Modelica Modeling Language (ModelicaML)
A UML Profile for Modelica

by

Wladimir Schamai
wladimir.schamai@eads.net

EADS Innovation Works (Hamburg, Germany)
Linköping University (Linköping, Sweden)

Technical reports in Computer and Information Science are available online at
Abstract: This document presents the Modelica Modeling Language (ModelicaML) a UML Profile for Modelica. The purpose of the ModelicaML is to enable an efficient and effective way to create, read or understand, and maintain Modelica models. ModelicaML is defined as a graphical notation and facilitates different views (composition, inheritance, behavior) on Modelica models. It is based on a subset of the OMG Unified Modeling Language (UML) and reuses concepts from the OMG Systems Modeling Language (SysML). ModelicaML is defined towards generation of executable Modelica code from graphical models.
# Table of Contents

1 Introduction ............................................................................................................................... 6
   1.1 Background .......................................................................................................................... 6
       1.1.1 OMG Unified and Systems Modeling Language (UML/SysML) .................................. 6
       1.1.2 The Modelica Language ............................................................................................ 7
       1.1.3 Object-Oriented Languages for Systems Modeling .................................................... 7
   1.2 Motivation .......................................................................................................................... 8
2 ModelicaML Introduction .......................................................................................................... 9
   2.1 ModelicaML in the Context of the UML and Modelica ..................................................... 9
   2.2 ModelicaML from Different Perspectives ......................................................................... 10
   2.3 List of UML Meta-Classes Used in ModelicaML ............................................................... 17
   2.4 Alternatives for Implementation of ModelicaML in UML Tools ....................................... 17
3 Application Example ............................................................................................................... 19
   3.1 Modelica Code Example ................................................................................................... 19
   3.2 ModelicaML Model Example ........................................................................................... 20
       3.2.1 Packages, Classes, Inheritance .................................................................................... 21
       3.2.2 Components, Variables, Ports ..................................................................................... 23
       3.2.3 Connections .................................................................................................................. 25
       3.2.4 Equations or Algorithm Statements .......................................................................... 25
       3.2.5 Asserts .......................................................................................................................... 26
       3.2.6 Conditional Equations or Algorithm Statements ....................................................... 27
       3.2.7 Initial Equations or Algorithm ....................................................................................... 30
       3.2.8 Hierarchical Modeling of Statements/Equations ......................................................... 30
       3.2.9 Calculation Models and Calculated Properties ............................................................ 30
       3.2.10 Explicit States or Modes ............................................................................................ 31
       3.2.11 Textual Requirements, Measure of Effectiveness ....................................................... 32
4 ModelicaML Profile - Full List Of Stereotypes ...................................................................... 36
   4.1 ModelicaClassConstructs ................................................................................................. 36
       4.1.1 Stereotype: «Block» ..................................................................................................... 36
       4.1.2 Stereotype: «CalculationModel» (is not a Modelica concept) ....................................... 36
       4.1.3 Stereotype: «Connector» .............................................................................................. 36
       4.1.4 Stereotype: «Function» ................................................................................................ 36
       4.1.5 Stereotype: «Model» .................................................................................................... 37
       4.1.6 Stereotype: «ModelicaClass» (is abstract) .................................................................... 37
       4.1.7 Stereotype: «ModelicaPackage» .................................................................................. 37
       4.1.8 Stereotype: «Record» .................................................................................................. 37
       4.1.9 Stereotype: «Type» ....................................................................................................... 37
   4.2 ModelicaCompositeConstructs ............................................................................................ 37
       4.2.1 Stereotype: «CalculatedProperty» (is not a Modelica concept) .................................... 37
