Modelica CDV
A Tool for Visualizing the Structure of Modelica Libraries

Martin Loeffler[1], Michaela Huhn[1], Christoph Richter[2], Roland Kossel[3]
[1]Technical University of Braunschweig, Institute for Programming and Reactive Systems, Germany
[2]Technical University of Braunschweig, Institute for Thermodynamics, Germany
[3]TLK-Thermo GmbH, Germany
m.loeffler@tu-bs.de, m.huhn@tu-bs.de, ch.richter@tu-bs.de, r.kossel@tlk-thermo.de

Abstract

The simulation language Modelica is an object oriented language with all the advantages and potential drawbacks that are characteristic for object oriented programming languages. The reusability of source code and the possibility to develop nicely structured libraries using inheritance, aggregation and polymorphism are two of the main advantages object oriented languages have to offer. Although there are good mechanisms given to structure libraries, one of the drawbacks is that it can become very hard to understand large libraries especially for users who just want to use them to carry out simulations without getting into all the details. The presented work has the goal to provide an easy to use tool that is capable of graphically visualizing the structure of Modelica libraries and that therefore enables the developer as well as the end user of Modelica libraries to better control and understand the structure of libraries.

Keywords: Object oriented modeling; visualization; class diagram

1 Introduction

The object oriented character of Modelica is one of its very important features. It enables the developer to reuse code in a very efficient way, improves team work in the design process of a library and helps the user to easily exchange components in a simulation. There are other advantages that could be added to this list. But there also are drawbacks that almost every Modelica developer and user knows from his own experiences. A growing aggregation depth and multiple inheritance can make a library almost illegible. Tracking bugs, implementing new models or changing existing models might become very difficult for developers that are not completely familiar with the library and its structure. The sustainability of the library might be in danger while it should be improved by using object oriented techniques. Having been in this situation gave us the idea to develop a tool that helps to analyze the structure of Modelica libraries. The developed tool, Modelica CDV, enables the Modelica developer to improve the structure of the developed library and offers the user of the library a simple way to understand its structure. The tool will be available as freeware.

2 Modelica Libraries

To build and maintain models on the basis of Modelica libraries the developer needs a clear and detailed understanding not only of the library elements but also of the object oriented structure of the library and in particular of the relations among the elements. The Modelica Association coordinates the development of free libraries. There are for example libraries available to simulate multi-body systems, to model fuel cells and to model magnetic actuators and drives. For more information about available libraries see [1]. The structuring concepts of Modelica like multiple inheritance, polymorphism, aggregation, and composition are a prerequisite for the compatibility and reuse of submodels (for details see [2]). Consequently, they considerably contribute to the efficiency and conciseness of modeling in Modelica. However, if the structuring concepts are used in combination in a large library, the overall structure may become far from trivial.

Modelica is based on a Cardelli type system [3] and supports multiple inheritance. In difference to nominal type systems like in Java [4], subclasses cannot only be declared explicitely by a keyword like *extends*, e.g. “model A extends model B”, but also implicitly by the fact that a class extends the set of public attributes of another class. For building a subtype this way, all public attributes of the desired base class have to be implemented (using the same names and types) manually. Hence, to reconstruct the inheritance hierarchy within a library the information about the explicitly stated extensions has to be joined with the analysis on extensions implicitly given by inclusion of public attributes.

Modelica provides powerful mechanisms for polymorphism which are an essential device for compositionality and exchangeability of submodels. The first mechanism for polymorphism concerns exchangeable objects, i.e. using the keyword *replaceable* an exchangeable object is built as an attribute of a class. When the class gets instantiated the type of the object can be changed in the class modifier. This way an element of a circuit (for example a resistor) can easily be replaced by another element (for example a capacitor).

The second mechanism for polymorphism is *local classes*. A local class is used if one wants to replace a number of objects within a complex model but all replacements have to coincide on the type, e.g. in an electric circuit several resistors may be replaced under the condition that all of them belong to the same resistor type. Technically, exchangeability is achieved by declaring a local class as a parameter of the model. The parameter is set to a concrete type when instantiating the model and all submodels (objects) that are derived from the local class within the model are set to that concrete type.

