O. Johansson, A. Pop, P. Fritzson
Linköping University, Sweden

ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries
pp. 445-454

ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries

Olof Johansson, Adrian Pop, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{olojo,adrpo,petfr}@ida.liu.se

Abstract

This paper presents ModelicaDB, a tool that provides several kinds of queries on repositories of Modelica models.

The Modelica language has a growing user community that produce a large and increasing code base of models.

However, the reuse of models within the Modelica community can be greatly hampered in the future if there are no tools to address a number of management issues (i.e. scalable searching, analysing, crossreferencing, checking, etc) of such a large repository of models.

We try to address these issues by providing the Modelica community with a ModelicaDB database for storing models and services for quering this database to perform a wide range of model engineering tasks in a scalable fashion.

In the long-term, this work also aims at providing integration between Modelica tools and advanced product development processes that rely on database technology.

1 Introduction

The long-term goal of this work is to efficiently integrate Modelica modeling simulation environments into the overall product development process for products that require advanced systems engineering [2].

There are many engineering processes that precede modeling and simulation, and which output information that defines structure, configuration, and input parameter data to simulation models of the product.

The following are of particular importance:

- The implementation structure of the product that defines its hierarchical division into different subsystems, their components with parameter values, and component interconnections.
- Run-cases derived from the product requirements, which define critical behaviour that the product must achieve, and for which alternative implementation structures' behaviour is evaluated with simulation.

Given these, the simulation model designer can select valid component models from Modelica libraries for the components in the implementation structure, and verify that the simulation component's parameter values are compatible and transferable from the information in the provided implementation structure.

With valid component models assigned and mappings of their parameters, other tools can assemble a final simulation model setup for execution and post-processing of the simulation results for evaluation analysis in subsequent engineering processes.

One purpose of ModelicaDB is to provide fast access to possibly relevant component models in Modelica Libraries, such that the assignment work can be speeded up with automation tools.

In many cases, a matching model component will not be available and ready for use in the Modelica Libraries, so the task of selecting component models is augmented by writing new ones or assembling valid component models from other components in the Modelica library.

Such work is a creative design task, which is significantly aided if the designer has tools for searching, analysing, crossreferencing, and checking the libraries.

The used libraries have been developed by experienced library developers, and contain valuable design-pattern knowledge of how to properly design and implement models, components and libraries. With fast browsing and navigation tools, the designer can quickly find similar designs to the one that is needed, study how they are used/reused as components in other simulation models, and get a good understanding of how to build a new simulation component.

The continuous development and improvement of Modelica libraries by the Modelica design group and similar efforts within companies, indicates that tools with ModelicaDB functionality would be valuable to the Modelica community [7] for many other purposes than we intend here.

Preliminary statistics from the Modelica 2.1 and Modelica 1.5 libraries definitely show that this kind of tool would be helpful for many engineers for grasping, sharing, reusing, and following the large efforts in simulation model development work that is being simultaneously conducted by many people.

The following sections of the paper provide statistics from the current releases of the Modelica libraries, examples of use-cases for ModelicaDB, an overview of the product development process and the intended role of ModelicaDB, the functionality in the user interface of ModelicaDB, an example of an SQL-query on the database and finally results, experiences, future work and conclusions.

2 Statistics from Modelica Standard Libraries

Preliminary analysis of the Standard Modelica 1.5 and 2.1 libraries give the following statistics:

Modelica Library	V 1_5	V 2_1	V 2_1+
Source files	36	87	144
Imports	93	286	343
Class definitions	910	1447	3141
Components	1628	4636	6915
Equations	1055	2768	3262
Algorithms	99	633	1290
Component_refs	30304	60838	92636
Expression_lists	14736	23715	25354
Real literals	4413	5833	33158
Comments	1720	4755	5649
String Comments	1322	3722	5611
Annotations	1326	3120	5093
String literals	3503	7218	13350
Integer literals	33187	59604	67373
Other	88991	157760	235806
Total elements:	183323	336422	499125

The number columns show the Modelica language element count from different releases of the Modelica standard libraries. Modelica 1_5 was downloaded from the public library page [8]. Modelica 2_1 and 2_1+ were obtained from the Modelica CVS repository 2004-11-15.

V 2_1+ includes the following libraries: Modelica, ModelicaReference, ModelicaTest, Modelica_Fluid, Modelica_Interpolation, Modelica_Media and TeachingMaterial.

The source code directory contents of the libraries was converted to a single xml file for each library release by ModelicaXML, which then were preprocessed for import into ModelicaDB.

The *Imports* row is an indicator of reuse. The Component_refs row gives the count of the uses of a component variables in expressions.

The *Comment* row is a higher level parse node for *String_comments* and *Annotations* .

String literals and *Integer literals* are heavily used within annotations, especially for graphical object annotations in Modelica diagrams.

