Towards a model driven Modelica IDE

Roland Samlaus[1] *ClaudioHillmann*[1] *BirgitDemuth*[2] *MartinKrebs*[2]
Fraunhofer Institute for Wind Energy and Energy System Technology[1]
Technische Universität Dresden, Institut für Software- und Multimediatechnik[2]

Abstract

Model Driven Software Development evolved into a common way of creating software products. Describing software in a more abstract way simplifies and speeds up the development process and generated code turns out to fulfill high quality standards. As a subcategory of model driven development Domain-Specific Languages concede to express problems in a domain specific way. By defining a languages grammar, an editor that provides basic support for developers can be generated automatically. This paper describes how these concepts are utilized for the creation of a Modelica Integrated Development Environment (IDE). Helpful functionality is implemented in a model driven way to maximize assistance during the development process. Thus the developer receives a tool that allows to survey large scale projects and provides functionality that is well known in other popular programming languages. Furthermore an approach for semantical verification of Modelica documents during the development process is presented. This allows to detect and correct errors early.

Keywords: Modelica, IDE, OCL, verification

1 Introduction

In the last years the importance of Modelica in the field of engineering increased significantly. Many companies utilize the language for modeling and simulation of physical systems. Thus the support for developers became a vital issue to enable the survey of extensive projects. One approach is the development of libraries as done by the Modelica community. Libraries provide common functionality that can be easily reused and extended and thereby increase the speed of development and ensure high quality of the resulting models.

An additional approach is the usage of development tools. These support the engineer during the implementation of big physical systems (e.g. Dymola[1] and OpenModelica[2]). Although the above mentioned tools turned out to be very helpful, supplementary features are desired to ease the handling of complex models. This topic is well known in the software engineering community as well. Therefore we aim at transferring main features to the Modelica world. The structure of our Modelica Integrated Development Environment (IDE) and how it was created by model driven technologies is explained in section 2.

Besides the goal to facilitate daily work for engineers with Modelica, the quality of the outcome has to be addressed. Regarding this purpose the developer has to be encouraged to create syntactically and semantically correct documents. While the syntax can be checked easily, semantic correctness may be hard to prove because semantic constraints may demand extensive calculations. This issue is addressed in section 3 and our approach of utilizing Object Constraint Language (OCL) for verification of Modelica models is stated.

In section 4 some components of the IDE e.g. the editor and views are presented. Finally a summary and future work will conclude this paper in chapter 5.

2 The MDSD approach

Model Driven Software Development (MDSD) is an approach that eases the development process by providing a higher abstraction level of the software that is being developed. Compared to pure code-based development, the architecture can be described in a more conceptual and structured way. A common way is to design the structure and behavior of software with the help of Unified Modeling Language (UML) diagrams. But graphical representation is not the only popular way of an abstract description. In the MDSD community the usage of Domain-Specific Language (DSL)s became more and more important.

DSLs allow the definition of problems in a domain specific way and therefore provide an easy way to ex-

[1]http://www.dymola.com

[2]http://www.openmodelica.org

press ideas to the domain experts. This can be done graphically as well as in a textual syntax. The Modelica language can also be seen as a textual DSL, since it allows the developer to describe physical systems.

As Eclipse builds the basis for our IDE it is evident to use Eclipse Modeling Framework (EMF) as platform for our models. All data being processed by EMF are based upon ECore that is more or less aligned on Object Management Group (OMG)s Essential Meta Object Facility (EMOF) [3] standard. All generated objects extend a basic interface that allows interoperability between objects of different meta-models. More details about the functionality of Ecore can be found on the EMF website [4].

The project structure of the IDE has been defined with the help of EMF tools. Several ways [8] of describing an Ecore-based meta model are available, like Java annotations or XML, whereas the usage of a UML-like graphical editor might be the most appealing way for developers. The Modelica DSL was created with Xtext[5], an Open Source framework for the development of DSLs. It also uses the functionality provided by EMF and therefore enables to easily interact with our data models as described in Section 2.2.

In the next sections the creation of the Modelica DSL is described and the underlying data structure explained. Figure 1 gives an overview of the used tools.