The third and most complex mechanism for polymorphism is variable inheritance. It is used for modeling generic objects that can assume the shape of all objects of a category.

```
model GenericResistor
   replaceable model ResistorModel =
                    CeramicResistor extends Resistor;
protected
   extends ResistorModel
end GenericResistor;

model Circuit
   GenericResistor resistor(redeclare model
   ResistorModel = SyntheticResistor);
end Circuit;
```

Figure 1: Modelica code example for variable inheritance

Figure shows the implementation of a generic resistor using variable inheritance. The generic resistor is derived from the exchangeable local class "ResistorModel". Changing the type of the local class when instantiating an object of the generic resistor results in changing the base class of the generic resistor. The advantage compared to the other mechanisms is that the exchange of an object is not only possible inside its container class.

The keyword *redeclare* cannot only be used for exchanging objects. The expression "redeclare record extends GeometryData" for example allows the extension of the already existing record "GeometryData". Additional attributes can be created easily that way.

Member variables (also called attributes) are typed. They are either simple or complex. Complex attributes representing associations between objects further enhance the structuring concepts of Modelica libraries. An association represents a dependency relation between objects. Of particular interest are aggregation and composition as two specific associations that express whole/part relations. For instance, a car can be modeled as composition of a motor, wheels, etc. In Modelica, composition is mainly used to assemble complex objects. Composition, the stronger form, is most appropriate in the context of physical objects where the components are assigned to a unique compositum, e.g. a motor object is part of a particular car. Whether a complex attribute represents an aggregation or composition is hardly to analyse statically because it results from the context in which the attributes will be used.

The structuring concepts of Modelica significantly ease the work of a library developer but they complicate the analysis of a library for the user.

Modelica libraries are stored in hierarchically structured directories. The position of a file reflects the affiliation of the classes stored in this file with respect to a package, because directories represent packages. Thus the structure of the repository partially reflects the structure of the library. Alternatively, all classes of a library can be stored within one file. Then the affiliation to packages only depends on the arrangement of the classes inside the file.

For analyzing the object oriented structure of Modelica libraries it is important to understand the structure of the classes, objects and dependencies among each other. Because the "algorithms" and "equations" inside a class are less relevant for the structure of the library, they are neglected in this paper.

To summarize, Modelica provides a great variety of structuring concepts that may be used in combination and lead to complex library structures that are hardly understandable on the basis of pure code review.

3 Parsing

Parsing is the transformation of text files into an internal data representation. Most of the Modelica parsers are used in commercial applications and are not available for developers. However, there are some free parsers available. An example is ModelicaXML that parses Modelica code to an XML representation [5]. We still did not use one of the free parsers to be as independent as possible. The parser that was developed within the scope of this work only parses information that is required for an object-oriented analysis of the library but could easily be extended to parse more information.

In our application the goal of parsing is the creation of an internal data structure that contains all information of a Modelica library that is required for drawing a class diagram. Therefore every class of a library will be represented by an object containing the following information

- name and type (i.e. package, model,…)
- path of the text file that contains the class
- code
- local classes
- all attributes (member variables)
- all associations (inheritance, aggregation)
- version of the class (if specified)
- package(s) the class belongs to

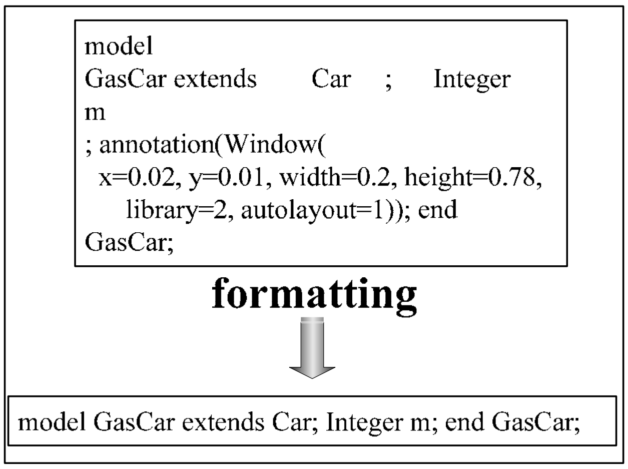
The process of parsing a Modelica library can be decomposed into the following steps

- read in the code
- format the code
- detect classes
- detect attributes of classes
- detect associations
- analyze redeclarations
- perform name lookup
- analyze dependencies caused by Cardelli type system

The steps have to be performed in the given order due to their interdependence. Detecting associations for example cannot be performed before all classes have been detected.