The above statistics shows that the size of the standard libraries is substantial. Commercial Modelica development tools [1],[3] provide user interfaces with tree views of the package hierarchy, connection diagrams, and string based text searches, for quick navigation in the libraries.

ModelicaDB adds additional search facilities and other types of tree views on the libraries, that can help to speed up the task of creating a new simulation component that efficiently reuses existing component models and design-pattern knowledge.

3 Use-Case Examples for ModelicaDB

The following section briefly describes use-cases that illustrate use of additional types of views on Modelica library structures. The views are computed in ModelicaDB, and presented in a tree- or list- based user interface that enables quick navigation with pointing and clicking.

3.1 Finding Relevant Simulation Components

The following example use cases illustrate subtasks in the process of finding reusable components and code sections for building a new simulation component.

- Finding component models with knowledge of the SI-units their instances will need.
- Finding component models with knowledge of their connectors.
- Finding equations with knowledge of the type of the variables used in the expression.
- Finding algorithms with knowledge of their function call parameters.

3.2 Finding Relevant Component Using a Categorization Tree

Categorization trees are an “add-on” feature to Modelica libraries, implemented with annotations.

Categorization trees allow a user to with a few clicks down the tree find a set of relevant component models. The categorization tree itself is an aid for remembering where to find certain components.

Modelica objects can be annotated with a category, which makes them easier to find with the aid of a category system (also known as classification system, focusing on some aspect). Examples of categories are “Electrical Components”, “Motors”, “Transistors”, “Capacitors” etc.

A category system is organized into a tree, where the root category node contains all Modelica objects that have that type of category or any of its sub categories. Sub nodes in the category tree increase the specialization in the categorization. Leaf nodes in the categorization tree usually justify their existence if there are 5-25 component models under this node.

There are many standardized categorization systems used in industry. Classification trees applied on electrical components are specified by IEC [22], and applied in succeding industrial standards like RosettaNet Technical Dictionary which contains a much larger library of classes [18]. ISO-31 [23] categorizes quantities and units into 13 chapters and is well known from the Modelica SIunits in the standard libraries. There are other examples of large classifications systems for standard terms used in e-business.

Commercial design tools for the design processes immediately before simulation, like process and instrumentation diagrams (P&ID), electrical design and control system design usually contain a categorization system for reusable components in their component library catalogues.

The following use cases examples can be well supported with a classification tree.

- Finding a component model for a certain purpose.
- Finding connectors for a certain purpose.
- Finding equations for a certain purpose.

The category method of finding Modelica components requires a library administrator to manually organize or load a standardized categorization tree , and then annotate the component models with their classifications, see section 3.7.

Once the classification tree structure is decided, pattern maching searches in the Modelica repository can be used to populate the categories. For example, a classification tree for equations that compute a value of a certain SIunit type, can be organized according to ISO-31 and the standard SI-units library, and populated with equations whose left hand side variable matches the corresponding SIunit type category.

3.3 Verifying that a component is trustworthy

Simulation results must be accurate in order to provide correct decision support to the product design process. The following use-cases illustrate ways the engineer can determine this.

- Finding examples of usage
- Computing statistics of reuse in other models
- Computing statistics of use of certain design pattern

The last use-case applies when the engineer has designed a new simulation component, and wants to check to what extent others library developers have used a similar design pattern. Such statistics can indicate if this is a good way to solve the problem, and can direct more detailed searches for gaining further confidence, or ideas of how to improve a design.

3.4 Finding relevant design patterns

There are many ways to solve a type of problem. Some of these may prove to be better than others and tend to re-occcur in many places as design patterns. The characteristics of components that play a certain role in the re-occuring interconnection structure of the design pattern, can be used as search criteria.

3.5 Finding relevant naming conventions and documentation

Following naming conventions is important for efficient communication in a large community. Naming conventions usually vary between different engineering disciplines due to the history of their body of knowledge and decisions made by library authors.

When extending or reusing a library, it is valuable to follow the relevant conventions to ease reuse within the community. The use-cases below illustrate how this could be supported.

- Finding Naming Conventions for Variables and Parameters

Pattern search of sorted listings of variable names for a certain type of SI-unit variable, which may play a certain role in an equation.

- Finding References to Literature and Documentation

Searching documentation strings of pattern matched components for references (e.g. brackets or other text patterns that indicate a reference)

3.6 Checking that the New Component Follows Design Rules

Engineering domains may pose restrictions on simulation models that are not possible to enforce directly in Modelica. The same applies for company specific design rules that accumulate from experience, and quality assurance procedures that reduce the cost for errors and maintenance.

The following use-cases are simple examples of design rule checking.

3.6.1 Checking Naming Conventions for Classes, Components

Pattern search and listing of Modelica object identifiers names in a model that do not follow a certain style, or convention.