2.1 Metamodeling of Modelica with Xtext

For the textual syntax definition of Modelica we use the tool Xtext, that is part of the Eclipse Modeling Project. Xtext's syntax for defining language grammars closely resembles the Extended Backus-Naur Form (EBNF) notation. Based on this grammar several components are generated automatically. 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 [3][10] creates an Abstract Syntax Tree (AST) and a Concrete Syntax Tree (CST) of the given text document. Thereby we get an editable tree-like data structure that can be processed by additional software components. The AST represents the structure of a Modelica document. Additionally the CST keeps all information about the concrete representation inside the document e.g. literals and white spaces. Based on the grammar, syntax highlighting and basic code completion for the generated editor is provided. The parser recognizes syntactical errors and displays them inside the editor and in a separate problems view. The view contains detailed descriptions of the errors and allows to jump into the erroneous area of the document.

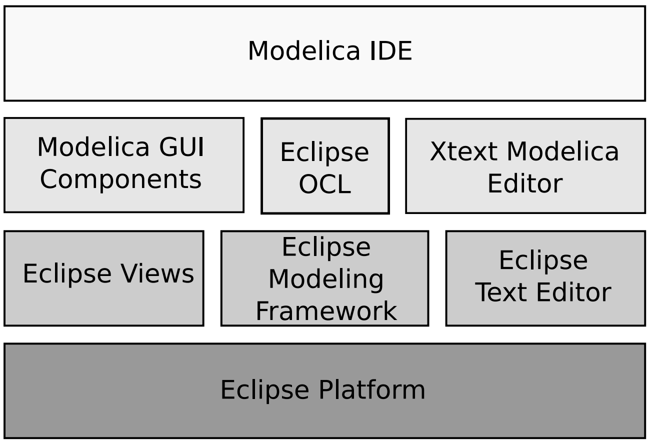


Figure 1: Overview of tools used in the Modelica IDE

Additional to these basic tools, other components are generated that use the AST, for example an outline view that represents the inner structure of opened documents, i.e. inner classes, sub-packages and component declarations. One of the main amenities of Xtext is the integrated resolution of references. 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 is again 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 created, e.g. the proposal of classes that can be referenced at a certain position inside the document.

The usage of Xtext for grammar definitions is covered below by a small example. Listing 1 depicts Modelica's grammar definition for the element `ImportClause`.

An import clause starts with the keyword `''import''` followed by an optional abbreviation definition. Optional rules are defined by a question mark. In many Modelica documents `SIunits` are imported with the definition of an abbreviation: `''SI = Modelica.SIunits;''`. This allows to write shorter expressions when using type definitions from the `SIunits` package.

[3]http://www.omg.org/
[4]http://www.eclipse.org/modeling/emf/
[5]http://www.eclipse.org/Xtext/

```
"import"  (abbrev=Name "=")?
reference=[AbstractContent]
wildcard?=".*"? comment=Comment?;
```

Listing 1: Import clause definition in Xtext

The next rule defines a reference to an AbstractContent. AbstractContent is another grammar rule that represents different kinds of Modelica class contents, e.g. the standard class structure or an extends clause. For more details on the Modelica class structure see [9]. In Xtext square brackets define references to other objects. This is a Xtext specific feature that is not defined in EBNF. A generic linking mechanism is provided that resolves the reference by searching for an AbstractContent whose name attribute of type String matches the given input.

Import clauses are not restricted to import single Modelica classes but also a set of sub-classes inside a package. This is indicated by the use of a wild card "*". Using a reference to AbstractContent does not respect the fact that only packages can be referred to when using wild cards. In this definition we do not distinguish between Modelica classes, models, packages and so forth on the syntax level. Therefore the correctness of the rule has to be regarded by the semantics of Modelica. This is surveyed in section 3. Finally an optional comment adds additional information about the import for developers.

Altogether the grammar definition consists of over 100 rules. Performance issues during parsing prevented us from reusing the language definition from the Modelica language specification [9], so that we were forced to create an optimized version. Moreover, the complexity of the resulting AST would have made it difficult to modify or investigate the parse results. As an example, the definition of expressions has been simplified. We do not distinguish between logical expressions, terms, or factors but only define a single type of expression. If needed, the type can be derived by investigating the operator of an expression. This enabled us to reduce the number of rules to 4 compared to 12 defined in the Modelica language specification [9].