The following passage explains the parser in more detail.

Reading in a Modelica library starts with reading in the required text files. There are in general two ways to store Modelica libraries: all classes and packages of a library can be stored in one file or they can be split up into different files that are nested in folders to represent the affiliation to packages. If the classes are stored in several files, there needs to be a file "package.mo" in each folder declaring the package. All other classes declared in files within this folder or in sub-folders belong to this package. The parser takes the path of the folder that contains the library as an input and checks, whether a file "package.mo" exists in the specified path. The parser will then read in the "package.mo" and all dependent files in the same folder and will register the package affiliation for each file read in. This process is repeated for each subfolder containing a "package.mo". If there is only one file containing the whole library, the content of this file can be read in without analyzing the package affiliation.


```
model
GasCar extends      Car    ;      Integer
m
; annotation(Window(
  x=0.02, y=0.01, width=0.2, height=0.78,
    library=2, autolayout=1)); end
GasCar;
```

```
model GasCar extends Car; Integer m; end GasCar;
```
Figure 2: Example for formatting process during parsing

The entire library is internally available as a character string after this first step. The parser formats the code in a second step carrying out the following operations:

- word-wraps get removed
- superfluous space characters get removed
- annotations other than "version information" get removed (i.e. annotations for documentation, icon information)

This second step simplifies the following process by transforming the unformatted string into a well formatted character string containing only the information necessary to perform an object-oriented analysis. Figure 2 shows an example for the formatting process.

The next step is to detect all classes and to create objects representing them. This step requires considering the nested structure of classes in Modelica. Classes are detected recursively starting at the top level. The beginning and the end of a class is detected on the base of keywords. If a class has been found, an object with all necessary attributes for its representation is generated. Not all attributes of the represented object can be generated at this time because some information (associations, member variables) is not yet available. The name of the class, its source code, comments and information about its package affiliation are stored. The code of a nested Modelica class is not stored within the containing class if it is not a local class. Local classes are for example used for polymorphism. The model "ResistorModel" from Figure 1 is a local class.

Detecting attributes of each class includes detecting the type and the name of each attribute as well as its default value and the comment if specified. Start values and further attributes such as min and max values are omitted but could be included in a future version.

The keyword *extends* defines inheritance in Modelica. A derived class inherits all attributes from its super class, implying that all attributes of the super class have to be copied into the derived class However, this is not possible at that point because all classes are represented by an object but thery are not linked to each other. This means that copying the attributes from super to derived classes has to be perfomed after the name lookup. If an attribute is not primitive it is handled as aggregation.

```
package BaseGeometry
        replaceable record GeometryData
                Real length;
        end GeometryData
end BaseGeometry;

package Cylinder extends BaseGeometry;
        redeclare record extends GeometryData
                Real diameter;
        end GeometryData;
end Cylinder;
```

Figure 3: Modelica code example for redeclaration

The keyword *redeclare* indicates in Modelica that the type of the object can be changed. Figure 3 shows the extension of a record "GeometryData" within the package "Cylinder" "GeometryData" is extended by the attribute "diameter". If the class "GeometryData" from "Cylinder" is used somewhere, it contains the attribute "diameter". The parser handles this just like inheritance. A new class is internally generated that contains all attributes from "GeometryData" plus the attribute "diameter". This class belongs to the package "Cylinder" and will later be displayed as derived from "GeometryData".

All information collected until now is represented as character strings. As mentioned earlier, it is very often necessary to have a link to an object representing a certain class. The already described situation of inheritance is a very good example in which all attributes of the super class have to be copied to the derived class. If there is a definition like "model GasCar extends Car" the parser has to resolve the class "Car". The problem arising here is that there might be several classes with the name "Car" within the analyzed library because Modelica allows different classes having the same name. Name lookup ensures that the correct class is used within the given context. In general all classes within the package containing the class whose name has to be resolved are possible choices. It is also possible to give a bit more specific information when extending a class. One could for example write "model GasCar extends Basics.Car" "Basics" is in this case the package that the class "Car" belongs to. Another important case to be considered is the "import" statement. With this keyword, namespaces can be defined or used. If there is a definition like "import myCar = Basics.Car" the name lookup has to be performed with "Basics.Car" instead of "myCar".