3.6.2 Checking Complete Documentation

Pattern search that for instance all components in a class have a comment string etc.

3.6.3 Checking Use of SIunits

Pattern search for variables whose type is not derived from SIunits, and are not an array index or similar.

3.7 Managing Product Specific Library Development

While developing a complex product that requires systems engineering, much can be gained by reducing the number of variants of a certain type of simulation component.

The following use cases show how a library developer can direct the users to the best components for various purposes, and identify targets for refactoring amongst existing components.

3.7.1 Finding Candidate Components for Categorization

Various pattern searches that detect component features that make them interesting for a certain classification, and perhaps exclude already classified components

3.7.2 Finding Duplicates or Variants of the Same Models

Duplicates or variants of the same models can be found by pattern searches that compares component sets of variable types and equation patterns within a class definition. Patterns that detect the same equations, based on variable types, where the variables themselves just have different names.

3.8 Additional Analyses and Metrics

Michael Tiller presented analyses and metrics in [21], which inspire development of additional reporting applications which can be computed with SQL-queries on ModelicaDB.

3.9 Automatic Composition and Configuration of new Models

ModelicaDB augments the work presented in [12] on automatic composition and configuration of new models. Using ModelicaDB, designers can compose new models by blending template like models with configuration information stored in other sources (text or XML files, databases, etc) to create new models which are configured accordingly.

4 The ModelicaDB Context and Architecture

Figure 1 shows the role of ModelicaDB amongst some of its surrounding engineering processes, connected with major workflow arrows. Engineering tools (FMDesign, ModelicaDB, ModelicaXML, Modelica Simulation tool) support some of the processes. Engineering models are stored in files (Simulation program, Modelica libraries) and in databases (FMDesign database, ModelicaDB database).

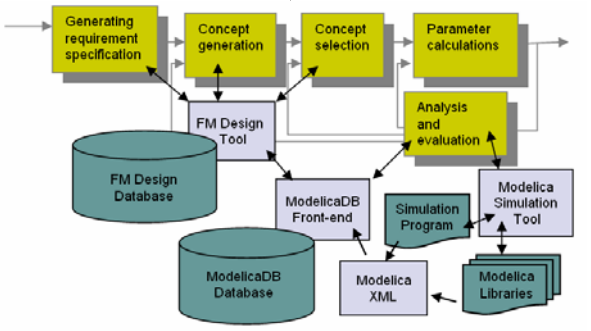


Figure 1. ModelicaDB in its context

The ModelicaDB front-end and database are described in more detail below. We start by briefly describing the role of the other tools in the integrated framework [15].

FMDesign is a tool for designing product concepts with the aid of integrated requirement trees, function-means trees, product concept trees, and implementation trees. The implementation tree specifies the product structure and its interacting components on a level that is detailed enough so its behaviour can be modeled and simulated.

The simulation is deferred to one of the existing Modelica Simulation Tools [1][3][10]. All manual editing of simulation models are performed in one of these tools, and the component models are stored in Modelica Libraries. The simulation program is generated from the configuration information stored in the implementation tree for the product concept.

4.1 ModelicaXML Files

ModelicaXML is a program that converts Modelica source code into XML-files [1]. Recent additions to ModelicaXML is functionality for converting a whole Modelica Library stored in a directory structure into one XML-file. The size of the created files for the standard Modelica libraries version 1.5 and 2.1 is 16 MB and 30 MB respectively.

4.2 ModelicaDB Front-End Tool

This tool parses the ModelicaXML file and builds an object structure in primary memory which can be synchronized or stored into tables in the ModelicaDB database.

The tool also contains a graphical user interface, for fast navigation of the component model level Modelica language constructs.

More detailed constructs like expressions are modelled as parse nodes in the database.

Appendix A shows the class diagram for the UML-model [6] that serves as design specification of ModelicaDB. [20] documents the whole UML-model that was used for generating most of the ModelicaDB specific source code that implements the front end and database. The core specification for designing the ModelicaDB UML model was the ModelicaXML DTD [11]. The reference work used for its documentation was [4] and [9].

The user interface displays the results of queries specific for the use-cases described in section 3 such that found Modelica objects can be quickly inspected, and further navigated, including retrieving and displaying the original Modelica source code from the files. Section 5 gives an overview of the user interface.

4.3 ModelicaDB Database

This is a relational database that is used for processing declarative SQL-queries that do complex searching, compute the analyses, crossreferences, and checks.

The structure of the database is given in the UML class diagram in Appendix A. The database schema can be downloaded from [20] .

The benefit of using an SQL-database instead of navigating parse trees, is that the SQL database optimizer in cooperation with indexes on tables can compute complex queries much faster on a large library, than a traditional procedural or object-oriented program which navigates the parse tree structures.