       4.2.2 Stereotype: «Component» ........................................................................................... 38
       4.2.3 Stereotype: «Connection» ............................................................................................ 38
       4.2.4 Stereotype: «ConnectionPort» ..................................................................................... 38
       4.2.5 Stereotype: «FunctionArgument» ............................................................................... 38
       4.2.6 Stereotype: «InstanceOfAClass» (is abstract) ............................................................... 39
       4.2.7 Stereotype: «Variable» ................................................................................................ 39
   4.3 ModelicaBehaviorConstructs ............................................................................................... 39
       4.3.1 Stereotype: «Algorithm(Code)» .................................................................................. 39
       4.3.2 Stereotype: «AlgorithmLoopProperties» (is abstract) .................................................. 40
       4.3.3 Stereotype: «AlgorithmStatements(CodeInDiagram)» ................................................ 40
       4.3.4 Stereotype: «Assert» .................................................................................................... 40
       4.3.5 Stereotype: «ConditionalAlgorithm(Diagram)» .......................................................... 40
       4.3.6 Stereotype: «ConditionalEquations(Diagram)» .......................................................... 40
       4.3.7 Stereotype: «Equations(Code)» ................................................................................... 41
       4.3.8 Stereotype: «Equations(CodeInDiagram)» .................................................................. 41
       4.3.9 Stereotype: «EquationsLoopProperties» (is abstract) .................................................. 41
       4.3.10 Stereotype: «If» .......................................................................................................... 41
4.3.11 Stereotype: «InitialAlgorithm(Code)» ................................................................. 41
4.3.12 Stereotype: «InitialConditionalAlgorithm(Diagram)» ........................................ 41
4.3.13 Stereotype: «InitialConditionalEquations(Diagram)» ......................................... 42
4.3.14 Stereotype: «InitialEquations(Code)» ................................................................. 42
4.3.15 Stereotype: «ModelicaStateMachine» ................................................................. 42
4.3.16 Stereotype: «StateDependency» (is not a Modelica concept) ............................... 42
4.3.17 Stereotype: «When» ............................................................................................ 42
4.4 ModelicaRelationsConstructs .................................................................................. 42
4.4.1 Stereotype: «ExtendsRelation» ................................................................. 42
4.4.2 Stereotype: «TypeRelation» .............................................................................. 43
4.5 ModelicaRequirementConstructs ............................................................................. 43
4.5.1 Stereotype: «MeasureOfEffectiveness» (is not a Modelica concept) .................... 43
4.5.2 Stereotype: «Requirement» (is not a Modelica concept) ...................................... 43
4.5.3 Stereotype: «RequirementInstance» (is not a Modelica concept) ....................... 44
4.5.4 Stereotype: «RequirementRepresentative» (is not a Modelica concept) ............ 44
4.5.5 Stereotype: «Traceability» (is not a Modelica concept) ........................................ 44
4.6 ModelicaSimulationConstructs ................................................................................ 44
4.6.1 Stereotype: «CodeGeneration» (is not a Modelica concept) ............................... 44
4.6.2 Stereotype: «ExcludeFromCGIfUserInteractive» (is not a Modelica concept) ...... 44
4.6.3 Stereotype: «SimulateCommand» ...................................................................... 45
4.6.4 Stereotype: «UserInteractive» (is not a Modelica concept) ................................... 45
4.7 ModelicaAnnotationConstructs ............................................................................... 45
4.7.1 Stereotype: «Annotation(CodeGeneration)» ..................................................... 45
4.7.2 Stereotype: «Annotation(Documentation)» .......................................................... 45
4.7.3 Stereotype: «Annotation(SimulationExperiment)» ......................................... 45
4.7.4 Stereotype: «Annotation(VersionHandling)» .................................................... 46
4.7.5 Stereotype: «Annotation» .................................................................................. 46
4.8 ModelicaPredefinedTypes ....................................................................................... 46
4.8.1 Primitive Type: ModelicaBoolean ............................................................... 46
4.8.2 Primitive Type: ModelicaInteger ................................................................. 47
4.8.3 Primitive Type: ModelicaReal ................................................................. 47
4.8.4 Primitive Type: ModelicaString ................................................................. 47
5 Conclusion and Outlook ............................................................................................. 48
Table of Figures