It is also possible to use short notations such as "package C = B;" in Modelica. In this case all classes of package "B" get duplicated and copied into package "C". The declaration can be interpreted as equivalent to its long form "package C extends B". The associations of a class are stored inside the representing object. Thereby associations from and to another class are available.

Modelica uses the Cardelli type system which is a structural type system. Many other object oriented languages such as Java use a nominal type system (see [3] for more information). There has been a discussion within the Modelica language group to change the Modelica type system in a future version but no decision has been made so far. To analyze type equivalence of classes or to find subtypes of a class it is necessary to compare all public attributes of each class. If two classes have the same public attributes they are type equivalent within the scope of a Cardelli type system. If they have at minimum the same public attributes the class with more public attributes is a subtype of the other class. When there is a large number of classes with many attributes, comparing all of them with each other can take quite some time.

After all parsing steps have been performed the internal data representation is available and can be used for displaying the object oriented structure of the library in the style of a UML class diagram (see [6]).

4 Layout

For the automated graphical representation of the structure of Modelica libraries, the selection and tuning of the layout algorithm is crucial. Thus, we consider three types of layout algorithms for drawing the object oriented structure of Modelica libraries. Layout algorithms rely on graph theory. The object oriented structure of libraries is interpreted as a directed graph where a node represents a class and an edge represents an inheritance relationship or aggregation. The selected layout algorithms optimize the graphical representation according to the following goals:

- Minimization of the area used for the resulting chart because libraries can be large.
- Minimization of the number of crossing edges supports the understanding and conceives the diagram. If there is no crossing edge the graph is called planar. The direction of edges is also an important fact: A class that inherits from another should be placed below the class it inherits from.
- Computational efficiency is important so that even for large libraries the class diagrams are generated in an acceptable time.

The first group of layout algorithms relevant for the representation of structural information is the group of the so-called “tree algorithms” [7] which are available in many variations. They are especially suited to illustrate the inheritance relationships of classes while their runtime is linear. A class is represented by a node and all derived subclasses become children of this node. However, tree algorithms are less appropriate for the layout of aggregation relationships since the generated trees can become wide and might need a large area.

Another way to layout graphs is the “Spring Embedder” [8].The graph represents a physical model where nodes repulse each other. The closer two independent nodes get, the larger the mutual force of repulsion becomes. Nodes also gravitate towards each other in case of a common edge. This algorithm works iteratively. All forces are calculated and the nodes are relocated according to the affecting forces. After a certain number of iterations all forces will be balanced. The size of the resulting chart is small but it might contain a lot of crossing edges. Moreover, runtime of the Spring Embedder may become a critical issue.

The third group of algorithms minimizes the set of crossing edges.

The Spring Embedder and the algorithm for minimizing crossing edges do not care about the direction of edges and the right adjustment of classes that inherit from another. Each algorithm has its advantages and drawbacks. In order to get a good layout, it is necessary to use two algorithms in combination.

Modelica CDV generates a class diagram closely related to the UML notation (see [6]). The most important elements, the classes, are represented by rectangular boxes containing the name, the attributes and the operations. Here we will omit the operations, i.e. equations and algorithms of a model.

The layout module in Modelica CDV is still subject of discussions and experiments as the advantages and disadvantages of different variants have to be balanced carefully.

A class diagram is considered as a directed graph where a node represents a class and an edge represents an association, i.e. inheritance or aggregation.

For a good layout of a Modelica library several partially contradicting criteria have to be taken into account.

Usually, derived classes are placed below their super class. Consequently, the algorithm tries to do this the same way. Another optimization criterion is a short distance between classes having a relation since one often needs to look at the associated classes as a whole to understand the aggregation or inheritance structure. Hence, minimizing the distance improves readability. But when minimizing the distance between nodes one also decreases the area on which classes are placed. Unfortunately, in larger diagrams with a lot of complex associations you often will not see the end of an association without changing the point of view. Additionally, crossing edges are complicating readability. To simplify the detection of crossing edges, associations are restricted to horizontal and vertical straight lines.