The performance benefit of a database is first noticed when the number of stored language objects exceed a certain breakpoint.

Writing SQL-queries may be tricky at first, but usually results in little code for a complex task. With a reusable set of SQL-queries for various types of searches and analyses, a new query variant can quickly be written using copy and modify, while verifying that it produces expected results by executing it with paremeters that match a small but well known example model.

5 ModelicaDB Functionality

This section gives a walkthrough of ModelicaDB front-end functionality available in its user interface. An UML class diagram of the user interface design is given in Appendix B. Tree views have a look and feel similar to the windows file explorer, where the folder icons indicate the class restriction or other meta-classes shown in the UML-diagram in Appendix A. When an object shown in a tree view is double-clicked, a form appears which shows the objects attribute values and direct relationships to other objects.

5.1 ModelicaRepository Main Window

This window allows the user to open a Modelica repository file stored in a fast binary format. The user can also login to the database, load the complete repository for caching at the local workstation or synchronize the cached version with the latest changes in the database.

The main window provides a category tree for finding suitable Modelica models, and open a Modelica Model window for these.

Two different types of catalog windows can also be opened from the Modelica repository window. The class catalog window shows categorization trees for Modelica classes, that are organized according to a suitable standard, which allows engineers to quickly find relevant component models for a certain type of product component. Section 3.2 gave a use-case example.

5.2 ModelicaModel Window

The ModelicaModel window provides navigation of all Modelica objects that are recursively owned by a ModelicaModel object, see Appendix A.

In ModelicaDB, a ModelicaModel object is the root of all packages and component models that are assembled within one particular ModelicaXML file. Different versions of the Modelica Standard Libraries, are for example rooted in different ModelicaModel objects in the ModelicaRepository.

The window provides import functionality for ModelicaXML files, and various menu commands for searching, analyzing, crossreferencing, and checking selected Modelica objects.

Below the command menu, the window shows the package hiearchy tree which can be expanded down to class definitions, their components, equations and algorithms.

A separate class browser window can be opened for a selected class, which shows its inheritance hierarchy as a tree. Classes that have no superclasses are shown as parallel root node sorted by the identifier name. Classes that are extended from multiple superclasses, are rooted in the first declared superclass in the tree. The second and remaining extended superclasses are listed together on the level below the class with special object icon, followed by the subclasses that extend the class. Icon superclasses can optionally be filtered away, with a special view setting.

A separate model browser can be opened for a class or component, and shows its part-of structure as a tree. When expanding a component node in the tree, its defining class is shown on the level below, and can be further expanded in a similar way.

The Modelica model window also provides access to various types of two dimensional diagrams, which lay out various structures and interconnections of component models in different views, and are intended as support for seminar discussions on library design and refactoring issues. These diagrams are still in their early design stages, and need some prototype iterations to become useful.

5.3 Report Window for Result Sets

Result sets from searches, analyses, crossreferences and checks, are displayed as interactive report listings in a separate report window with numbered rows. Each row is associated with one object in the Modelica database. If the row is double clicked further details about this object can be inspected in a form. A report row may optionally contain a short text message that further explains the reasons for including its associated object in the report.

Examples of result rows, and their text messages for two use cases is given below.

5.3.1 Finding Component Classes with Knowledge of the SI-units Their Instances Will Need

This is a simple use-case that also can be executed in existing Modelica tools, for instance using the Search facility in Dymola or evaluating a pattern search expression in a MathModelica document cell. This use case can be a benchmark for comparing the time it takes the user to complete the use-case with various user interface implementations.

An instance of this use-case in the ModelicaDB front-end can be as follows:

1) The user has opened the ModelicaModel window on the Modelica 2.1 standard library. In one of the tree-views the users selects the Modelica object that represents Modelica.SIunits.Resistance, which is defined as:

```
type Resistance = Real (
    final quantity="Resistance",
    final unit="Ohm",
    min=0);
```

2) The user issues the *Report* command from the windows top menu, and gets a list of all use cases that can be reported in a dialog box.

3) The user selects "*Find component declarations for predefined types*", which is the short name for this use case.

4) The ModelicaDB front-end processes the query and presents the result rows for the found components in the Report window sorted according to the component variable name. There they can be clicked for further inspection in a form, or set in the focus of one of the available browser window types which better shows a Modelica objects surrounding context.

5.3.2 Computing Statistics of Use of Certain Design Patterns

This is a more complex use-case that illustrates the benefit of storing large Modelica libraries in a relational database.

The use-case instance is checking to what extent other designers have created component models that uses the simulation model of Electromotoric force `Modelica.Electrical.Analog.Basic.EMF` in direct connection with a current sensor component of `Modelica.Electrical.Analog.Sensors.CurrentSensor`.