2.2 Project structure definition with EMF

For the sake of reuse, Modelica documents should be structured in projects. When defining e.g. a wind energy plant, every component like Tower, Nacelle or RotorBlade ought to 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.

Separating components into projects requires a mechanism that enables linking between models inside separate projects. A WindTurbine e.g. reuses the component Tower for the definition of a new wind turbine. Therefore a link between the wind turbine's definition and the document where the tower is defined in has to be established. This ensures the existence of the reused component and enables the user to quickly display the tower's definition. In order to be able to find this kind of references quickly, meta data must be provided that holds additional information about the class structure and location of Modelica files. This data structure is defined with EMF. 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. When a source file is parsed, the internal structure is analyzed and stored in an index file. These files are again coupled to the ModelResource. Hence ModelResources know the name space of all contained models and allow quick linking by the use of qualified names. Qualified names distinctly address a component inside a name space like in Modelica.SIunits.Angle whereas Modelica and SIunits represent packages and Angle is a type definition inside this package.

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 our requirements (see figure 2). Using ModelResources instead of projects as management unit 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, every 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 are kept inside the archive, therefore 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; Secondly, using libraries speeds up the development process because handling of a huge number of source files can lead to a slow development environment and an 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. 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.

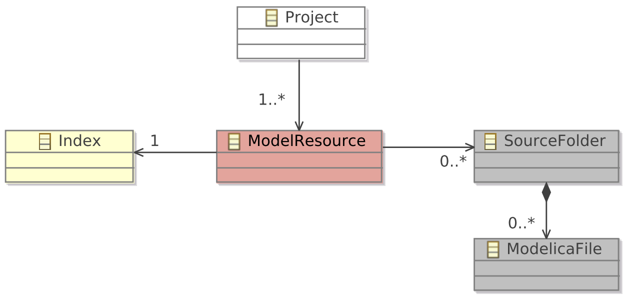


Figure 2: Data structure of the Modelica IDE

3 Verification of Modelica documents

Enhanced verification of code is desired during development to immediately ensure the correctness of the created models. Two kinds of verification can be distinguished - dynamic and static verification.

3.1 Dynamic versus static verification

Dynamic verification of Modelica models is, at the moment, a difficult topic because for this issue models have to be interpreted on instances. For instance, to ensure that 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++ and execution of the resulting program code. That means no interpreter working directly on Modelica source code is available but the code is transformed to another kind of programming language and then executed.

The same problem is faced when trying to debug Modelica code like it is done in other object oriented languages, e.g. Java. It may be possible to implement an interpreter for basic Modelica language constructs and simple models that do not contain any equations. But at the moment there is no solution available that solves Differential Algebraic Equations (DAE)'s during development time.

The MDSD-community is, by the way, facing the same problem. Instead of generating code that has to be executed, interpreters are often desired that create functionality based on objects. Therefore solving the problem of interpreting models directly could solve the problem of debugging and verification as well as reduce the required time during development.

However static verification is currently done for a wide range of models (e.g. Modelica, UML, Java, . . .). Several techniques are available for the verification of models. One could write Java-Code or utilize specialized languages like Check, that is delivered with the Xtext-Framework. In our IDE we use OCL [2] which is a standard language of the OMG. It is spread in research and industry and thus is typically to be understood by many developers. The static verification of Modelica with OCL is presented in the next section.

3.2 Static verification with OCL

In the Modelica specification the semantics of the language is verbally specified. We interpreted the Modelica semantics as Well-Formedness Rules (WFR)s and found 201 WFRs which we translated into OCL constraints. The WFRs differ from each other in terms of complexity. Some constraints are quite easy to define and the execution time is short. Others are complex and often recursive. The complexity and recursivity is conditional upon the underlying Modelica metamodel and OCL as a language that specifies navigation paths through a model. This can cause the verification to take a long time because large parts of the AST have to be considered during the interpretation of the constraints. In the following OCL will be shortly introduced. Then the definition of some WFRs is explained.