Figure 1: Extract from the UML meta model ([2], p. 426) ...............................................................8
Figure 2: ModelicaML profile definition ..........................................................................................10
Figure 3: Domain-specific icons in ModelicaML ...........................................................................10
Figure 4: Example of a UML model in the model browser ............................................................21
Figure 5: Example of ModelicaML package and classes representation on a diagram ...............22
Figure 6: Example of inheritance ..................................................................................................22
Figure 7: Example of ModelicaML components representation .....................................................23
Figure 8: Example of ModelicaML variables representation ..........................................................24
Figure 9: Port example ...................................................................................................................24
Figure 10: UML Composite Diagram (similar to the Modelica Connection Diagram) ..................25
Figure 11: Example of Modelica code generated from the model ..................................................25
Figure 12: Example of non-conditional equations ..........................................................................26
Figure 13: Possibility to define loops ..............................................................................................26
Figure 14: Assert example ..............................................................................................................27
Figure 15: Example of condition algorithm statements .................................................................28
Figure 16: Example for Modelica "if" construct .............................................................................28
Figure 17: Conditions for "if" or "when" .......................................................................................29
Figure 18: Modelica code generated from an Activity Diagram ....................................................29
Figure 19: Example of conditional equations ..................................................................................30
Figure 20: Example of calculated properties ..................................................................................31
Figure 21: Example of explicit states of the tank .........................................................................32
Figure 22: Example of explicit states or modes of the controller ...................................................32
Figure 23: Example of textual requirements ...................................................................................33
Figure 24: Example of requirements behavior ...............................................................................34
Figure 25: Example of binding requirements to a design model ...................................................34
Figure 26: Example of requirements evaluation during system simulation ..................................35
1 Introduction

UML [2], SysML [4] and Modelica [1] are object-oriented modeling languages. They both provide means for representing a system as a classified object and to describe its internal structure and behavior. UML/SysML facilitates efficient capturing of relevant system requirements, -design or -test data by means of graphical formalisms, crosscutting Constructs and views (diagrams) on the model-data. Modelica is defined as a textual language with standardized graphical annotations for model icons, and is designed for efficient simulation of system dynamic behavior.

This document is a further step towards the application of the Model-Based Systems Engineering (MBSE) paradigm, using standardized, graphical, and executable system modeling languages. It presents further development of the ModelicaML language which enables an integrated modeling and simulation of system requirements and design (including hardware and software). This approach combines the power of the OMG UML/SysML standardized graphical notation for system and software modeling and the modeling and simulation power of Modelica. It facilitates creation of executable system-specification and -analysis models that can simulate both the time-discrete and the time-continuous behavior of a system.

For reading this document a basic knowledge about the Modelica language [1], the UML [2] and SysML [4] is required.

This document presents the ModelicaML and provides an example of its application. However, neither the validation of ModelicaML models nor the Modelica code generation1 from ModelicaML models are discussed here.

1.1 Background

Some research work done previously [22],[21] already identified the need for integrating UML/SysML and Modelica, and has addressed integration issues to some extent. For example, [7] has identified the basic mapping of the structural concepts of Modelica to SysML. It was also pointed out that the SysML Parametrics concept is not sufficient for modeling equation-based behavior of a class. By contrast, [9] leverages the SysML Parametrics concept for integration of continuous-time behavior into SysML models. [8] presents a concept to use SysML for integrating models of continuous-time dynamic system behavior with SysML information models representing systems engineering problems, and provides rules for graph-based bidirectional transformation of SysML and Modelica models.

The main focus of this paper is the representation of Modelica concepts using graphical notation that is based on a subset UML/SysML, which can be translated into executable Modelica code.

The graphical modeling capabilities of Modelica tools, currently available on the market, are limited to one diagram type: Connection Diagram. This diagram shows Modelica components (instances of classes) and their interconnection (connection clauses). In contrast, UML/SysML defines a comprehensive set of diagrams and cross-cutting relationships for the modeling of different views on the system data, such as requirements, logical or physical structure, architecture, and behavior. However, the UML/SysML are general purpose graphical languages. They do not define any executable action language and suffer of loosely defined semantics.

1.1.1 OMG Unified and Systems Modeling Language (UML/SysML)

UML [2] and SysML [4] are general-purpose modeling languages that enable engineers to create and manage models of engineered systems using graphical notation. SysML is based on the UML. It reuses a subset of UML 2.1 and extends this by adding new modeling elements and two new

---

1 There is a prototype implementation of a ModelicaML2Modelica code generator.
diagram types. Through these extensions, SysML is capable of representing the specification, analysis, design, verification, and validation of any engineered system. MBSE promotes the usage of models as primary engineering artifacts. However, textual requirements are still the main vehicle for communicating and agreeing on system specification in a system development process. SysML provides mechanisms to include textual requirements into models. In doing so, traceability of textual requirements to design artifacts and test cases is facilitated. The logical behavior of systems is adequately captured in SysML through a combination of activity diagrams, state machine diagrams, and/or sequence diagrams. With the recent adoption of the Foundational Subset of UML specification, SysML activity diagrams can be executed to support discrete event simulation in a standard way. In addition, SysML includes Parametrics to support the execution of constraint-based behavior such as continuous-time dynamics in terms of energy flow. However, the syntax and semantics of such behavioral descriptions in Parametrics have been left unspecified to interoperate with other simulation and analysis modeling capabilities.