1) The user has opened the ModelicaModel window on the Modelica 2.1 standard library, and opened the package hierarchy tree down to `Modelica.Electrical.Analog.Basic`. The users selects the EMF class with a first click, and then adds the `CurrentSensor` class to the selection by shift-clicking it in another model browser window showing the `Electrical.Analog` package.

2) The user issues the Report command from the windows top menu, and selects "*Sum connected component uses.*".

3) The ModelicaDB front-end generates an SQL-query with attribute value information from the selected objects as restricting search criteria, sends the query to ModelicaDB and displays the result below in a Report window.

```
0001 ModelicaModel Modelica1_5: count=1
0002 ModelicaModel Modelica2_1: count=1
```

6 SQL-Query Example

The following example show the SQL-queriey for the use case described in 5.3.2.

```
select mmFound.name, count(*)
from class cl1,
     class cl2,
     class clFound,
     modelicamodel mmFound,
     component co1,
     component co2,
     equation eqFound,
     parsenode pn1,
     identifierreference ir1,
     identifierreference ir2
where cl1.identifier = 'EMF'
  and cl1.lowid = co1.classifier_lowid
  and ir1.modelicaobject_lowid = co1.lowid
  and ir1.parsenode_lowid = pn1.lowid
  and pn1.nodeType='equ_connect'
  and ir2.parsenode_lowid = pn1.lowid
  and ir2.lowid != ir1.lowid
  and ir2.modelicaobject_lowid = co2.lowid
  and co2.lowid != co1.lowid
  and co2.classifier_lowid = cl2.lowid
  and cl2.identifier = 'CurrentSensor'
  and pn1.modelicaelement_lowid =
eqFound.lowid
  and eqFound.class_lowid = clFound.lowid
  and clFound.model_lowid = mmFound.lowid
group by mmFound.name
```

The query returns the name of the found ModelicaModel objects, and counts the number of connect equations in the found model, that refers to components that are declared as classes with the name 'EMF' and 'CurrentSensor'.

7 Results and Experience

A first prototype version of ModelicaDB has been implemented that verified the approach. More work is required to cover more advanced features of the Modelica language.

Most of the implementation work was rather straitforward, once the UML models in Appendix A, and underlying detailed specifications [20] were completed. The exception was the currently 408 mapping rules that convert the parsed ModelicaXML elements into connected object structures according to the UML model in the ModelicaDB front end, so they can be stored in the database.

The Modelica grammar and ModelicaXML structures contain many details and requires several passes to resolve all references. This also involves searching the name spaces according to the static and dynamic lookup functions (Chapter 3 in [4]), and resolving identifier references to imported classes in libraries that are not in the current ModelicaXML file.

Other issues that require more work are:

- ModelicaXML-to-ModelicaDB mapping rules, which are currently initially generated from preprocessing of large representative ModelicaXML files, and then manually extended with actions that specify how priority sorted matching patterns of XML-elements are stored into objects in the ModelicaDB front-end. To get better verification of full grammar functionality coverage, the rules should be generated directly from the ModelicaXML DTD, or another formal Modelica grammar specification, but such an approach requires more research.
- How to represent modifications in ModelicaDB, so the users SQL-query pattern searches also hit modified classes, without the need for expensive processing of modification "deltas" in parse node trees.

Some other interesting research results that came out of this work are

- Identification of semantic equivalent functionality between the Modelica language and the industry standard ontology languages UML and Rosettanet technical Dictionary [13]. Thus it is definitely possible to reuse relevant ontologies originating from other modeling languages for exchanging existing product data with Modelica simulation model development tasks. Other more distant future applications can be inferred from [5].

New technical results are:

- Formalized Modelica simulation model interchange format in the form of a DTD, for Modelica 2.1. This DTD contains 88 language elements, and is described in [11] and [9]. The latest version has some small modifications and can be downloaded from the reference URL at [11].
- Extensions of the ModelicaXML tool for packaging directory structures containing Modelica source code libraries into one XML-file.
- UML-model of the Modelica database.
- Implementation of a relational database for searching, analysing, cross-referencing and checking of Modelica libraries [20].

8 Future work

Future work will be determined after members of the Modelica Design group and involved researchers at Linköping University have tested ModelicaDB prototypes and given recommendations for future work.

9 Conclusions

This paper reports work on ModelicaDB – a tool that provides database storage and query of Modelica models. We believe that given proper integration with engineering product development tools, ModelicaDB will be of great value on finding related product models, quick access through categorization, and assisting with a number of other related tasks.

A first prototype of the tool has been implemented. A full database schema has been designed and tested against queries, a Modelica library parser that converts libraries into XML form has been implemented. The main remaining task is completing the set of rules that map ModelicaXML elements to ModelicaDB objects.

Acknowledgements

Swedish Foundation for Strategic Research, ProViking project Systems Engineering and Computational Design (SECD).

References

[1] Dynasim. *Dymola*, http://www.dynasim.se/.