3.2.1 OCL

OCL is a language that allows the definition of constraints on objects. Originally it was designed to enable more precise UML diagram definitions. Later OCL has been extended to a query language and is generally in metamodeling. Figure 3 displays a simple example of an UML class where the attribute `age` inside the class `Person` is constrained. The invariant ensures that the age is higher than 17 to fulfill the requirement of being an adult.

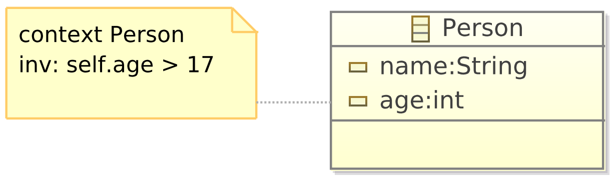


Figure 3: A simple OCL example

Besides invariants OCL also allows to define queries to collect objects. Furthermore the definition of new or derived model elements (attributes, associations, operations) is allowed and helpful to reuse common functionality. OCL also defines a standard library that provides basic functionality like operations on collections (e.g. checking if a set is empty). The language description and examples can be found in the OCL specification [2].

3.2.2 Modelica and OCL

As mentioned above, OCL is not restricted to be used for UML diagrams but can principally be applied to arbitrary object structures [5]. The only restriction is, that the interpreters need to be able to handle the constrained model. This usually means that the meta model of the restricted language must be available and the interpreter must be able to read the data format. As we use Xtext for the definition of our Modelica language, the resulting AST is based on Ecore. Hence an Ecore based OCL interpreter is needed. Therefore two popular OCL interpreters that fulfill this requirement were evaluated for verification, Eclipse OCL[6] and Dresden OCL[7] [13].

The basis for the verification of Modelica code is the AST of a Modelica document. It is created by the parser that Xtext generates based on the grammar definition. Thus the AST represents the structure of the Modelica document and can be used to check whether the structure is correct. OCL constraints are defined and evaluated on the AST.

The OCL constraints were used to measure the performance of both tools mentioned above. Furthermore the constraints were analyzed to find time consuming rules. This is important when dealing with user interfaces because users do not accept lags when editing documents because of verification tasks that are performed in the background. Two categories with different complexity and calculation times were detected (Table 1).

[6]http://www.eclipse.org/modeling/mdt/?project=ocl

[7]http://www.reuseware.org/index.php/DresdenOCL

	non-recursive constraints	recursive constraints
constant time	x	-
linear time	x	x
quadratic time	x	x
exponential time	-	x

Table 1: Classification of OCL constraints by their complexity

Short examples for non-recursive constraints will illustrate the definition of Modelica WFR in OCL. Listing 2 represents a constraint that can be evaluated in constant time:

```
inv predefined_string_type:
name <> 'String'
```

Listing 2: Non-recursive OCL with constant time constraint

Each component declaration in Modelica has a name. In our grammar this is reflected as a rule `ComponentName` with the attribute `name`. Because Modelica reserves some names for components (i.e. String), these names are not allowed for newly defined components. The OCL invariant given in Listing 2 ensures that the name `String` is not assigned to a component. As no other objects have to be considered, the calculation depends only on the object `ComponentName` resulting in a constant calculation time.

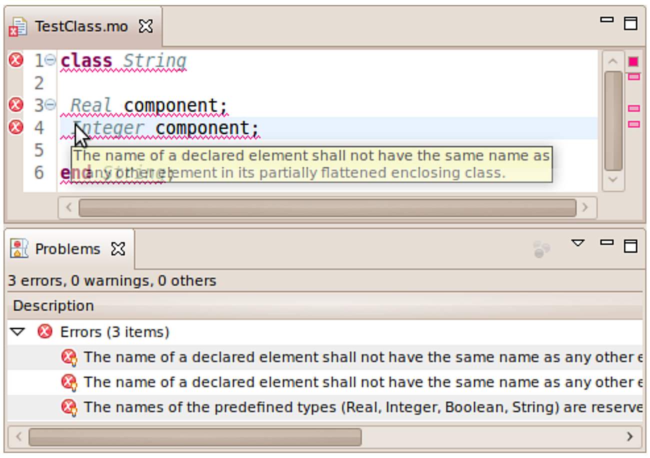
A linear non-recursive constraint is defined in Listing 3:

```
context Component
inv component_name_type:
not componentnames->exists(
name = self.type.name)
```

Listing 3: Non-recursive OCL constraint with linear time

Figure 4 shows the result of our validation in the editor and the problems view.