1.1.2 The Modelica Language

Modelica is an object-oriented modeling language for describing ordinary and differential algebraic equation (OAE and DAE) systems combined with discrete events, so-called hybrid DAEs. Such models are ideally suited for representing physical behavior and the exchange of energy, signals, or other continuous-time interactions between system components. Modelica models are similar in structure to SysML models in the sense that Modelica models consist of compositions of sub-models connected by ports that represent energy flow (undirected) or signal flow (directed). The models are acausal, equation-based, and declarative. The Modelica language is defined and maintained by the Modelica Association [14] which publishes a formal specification [1] but also provides an extensive Modelica Standard Library that includes a broad foundation of essential models covering domains ranging from (analog and digital) electrical systems, mechanical motion and thermal systems, to block diagrams for control. Finally, it is worth noting that there are several efforts within the Modelica community to develop open-source solvers, such as in the OpenModelica project [12].

1.1.3 Object-Oriented Languages for Systems Modeling

UML[2], SysML[4] and Modelica[1] are object-oriented languages and follow the object-oriented paradigm, such as encapsulation of object structure, data and behavior, which is essential for representing systems using models.

The class concept is the basic structural unit in Modelica. Modelica classes provide the structure for objects and contain equations, that ultimately are the basis for the executable simulation code. The most general kind of class is “model”. Specialized categories of classes such as “record”, “type”, “block”, “package”, “function” and “connector” have most of the properties of a “model” but with restrictions or enhancements.

The behavior of a Modelica Class is defined by its (mathematical) equations and statements (same as assignments in usual programming languages). Equations are located in the “equation” section and statements are located in the “algorithm” section. A Modelica class such as model, block or function, can have equation or algorithm sections in accordance with the restrictions defined.

In addition to atomic equations and statements, Modelica defines conditional constructs, which are allowed in both the equation section and the algorithm section. Furthermore, it defines which of the constructs can have nested constructs.

Constructs allowed in a Modelica algorithm section:
- for-statement (can have nested conditional constructs)
- while-statement (can have nested conditional constructs)
- if-statement (can have nested conditional constructs)
- when-statement (can have nested conditional constructs)
- simple-statement (cannot have nested conditional constructs)

Constructs allowed in a Modelica equation section:
- for-equation (can have nested conditional constructs)
- if-equation (can have nested conditional constructs)
- when-equation (cannot have nested constructs)
- simple-equation (cannot have nested conditional constructs)

The main structural unit of the UML, which can have behavior, is the UML Class (or in SysML the Block). A non-behavioral concept is the DataType. The UML defines multiple types of behavior that a BehavioredClassifier (a Class) can have (see Figure 1).

![Figure 1: Extract from the UML meta model ([2], p. 426)](image)

In conclusion, the UML, SysML and Modelica language are suitable for the modeling of systems and can represent both:
- The structure of a system (composition, inheritance, instantiation of classes and interconnection of components)
- The behavior of a system or its component (in UML/SysML: Class/Block Operations, State Machine or Activity Diagram which invoke block operations, in Modelica: Equations, algorithm statements)

1.2 Motivation

By integrating Modelica and UML/SysML, UML/SysML's strength in graphical and descriptive modeling is complemented with Modelica's formal executable modeling for analyses and trade studies. Vice versa, Modelica will benefit from using the selected subset of the UML/SysML graphical notation (visual formalisms) for editing and maintaining Modelica models.

Graphical modeling, as promoted by the OMG [13], promises to be more effective and efficient, regarding editing, human reader perception of models, and maintaining models compared to a traditional textual representation. A unified, standardized graphical notation for system modeling and simulation will facilitate common understanding of models for parties involved in development of systems (i.e., system-engineers, -designers, -testers, software-developers, customers or stakeholder).