[2] INCOSE. *International Council on System Engineering*, http://www.incose.org.

[3] MathCore. *MathModelica*, http://www.mathcore.se/.

[4] *Modelica: A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification version 2.1*, Modelica Association, 2004

[5] *Semantic Web Community Portal*, http://www.semanticweb.org/.

[6] OMG. *Unified Modeling Language*, http://www.omg.org/uml.

[7] Modelica Community, http://www.modelica.org/

[8] Moldelica Libraries (Ontologies), http://www.modelica.org/library/

[9] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press, 2003, http://www.mathcore.com/drmodelica.

[10] Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson and Andreas Karstöm. *The Open Source Modelica Project*, in *Proceedings of The 2th International Modelica Conference,*March 18-19, 2002, Munich, Germany.

[11] Adrian Pop, Peter Fritzson. ModelicaXML: A Modelica XML representation with Applications, in International Modelica Conference,3-4 November, 2003, Linköping, Sweden, http://www.ida.liu.se/~adrpo/modelica/

[12] Adrian Pop, Ilie Savga, Uwe Assmann and Peter Fritzson. *Composition of XML dialects: A ModelicaXML case study*, in *Software Composition Workshop 2004, affiliated with ETAPS 2004*,3 April, 2004, Barcelona.

[13] Olof Johansson, Adrian Pop, Peter Fritzson, *A functionality Coverage Analysis of Industrially used Ontology Languages*, in *Model Driven Architecture: Foundations and Applications (MDAFA), 2004*, 10-11 June, 2004, Linköping, Sweden.

[14] Pim Borst, Hans Akkermans and Jan Top, *Engineering ontologies*, in Int. J. Human-Computer Studies, 1997, no. 46, p 365-406

[15] Adrian Pop, Olof Johansson and Peter Fritzson, *An integrated framework for model-driven product design and development using Modelica*, in Proceedings of the 45th Conference on Simulation and Modeling (SIMS), 23-24 September 2004, Copenhagen.

[16] Mogens Myrup Andreasen. *Machine Design Methods Based on a Systematic Approach (Syntesemetoder pa systemgrundlag)*, Lund Technical University, Lund, Sweden, 1980

[17] RosettaNet, http://www.rosettanet.org

[18] RosettaNet, *RosettaNet Technical Dictioanry*, http://www.rosettanet.org/technicaldictionary

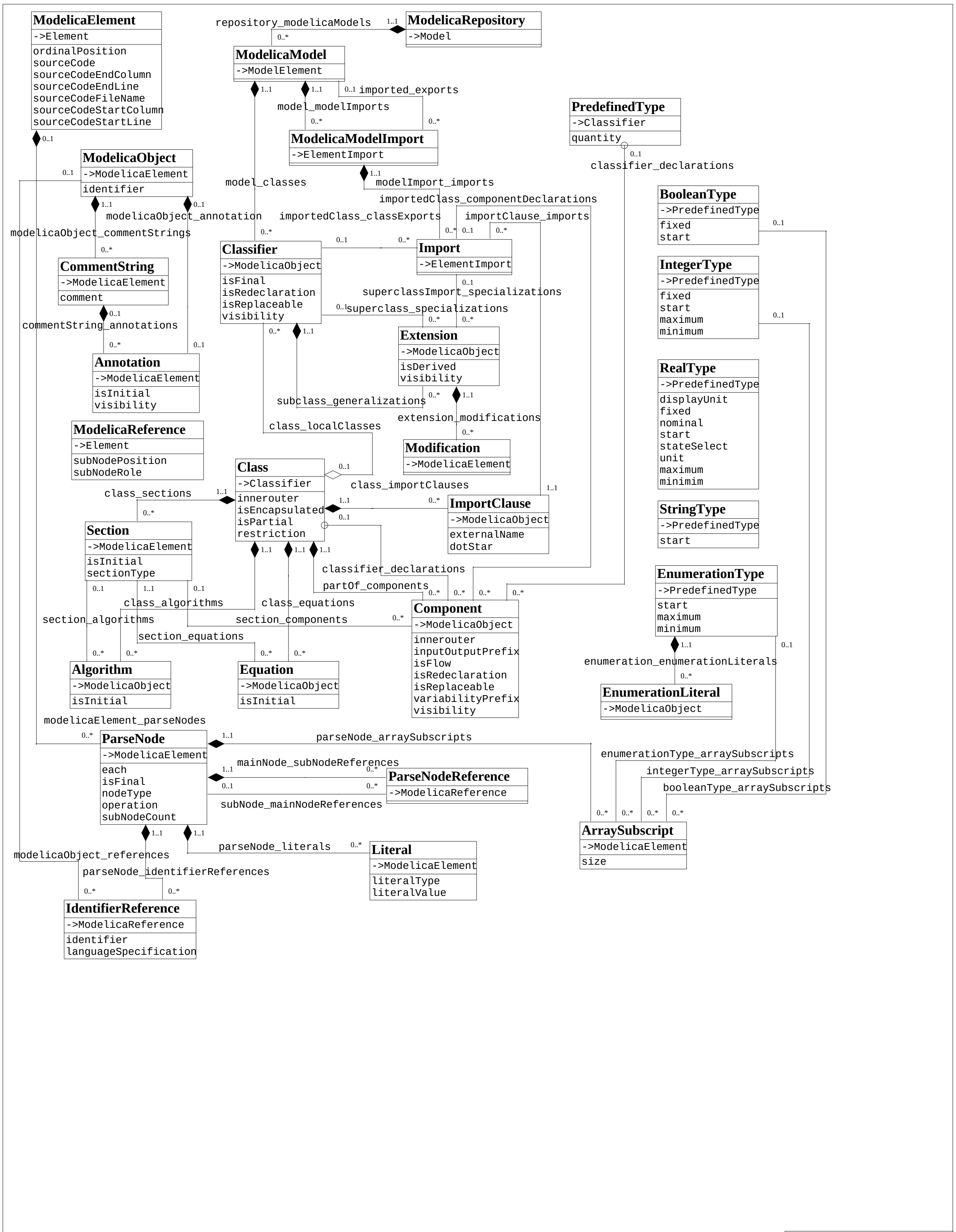
[19] Word Wide Web Consortium (W3C). Web Ontology Language (OWL), http://www.w3.org/TR/2003/CR-owl-features-20030818/