Components in Modelica classes must not have the same name as their type. It is e.g. prohibited to define a component `Angle Angle;`. The constraint in Listing 3 checks whether a name exists (`componentnames->exists()`) that is equal the name of the components type (`self.type.name`). The calculation time coheres directly with the num-



```
class String

  Real component;
  Integer component;

end String
```

The name of a declared element shall not have the same name as any other element in its partially flattened enclosing class.

3 errors, 0 warnings, 0 others

Description

Errors (3 items)

- The name of a declared element shall not have the same name as any other e
- The name of a declared element shall not have the same name as any other e
- The names of the predefined types (Real, Integer, Boolean, String) are reserve

Figure 4: Displaying verification errors

ber of instances that are defined in the component and thus increases linearly.

Another problem is the test for uniqueness which results in quadratic calculation time. If uniqueness of element names in an enumeration shall be ascertained, all elements have to be compared with each other (as long as the model elements are not stored in a relational database). Thus the calculation time grows quadratically to the number of elements. The corresponding OCL constraint is defined in Listing 4:

```
context EnumerationList
inv unique_enum_literals:
enumerationliterals ->isUnique(
componentname.name)
```

Listing 4: Non-recursive OCL constraint with quadratic time

Most of the OCL constraints defined for the verification of Modelica documents are of recursive nature because most WFRs originate in restrictions on inheritance structures. As a result many parts of the model have to be considered for verification. Although the effort of calculating non-recursive rules may result in quadratic time as seen above, recursive rules are even worse.

For complexity reasons, only one recursive constraint with exponential time is explained in this paper (Listing 5). The Modelica specification defines the rule: “The type prefixes `flow`, `input`, and `output` shall only be applied for a structured component, if no elements of the component have a corresponding type prefix of the same category.” [9] The function definition in Listing 5 returns a Boolean value that indicates, whether a Modelica class contains a component definition with the prefix `flow`, `input`, or `output` (the collection of components is defined in another function `collectIOComponents()`). Not only the analyzed class has to be considered for this constraint, but all super classes from which the instance inherits. This requires the invoking of the same function `containsIOPrefixes()` recursively and results in the complexity $\mathcal{O}(n^r)$ where `n` describes the number of super classes and `r` the recursion depth.

```
context AbstractModelicaClass
 def: containsIOPrefixes() Boolean =
  collectIOComponents()->size >0
   or
  collectExtendsClauses()->exists(
  containsIOPrefixes)
```

Listing 5: Recursive OCL constraint with exponential time

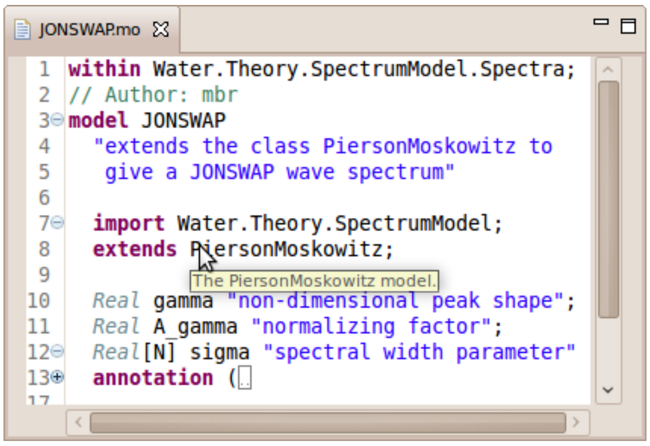
Because of the complexity of the resulting constraints not all WFRs from the Modelica specification have been implemented yet. In a first version of the Modelica IDE (Section 4 we decided to integrate Eclipse OCL because of its better interpreter performance. In addition to the rules from the specification, further constraints may be helpful for daily work. E.g., warnings could be displayed if to many subclasses in a document exist or the package structure is too deep. This functionality may be integrated in a later version of our IDE.