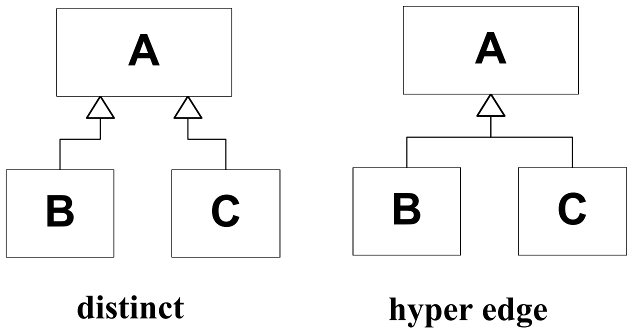


Figure 4: Edge notation for inheritance

After several experiments we decided that the user should direct the layout to his point of interest: Thus, the user may select the elements of a class he wants to see. This way, the user may choose the level of detail he is interested in. If detailed information is faded out, the user will see more classes and associations on the screen.

The size of a class is calculated dynamically before the layout algorithm starts. The size of classes depends on the settings given by the user. Specifying that additional attributes should be shown in the diagram will change the dimension of classes. Every time settings are modified all calculations have to be repeated because the algorithm calculates the absolute position of every class on the panel. If a class becomes larger it probably would overlap with another one otherwise. The width of the class representation results from the longest word that has to be displayed inside a class. For inheritance associations a "hyper edge" notation is used (see Figure 4). Aggregation associations are characterized by their multiplicity. Therefore an inscription would be needed which might be misleading at the hyper edge. For this reason aggregation is displayed as distinct edges.

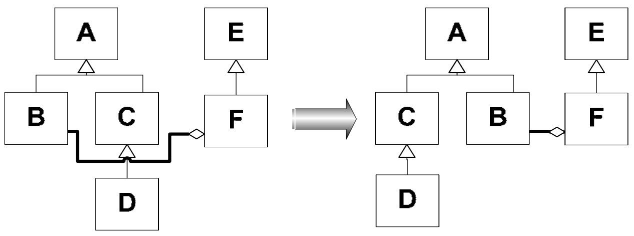


Figure 6: Swapping leaves for minimizing crossing edges

The algorithm for lay outing the classes first calculates the size of every class. Thereafter inheritance associations are analysed and layouted by a tree algorithm from bottom to top and the dimension of the resulting tree is calculated.

The algorithm for adding aggregation associations is still under investigation. After all aggregation associations have been added a swap procedure will

