Contents

Introduction
  Motivation

Simplifying the access to the AST data

Validating Modelica models
What we have got

- Modelica editor based on Xtext
- Ecore-based abstract syntax tree (AST)
- Automatic syntax validation based on the Modelica grammar
- Linking of variables and types and error markers for unresolved links
What we need

- Immediate validation of semantic rules that are defined by the Modelica language specification
- Type checking
- Custom rules, e.g. style check
- Further semantic rules for the restriction of model use (is it allowed to connect two components to each other?)
- Independence from third party tools
- The validation of models needs to be fast
Contents

Introduction

Simplifying the access to the AST data
  Adding additional methods and information to the AST

Validating Modelica models
Problems that occur when validating the AST

- The abstract syntax tree of a document can be very large
- Often no direct access to the objects of interest:
  - Get all components inside a class
  - Get all protected components
- Access to objects of extended classes
Adding methods to the AST for queries

- Implementing methods for the AST access with the validation language can lead to duplication of code and code that is hard to understand (as experienced with OCL)
- The language Xtend allows to add references, and operations to the meta model generated by Xtext
- Operations were added to efficiently query the parsed models for certain objects
- E.g. for all kinds of classes: `getComponents()`, `getAllComponents()`, `getAllSubClasses()`, `getAllExtendedClasses()`, `getModelicaType()`, ...

Modelica Types

- The method `getModelicaType()` is used to check whether a class extends a basic data type (`Real`, `Integer`, `Boolean`, `String`, `Enumeration`, `Unresolved`, `NoType`)
- Basic data types are modeled with Ecore and thus can directly be added to nodes of the Modelica AST
- The basic data types are checked in expressions
- Unresolved types are used for references that cannot be resolved
- `NoType` represents classes that do not extend from a basic data type
Contents

Introduction

Simplifying the access to the AST data

Validating Modelica models
  Language compliance
  The validation languages
  The validation languages
  Custom rules
Rules found in the Modelica language specification

- About 200 semantic rules could be found in the specification
- The rules vary in complexity
- Type checking for expressions has been implemented for basic types
- More rules can now be implemented, since types were introduced
Application of pure Java and OCL

- 44 rules were implemented with OCL and Java
- The performance has been measured by validating the Modelica standard library
- Compared to previous implementations the performance and extent of the OCL constraints could be enhanced significantly since queries are implemented efficiently with Java
Example of an OCL constraint

Operators may only be placed in an operator record or in a package inside an operator record (42)

```
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())
```
Example of a Java constraint

```java
public boolean isValid(EObject eObject) {
  MoOperator op = (MoOperator) eObject;
  AbstractMoClass clazz = op.getAbstractMoClass();
  if (clazz == null)
    return false;
  if (clazz instanceof MoRecord
      && ((MoRecord) clazz).isOperator())
    return true;
  if (clazz instanceof MoPackage) {
    AbstractMoClass parent = clazz.getAbstractMoClass();
    if (parent == null)
      return false;
    if (parent instanceof MoRecord
        && ((MoRecord) parent).isOperator())
      return true;
  }
  return false;
}
```
Comparing the performance

Overview of the validation performance:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>OCL Calls</th>
<th>Time (ms)</th>
<th>JAVA Calls</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique_element_names_comp</td>
<td>5039</td>
<td>9896</td>
<td>5039</td>
<td>140</td>
</tr>
<tr>
<td>protected_variables_dot_reference</td>
<td>225240</td>
<td>3842</td>
<td>225240</td>
<td>731</td>
</tr>
<tr>
<td>prefixes_structured_component_flow</td>
<td>22186</td>
<td>97</td>
<td>22186</td>
<td>3</td>
</tr>
<tr>
<td>function_no_multiple_algorithms</td>
<td>1446</td>
<td>77</td>
<td>1446</td>
<td>15</td>
</tr>
<tr>
<td>flow_subtype_of_real</td>
<td>22186</td>
<td>76</td>
<td>22186</td>
<td>47</td>
</tr>
<tr>
<td>stream_only_in_connector</td>
<td>22186</td>
<td>28</td>
<td>22186</td>
<td>15</td>
</tr>
<tr>
<td>stream_connector_exactly_one_flow</td>
<td>108</td>
<td>2</td>
<td>108</td>
<td>0</td>
</tr>
<tr>
<td>function_no_equations</td>
<td>1446</td>
<td>1</td>
<td>1446</td>
<td>0</td>
</tr>
<tr>
<td>nested_when_equations</td>
<td>35</td>
<td>0</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>operator_only_in_record_or_package</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>946774</td>
<td>20063</td>
<td>946774</td>
<td>3330</td>
</tr>
</tbody>
</table>
Comparing the validation languages

**OCL**
- Dedicated language for validation
- Easy to understand if compact
- Hard to understand when used on large data structures
- Slow (especially when processing collections)

**Java**
- General purpose language known by many developers
- The purpose of a constraint is hard to find out
- Fast (compared to OCL)
Code checks

- Both language can be used to create code style restrictions
  - Maximum number of classes inside package
  - Mandatory comment checks
  - ...

- Check whether connected models structurally fit together
Structural validity

- Structural validity can be checked in order to add additional semantics to models
- Library models can be annotated to restrict their use
- The connector instances of a model are annotated and connect clauses are validated
Validation

Demo
Conclusion and Future Work

- Validation is possible and can perform well
- It is possible to create rules to enforce structural validity
- Implementation of additional restrictions
- Code style checks
- Rules for structural validity of the OneWind library
- Custom OCL constraints inside annotations for library designers?
Thank you very much!

You can download OneModelica: http://www.onewind.de