Existing UML/SysML formalisms are typically translated into (and limited to) time-discrete or event-based simulations of a system or software. This limitation will disappear when Modelica comes into play. UML/SysML models will then be of a higher expressiveness and correctness, because they will become executable while covering simulation of hardware and software, with integrated continuous/discrete-time or event-based behavior.
2 ModelicaML Introduction

The main purpose of the ModelicaML is to enable an efficient and effective way to create, read or understand, and maintain Modelica models. ModelicaML is defined as a graphical notation and facilitates different views (composition, inheritance, behavior) on Modelica models. It is based on a subset of the OMG Unified Modeling Language (UML) and reuses concepts from the OMG Systems Modeling Language (SysML). ModelicaML is designed towards generation of executable Modelica code from graphical models.

UML/SysML provide the modeler with powerful descriptive constructs at the expense of loosely defined semantics that are marked as “semantic variation points” in the UML/SysML specifications. The intention of the ModelicaML is to provide the modeler with powerful executable constructs and execution semantics that are based on the Modelica language. Therefore, ModelicaML uses a limited set of the UML, extends the UML meta-model (using the UML profiling mechanism) with new constructs in order to introduce missing Modelica concepts and reuses concepts from the SysML. However, as the UML and SysML, ModelicaML is only a graphical notation. ModelicaML models are eventually translated into Modelica code. Hence, the execution semantics are defined by the Modelica language and ultimately by a Modelica compiler, that will translate the generated Modelica code into an executable form.

This chapter introduces the ModelicaML by providing the basics of the mapping between Modelica and UML and presenting ModelicaML from both: A perspective of a user that is familiar with Modelica and a perspective of a user that is familiar with UML/SysML.

Since the ModelicaML profile is an extension of the UML meta-model it can be used for both: Modeling with the standard UML and with the SysML.

2.1 ModelicaML in the Context of the UML and Modelica

ModelicaML is implemented as a UML Profile. It defines extensions of the UML meta-model, that are necessary for capturing Modelica constructs. UML Profiles (and Stereotypes) enable an extension of the UML meta-model in order to introduce concepts that are missing in UML or that are domain-specific. The full list of the UML extensions is presented in chapter 4. Furthermore, from the UML perspective, Modelica is a specific language. For integrating specific languages the UML defines opaque constructs, that are used for the definition of ModelicaML as well.

The term “meta-model” is used in the graphical modeling world as a synonym for the term “grammar” used in the programming languages world. As any grammar, a meta-model defines language concepts (classes) and interrelations between them.

The grammar of the Modelica language is described using the EBNF. Since ModelicaML is based on the UML meta-model, the “grammar” (meta-model) of the ModelicaML is that from the UML with the defined extensions. It is described using MOF (the underlying formalism for describing the UML meta-model itself), as illustrated in Figure 2. The translation of the graphical ModelicaML model into valid Modelica code is defined by code generation algorithms (that are not addressed in this document).

There is a difference between the graphical representation provided by the current Modelica tools and the ModelicaML notation. Modelica tools provide the modeler with one diagram, the Connection Diagram, for graphical modeling. A Connection Diagram shows Modelica class components (typically depicted as domain specific icons with connectors) of the class and their interconnection (connect clauses). The graphical notation is defined by the Modelica modeler (e.g.

---

2 SysML itself is also a UML Profile. All stereotypes that extend UML meta-classes are applicable to the SysML as well.
the developer of a library) and is not standardized by the language specification; it is usually specific to the domain of application.

![Image](90x485 to 505x734)

The UML defines how the concepts (e.g. class, a kind of behavior, etc.) defined at the meta-model ("grammar"), are represented graphically. Moreover, it defines different types of diagrams, that enable different graphical views on the user-model data (such as a UML class diagram which can show inheritance, classes that are nested, the composition of a class, or a UML composite diagram that shows the interconnection of components of a class).

Hence, the ModelicaML graphical notation is defined at the meta-level (grammar of the language). It defines how the Modelica language concepts (e.g. a model, function, connection, etc.) are represented graphically. Thanks to this abstract, unified notation, engineers from different domains or disciplines will share a common understanding of graphical models. Icons, specific to a domain of application, can still be included into a dedicated class compartment as illustrated in the following figure.

![Image](179x226 to 416x305)

Figure 2: ModelicaML profile definition

By contrast, the UML defines how the concepts (e.g. class, a kind of behavior, etc.) defined at the meta-model ("grammar"), are represented graphically. Moreover, it defines different types of diagrams, that enable different graphical views on the user-model data (such as a UML class diagram which can show inheritance, classes that are nested, the composition of a class, or a UML composite diagram that shows the interconnection of components of a class).