4 Modelica IDE

In this section the main parts of our IDE, which supports the developer in the creation and manipulation of models are presented. First the features of the generated and enhanced Modelica editor are presented (Section 4.1), then it is explained how additional views simplify the development process (Section 4.2).

4.1 Editor

Modern Modelica IDEs support the user in the development process by providing editors and related tools that ease the handling of big projects. Editors provide syntax highlighting to emphasize language specific keywords and to reveal the structure of the written code, making it easier for the developer to understand the code. Highlighting and the recognition of syntactical errors is provided by Modelica specific lexers and parsers that can either be hand written or generated by tools like ANTLR as described in Section 2.1. Furthermore semantical highlighting may be provided but should only be checked where the calculation can be done quickly. In Figure 5 the simple data type `Real` (gray, italic) is highlighted semantically while all other decorations are provided by the generated lexer. Code folding allows to reduce the complexity of the displayed code, e.g. by hiding annotations that contain arbitrary information. With the integrated mouse-over help, details about utilized classes that are defined as comments in the declaration can be explored by the developer.



```
within Water.Theory.SpectrumModel.Spectra;
// Author: mbr
model JONSWAP
  "extends the class PiersonMoskowitz to
   give a JONSWAP wave spectrum"

  import Water.Theory.SpectrumModel;
  extends PiersonMoskowitz;

  Real gamma "non-dimensional peak shape";
  Real A_gamma "normalizing factor";
  Real[N] sigma "spectral width parameter"
  annotation (
```

Figure 5: Xtext Modelica editor

Based on the index data, code completion supports the developer in choosing sub-components of classes. This is helpful during the definition of import clauses or components and helps to speed up the development process significantly.

Error markers for non-existent referenced classes are automatically created. This is an advantage compared to editors like the ones integrated in Dymola, or Modelica Development Tooling [8] because developers immediately recognize these kind of failures. A quick fix using a comparable mechanism as code completion tries to provide a suitable solution.

4.2 Views

The Modelica IDE has several views that display additional information on the projects data. The main view is the Eclipse project explorer that has been extended to fulfill the needs of a Modelica IDE. The contained packages and classes of Modelica files as well as the package structure of referenced libraries are displayed as shown in Figure 6. Each of these classes can be inspected in the editor whereas library documents are opened read-only to avoid the modification of the source code. References inside the documents are linked to allow quick browsing through the source code and the exploration of class declarations. The Eclipse outline view displays the inner structure of the opened document. All actions that are triggered in the user interface are implemented as Open Services Gateway initiative (OSGi) [9] events. Together with an Xtext based DSL that was specified for scripting purposes, we are able to record these actions and save them in a file. The engineer can edit or create a script document for automatic execution of actions. This includes the simulation of models with the coupled solvers Dymola and Mosilab [1].

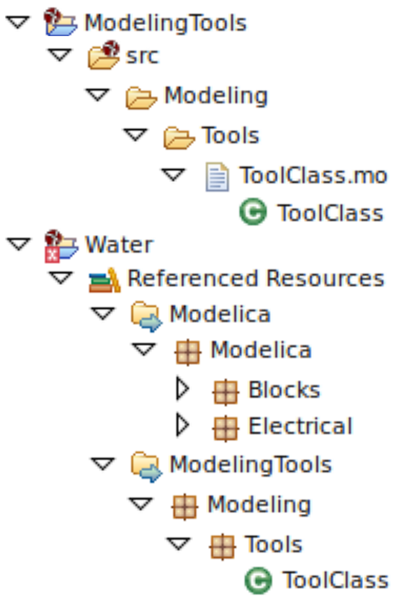


Figure 6: Modelica Project Explorer

5 Conclusion and future work

With the help of MDSD techniques we were able to implement a Modelica IDE in a short period of time. Generated code ensures high quality products and reduces the implementation time by automatic creation of frequently used components like data structures and serialization mechanisms. Since the generated code is based on the same data description (Ecore), interoperability is guaranteed. Therefore augmenting the generated editor with additional functionality became easy.