[20] Olof Johansson, ModelicaDB Project, http://www.modelica.org/projects/ModelicaDB/

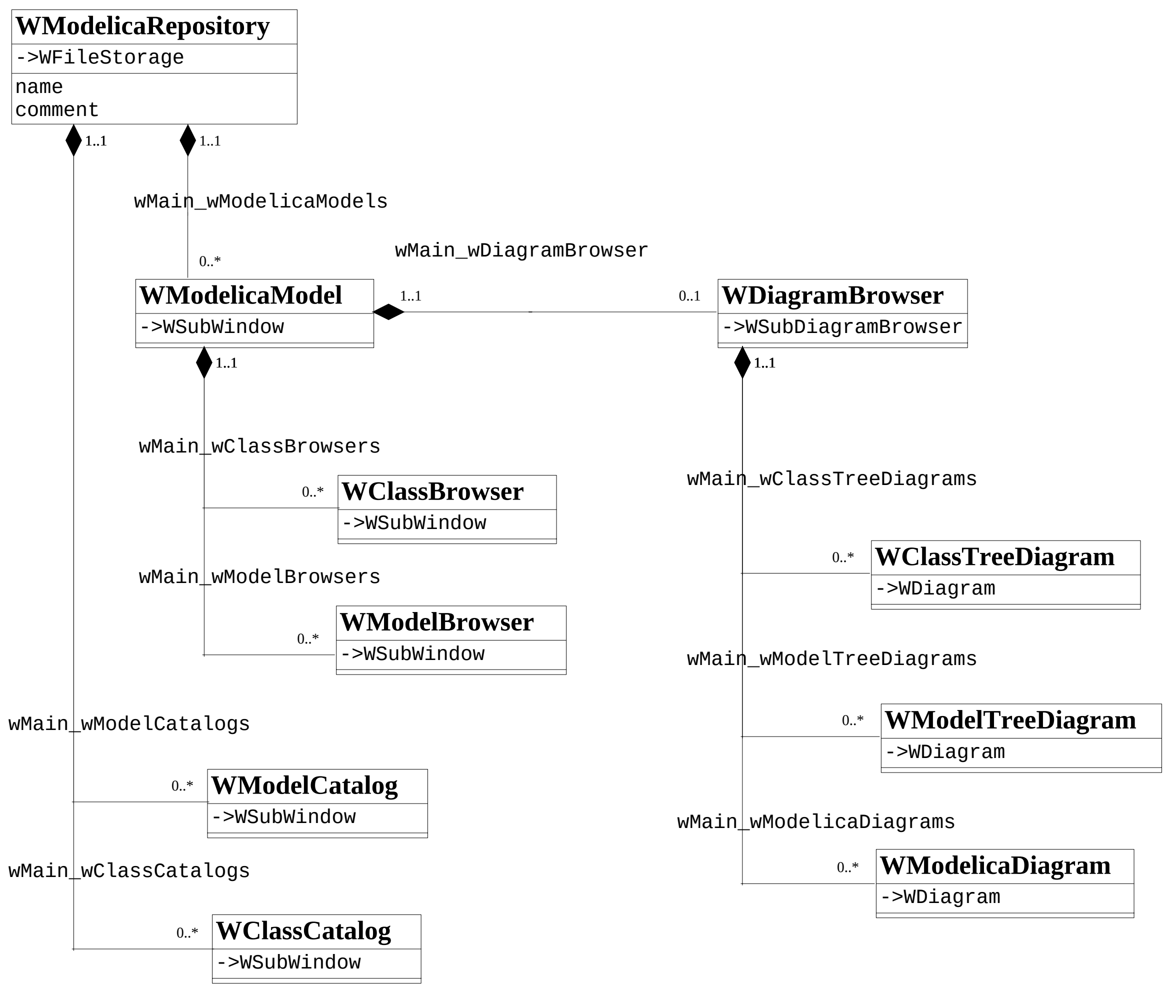
[21] Michael Tiller, Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications, in *International Modelica Conference,*3-4 November, 2003, Linköping, Sweden.