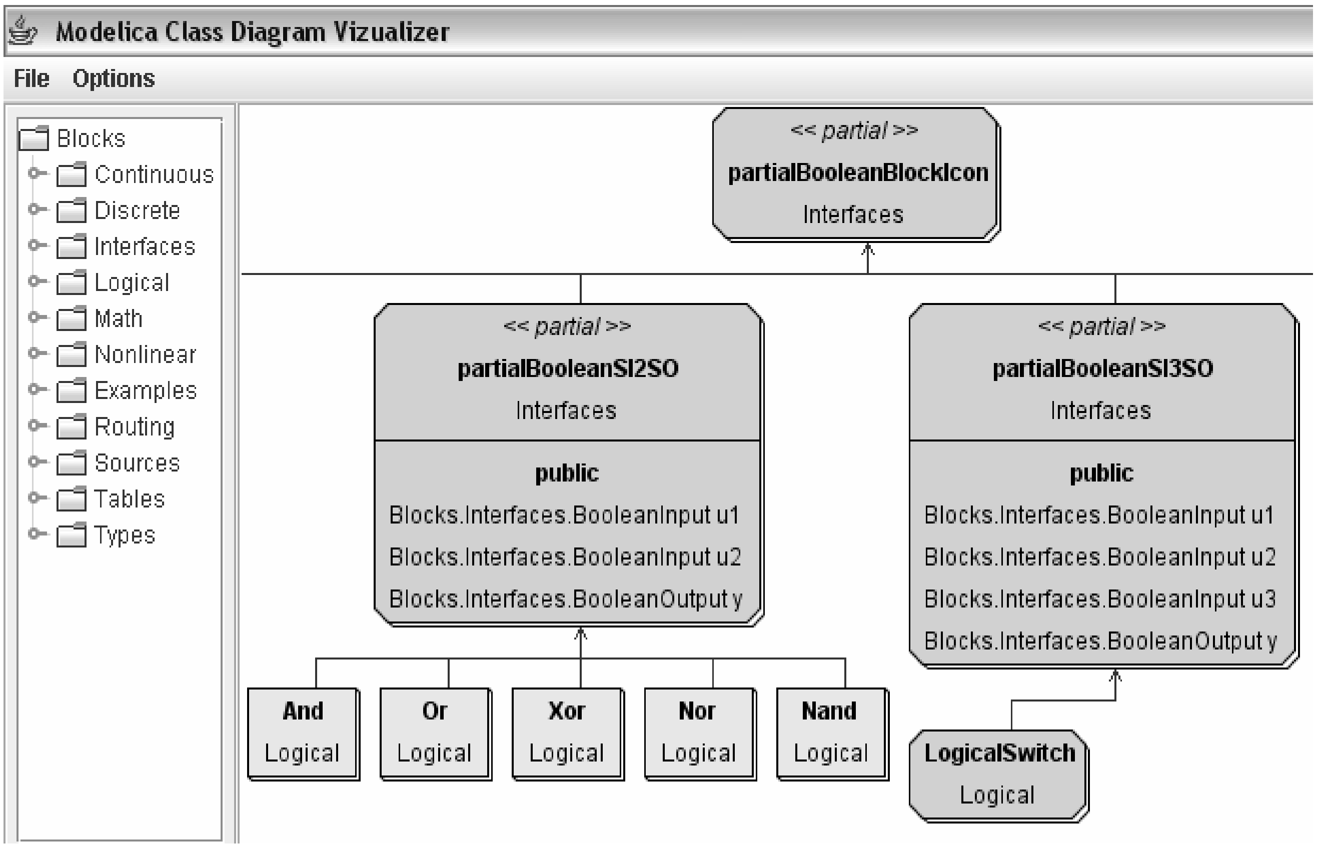


Figure 5: Screenshot of the Modelica Class Diagram Visualizer

remove crossing edges by swapping some leaves (inherited classes) of a tree or the trees itself (see Figure 6). Now nails for the associations will be calculated and all elements will be displayed. The algorithm for swapping and rearranging the leaves (classes) is still in a prototype state at the moment.

5 Using Modelica CDV

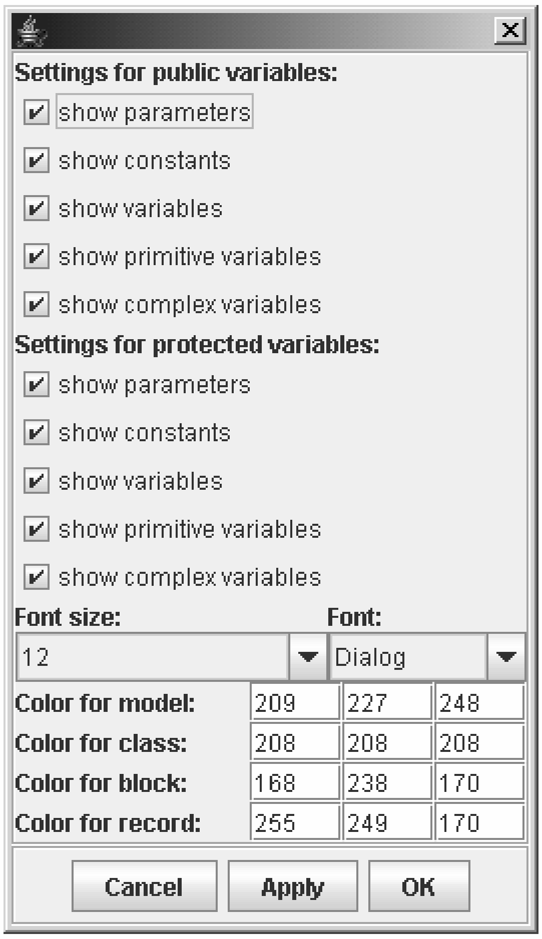
The purpose of Modelica CDV is to help the Modelica user as well as the Modelica developer with getting an overview over the structure of libraries.

