to the user.

However, we are able to point 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 [12]. This would however outweigh the benefit of the interpreted language that constraints can be defined and checked during runtime.

Beyond validation against the Modelica language specification, 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 Modelica language specification. 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 language specification that are accepted by some simulators can be identified easily.

The access to the AST maybe further enhanced by adding additional methods. Furthermore the introduction of caching, e.g. by remembering all base types of a class, which is necessary for fast type checking, would accelerate the validation with a reasonably cost of memory. This however would further reduce the complexity of constraint definition and may enable the use of OCL which seems to be more promising if e.g. library providers shall define constraints on how their models can be used.

The idea of further structural restrictions is promising and can enhance the definition of Modelica libraries. Besides the correct use of components, constraints may also be used to identify possible combinations of components, since the selection of usable items is reduced. This may help the user to find suitable solutions more efficiently by selecting components that are suggested by the IDE. Constraints may be defined for specific design aspects. Additionally, rules for connections may need to be expressed, e.g. regarding the cardinality of connections. An implementation based on role models is currently developed and will simplify the definition of structural constraints in the future.

References

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

[2] Peter Bunus and Peter Fritzson. Automated static analysis of equation-based components. *Simulation*, 80(7-8):321–345, 2004.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 edition, 1995. 37. Reprint (2009).

[4] Christoph Höger. Modim - a modelica frontend with static analysis. In *MATHMOD 2012 - 7th Vienna International Conference on Mathematical Modelling*, 2012.

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

[6] Malte Lochau and Henning Günther. A static aspect language for modelica models. In Peter Fritzson, FranÃ§ois E. Cellier, and David Broman, editors, *EOOLT*, volume 29 of *Linköping Electronic Conference Proceedings*, pages 47–57. Linköping University Electronic Press, 2008.

[7] Modelica Association. Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.3, 2012.

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

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

[10] Mirko Seifert and Roland Samlaus. Static Source Code Analysis using OCL. In Jordi Cabot and Pieter Van Gorp, editors, *OCL'08*, 2008.

[11] 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.

[12] Claas Wilke. Java code generation for dresden ocl2 for eclipse. Großer beleg (minor thesis), Technische Universität Dresden, Dresden, Germany, February 2009.