The introduced IDE enables the Modelica developer to create models fast and easy. The verification and referencing mechanisms ensure correctness throughout the development process. The views on the Modelica data structure help surveying large projects. Encapsulating source code into libraries speeds up the user interaction and encourages the developer to create

[8]http://www.ida.liu.se/ pelab/modelica/OpenModelica/MDT/

[9]Open Services Gateway initiative, http://www.osgi.org

models as components. This helps in creating frameworks that can be shipped and reused as libraries.

During the development some performance issues arose that should be considered in the future. This became evident when the Modelica Standard Library was imported as a source project. The low performance of parsing all library files is engendered by the complex Modelica grammar that causes a lot of back-tracking during parsing. Allowing the use of a different parser generator than ANTLR to generate a `LR(k) parser` might help solving this issue. Also future modifications of the grammar definition might speed up parsing, e.g. by defining a unique start and end terminal sign for annotations. In many documents the percentage of annotations compared to executable code is very high. These parts of code should only be parsed if they are needed for example when visualizing the models. Thus a second optimized parser could be introduced for annotations.

Performance issues are a big problem in our approach. Comparing the parser generated by Xtext with the one of the Modelica SDK[11] [12] which is based on the same parser generator technology (ANTLR) might be useful in finding the reasons for these problems. Furthermore the mechanisms of verification should be compared regarding completeness and performance.

A nice feature to have is an editor that allows the developer to compose models from components graphically, like Dymola does. As we use Ecore as basis for our data, the Graphical Modeling Project Graphical Modeling Project (GMP)[10] might be a suitable solution for this task. However, the embedding of graphical information inside annotations of the Modelica documents instead of separate files might cause problems when using GMP. In our opinion, layout information and executable code should be separated as both are independent concerns.

Furthermore, the refactoring of Modelica models should be addressed in the future. Many of the refactorings introduced in [7] would be helpful in Modelica, as it is suitable for most object oriented languages. In [6] an impressive way of source code refactoring by role definitions is explained and demonstrated based on EMFText[11]. At the moment serializing with Xtext is error-prone and therefore prevented us from integrating the refactoring tool into our project. This will be done as soon as the serialization problems are solved.

[10]http://www.eclipse.org/modeling/emf/

[11]http://www.emftext.org/index.php/EMFText

References

[1] J. Bastian, O. Enge-Rosenblatt, P. Schneider: MOSILAB - a Modelica solver for multiphysics problems with structural variability. Conference on Multiphysics Simulation - Advanced Methods for Industrial Engineering, January, 2010, Bonn, Germany

[2] The Object Management Group (OMG): OCL 2.2 Specification. 2010, http://www.omg.org/spec/OCL/2.2

[3] T.J. Parr, R.W. Quong: ANTLR: A Predicated-LL(k) Parser Generator. Software | Practice and Experience 25(7) (1995) 789-810

[4] F. Budinsky, S.A. Brodsky, E. Merks: Eclipse Modeling Framework. Pearson Education, 2003

[5] M. Seifert, R. Samlaus: Static Source Code Analysis using OCL. In: Proceedings of the Workshop OCL Tools: From Implementation to Evaluation and Comparison, OCL 2008, Satellite event of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008), September 28 - October 3, 2008, Toulouse, France

[6] J. Reimann, M. Seifert, U. Assmann: Role-Based Generic Model Refactoring. In: Lecture Notes in Computer Science (LNCS 6395) - Model Driven Engineering Languages and Systems, Springer, 2010, 78-92

[7] M. Fowler: Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, 1999

[8] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks: EMF: Eclipse Modeling Framework, Addison-Wesley, 2009

[9] Language Specification, Modelica - A Unified Object-Oriented Language for Physical Systems Modeling Version 3.1, May, 2009, https://www.modelica.org

[10] T. Parr: The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, May, 2007

[11] M. Tiller: Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications, In: Proceedings of the 3rd International

Modelica Conference, November 3-4 2003, Linköping, Sweden

[12] P. Harman, M. Tiller: Building Modelica Tools using the Modelica SDK, In: Proceedings 7th Modelica Conference, September 20-22 2009, Como, Italy

[13] M. Krebs: Verifikation von Modelica-Programmen mit OCL, Diploma thesis, TU Dresden 2010