The graphical user interface that was completely developed in Java using Swing [9] can be configured to best suite the users' preferences. The user may select color and font for each type of class and also choose to display or hide additional information that is usually not displayed in a class diagram but might be handy when analyzing libraries. Figure 7 shows a screenshot of the current configuration dialog. The user can decide to display parameters, constants and variables for both, the public and the protected section, in the class diagram.

Figure 5 shows a screenshot of a class diagram generated by Modelica CDV. The package "Blocks" from the Modelica Standard Library 2.2 was parsed and displayed for this example. Note that only the inheritance associations are displayed in the example screenshot. The different class types such as blocks (i.e. "LogicalSwitch") or models (i.e. "And") are discriminated by their graphical appearance both in shape and color and according to the user's settings. Information about the variables contained in each class is also displayed. The final version of Modelica CDV will also display connectors as small icons.

The package tree containing all classes of the parsed library is displayed on the left hand side of the class diagram as shown in Figure 5. When selecting a class with the mouse the respective block in the class diagram on the right hand side is highlighted and centered.

An important aspect when using Modelica CDV is the time it takes to parse, layout and display a library. The Modelica Standard Library in the version 2.2 containing about 2500 classes was parsed as a representative example which took about two minutes on a standard laptop computer. While the Modelica Standard Library is very likely much larger than most libraries that will be displayed with Modelica CDV, two minutes still is too long. The user can therefore choose to save the parsed library which cuts down the time requirements to a few seconds. The user just has to be aware that he is using a pre-computed version of the library when using the saved version. A warning is displayed to remind the user of the current operation status.


Settings for public variables:

- [✓] show parameters
- [✓] show constants
- [✓] show variables
- [✓] show primitive variables
- [✓] show complex variables

Settings for protected variables:

- [✓] show parameters
- [✓] show constants
- [✓] show variables
- [✓] show primitive variables
- [✓] show complex variables

Font size:	Font:
12	Dialog

Color for model:	209	227	248
Color for class:	208	208	208
Color for block:	168	238	170
Color for record:	255	249	170

Figure 7: Screenshot of the option dialog in Modelica CDV

6 Conclusions

Modelica CDV is a tool to help Modelica developers and users to better understand the structure of libraries by generating class diagrams closely related to the UML notation. It uses a combination of different layout algorithms to automatically generate a class diagram that is as readable as possible. In the class diagram additional information (i.e. about variables) may be displayed and it can be configured according to the users' preferences.

Visualizing the structure of libraries is the first step towards improving the readability which ensures its sustainability. The developed tool is available as freeware to the Modelica community. The current version only parses and displays information that is required for an object-oriented analysis. Additional information about equations and algorithms and enhanced analysis that might be of interest is currently neglected but might be included in a future version.

References

[1] Modelica – Modeling of Complex Physical Systems, Modelica Libraries, http://www.modelica.org/library/

[2] ModelicaAssociation, 2005, Modelica – A Unified Object-Oriented Language for Physical Systems Modeling, Language Specifications, Version 2.2

[3] A.B. Tucker (Ed.), The Computer Science and Engineering Handbook, CRC Press, 1997

[4] B. Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2002

[5] A Pop and P. Fritzon, 2004, ModelicaXML: A Modelca XML Representation with Application, Proc. of 4th International Modelica Conference, Hamburg, Germany

[6] M. Jeckle, C. Rupp, J. Hahn, Barbara Zengler, S. Queins, UML 2 glasklar, Hanser Verlag München, Wien, 2004

[7] M.Kaufmann, D. Wagner, „Drawing Graphs – Methods and Models“, Springer Verlag, 2001

[8] Guiseppe Di Battista, Peter Eades, Roberto Tamassia und Ioannis G. Tollis, “Graph Drawing – Algorithms for the Visualization of Graphs”, Prentice Hall, 1999

[9] Sun Microsystems, Java Foundation Classes, http://java.sun.com/products/jfc/