[22] IEC, IEC 61360, http://webstore.iec.ch for a fee

[23] ISO, ISO 31 Quantities and Units, Part 0-13. http://www.iso.org for a fee


ModelicaElement ->Element ordinalPosition sourceCode sourceCodeEndColumn sourceCodeEndLine sourceCodeFileName sourceCodeStartColumn sourceCodeStartLine
ModelicaRepository ->Model
repository_modelicaModels
ModelicaModel ->ModelElement
imported_exports
model_modelImports
ModelicaModelImport ->ElementImport
PredefinedType ->Classifier quantity
classifier_declarations
ModelicaObject ->ModelicaElement identifier
model_classes
modelImport_imports
importedClass_componentDeclarations
modelicaObject_annotation
importedClass_classExports
importClause_imports
modelicaObject_commentStrings
BooleanType ->PredefinedType fixed start
Classifier ->ModelicaObject isFinal isRedeclaration isReplaceable visibility
Import ->ElementImport
CommentString ->ModelicaElement comment
IntegerType ->PredefinedType fixed start maximum minimum
superclassImport_specializations
superclass_specializations
commentString_annotations
Extension ->ModelicaObject isDerived visibility
Annotation ->ModelicaElement isInitial visibility
RealType ->PredefinedType displayUnit fixed nominal start stateSelect unit maximum minimim
subclass_generalizations
extension_modifications
class_localClasses
ModelicaReference ->Element subNodePosition subNodeRole
Modification ->ModelicaElement
Class ->Classifier innerouter isEncapsulated isPartial restriction
class_importClauses
class_sections
ImportClause ->ModelicaObject externalName dotStar
StringType ->PredefinedType start
Section ->ModelicaElement isInitial sectionType
classifier_declarations
partOf_components
EnumerationType ->PredefinedType start maximum minimum
class_algorithms
class_equations
section_algorithms
section_components
Component ->ModelicaObject innerouter inputOutputPrefix isFlow isRedeclaration isReplaceable variabilityPrefix visibility
section_equations
enumeration_enumerationLiterals
Algorithm ->ModelicaObject isInitial
Equation ->ModelicaObject isInitial
EnumerationLiteral ->ModelicaObject
modelicaElement_parseNodes
ParseNode ->ModelicaElement each isFinal nodeType operation subNodeCount
parseNode_arraySubscripts
enumerationType_arraySubscripts
mainNode_subNodeReferences
ParseNodeReference ->ModelicaReference
integerType_arraySubscripts
booleanType_arraySubscripts
subNode_mainNodeReferences
ArraySubscript ->ModelicaElement size
parseNode_literals
Literal ->ModelicaElement literalType literalValue
modelicaObject_references
parseNode_identifierReferences
IdentifierReference ->ModelicaReference identifier languageSpecification

					Date Printed		
					2005-01-31 10:00:01		
	Domain Model	Filename	Date Added	Date Modified	#Cls	#Rel	#Att
DM.	ModelicaDB-21-10d	ModelicaDB-21-10d.odm	2003-03-04 08:03:07	2005-01-31 09:40:04	67	64	162
	Modelica 2.1 database design for ModelicaXML information exchange. The goal is a repository that can be round-tripped with Modelica source code.						
Dgm	2.0) Modelica High-Level Meta Objects		2004-04-27 14:36:52	2005-01-31 09:40:04	30	37	-
	Complete UML Model of Modelica 2.1.						



Date Printed
2005-01-31 09:59:20

	Domain Model	Filename	Date Added	Date Modified	#Cls	#Rel	#Att
DM.	appmodelicadb-21-10b	appmodelicadb-21-10b.odm	2004-11-05 21:08:37	2005-01-27 09:26:36	18	11	12
	Prototype application window structure for ModelicaDB V2.1x, including Profile support.						
Dgm	2.0) AppModelica		2004-11-05 21:19:14	2004-11-12 07:45:58	10	9	-
	Application windows for the Modelica module.						