Hence, the ModelicaML graphical notation is defined at the meta-level (grammar of the language). It defines how the Modelica language concepts (e.g. a model, function, connection, etc.) are represented graphically. Thanks to this abstract, unified notation, engineers from different domains or disciplines will share a common understanding of graphical models. Icons, specific to a domain of application, can still be included into a dedicated class compartment as illustrated in the following figure.

![Image](206x206)

Figure 3: Domain-specific icons in ModelicaML

### 2.2 ModelicaML from Different Perspectives

ModelicaML is intended to be used by both:

- Modelers who are familiar with the UML/SysML and would like to take advantage of the executable semantics of ModelicaML for building models that can be simulated,
- And modelers who are used to Modelica and would like to take advantage of the graphical modeling instead of the traditional text-based approach for creating Modelica code.
The following tables introduce the ModelicaML profile from a different perspective. For example:

- A user who is familiar with the UML, needs to know what the extensions (additional stereotypes and properties) of the UML are that should be applied? As mentioned above, the meaning (semantic) of these new stereotypes is still given by the Modelica language, which means that the UML modeler will have to get familiar with the Modelica language.

- In turn, a Modelica user needs to know which UML elements need to be used when a specific Modelica element\(^3\) should be represented, where define specific Modelica keywords can be defined, or how to define, for example, instance modifications or the actual equations.

The full list of the UML extension together with an application example is presented in the following chapters. The tables below should help the ModelicaML users to find the right answers\(^4\).

---

\(^3\) It is subject to the current implementation work of the ModelicaML editor to reflect the Modelica language wording, so that the Modelica modeler will not have to work with the UML/SysML wording.

\(^4\) The lists are sorted alphabetically.
Which ModelicaML stereotypes can I apply to which UML meta-classes?

<table>
<thead>
<tr>
<th>To a UML ...</th>
<th>you can apply the ModelicaML stereotype ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>«ConditionalAlgorithm(Diagram)»</td>
</tr>
<tr>
<td>Activity</td>
<td>«InitialConditionalAlgorithm(Diagram)»</td>
</tr>
<tr>
<td>Activity</td>
<td>«ConditionalEquations(Diagram)»</td>
</tr>
<tr>
<td>Activity</td>
<td>«InitialConditionalEquations(Diagram)»</td>
</tr>
<tr>
<td>Class</td>
<td>«Block»</td>
</tr>
<tr>
<td>Class</td>
<td>«Model»</td>
</tr>
<tr>
<td>Class</td>
<td>«CalculationModel»</td>
</tr>
<tr>
<td>Class</td>
<td>«Requirement»</td>
</tr>
<tr>
<td>Comment</td>
<td>«Annotation(Documentation)»</td>
</tr>
<tr>
<td>Comment</td>
<td>«Annotation(CodeGeneration)»</td>
</tr>
<tr>
<td>Comment</td>
<td>«Annotation(SimulationExperiment)»</td>
</tr>
<tr>
<td>Comment</td>
<td>«Annotation»</td>
</tr>
<tr>
<td>Comment</td>
<td>«Annotation(VersionHandling)»</td>
</tr>
<tr>
<td>Connector</td>
<td>«Connection»</td>
</tr>
<tr>
<td>Constraint</td>
<td>«Assert»</td>
</tr>
<tr>
<td>ControlFlow</td>
<td>«StateDependency»</td>
</tr>
<tr>
<td>DataType</td>
<td>«Connector»</td>
</tr>
<tr>
<td>DataType</td>
<td>«Record»</td>
</tr>
<tr>
<td>DataType</td>
<td>«Type»</td>
</tr>
<tr>
<td>DataType</td>
<td>«SimulateCommand»</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>«If»</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>«When»</td>
</tr>
<tr>
<td>FunctionBehavior</td>
<td>«Function»</td>
</tr>
<tr>
<td>Generalization</td>
<td>«TypeRelation»</td>
</tr>
<tr>
<td>Generalization</td>
<td>«ExtendsRelation»</td>
</tr>
<tr>
<td>NamedElement</td>
<td>«Traceability»</td>
</tr>
<tr>
<td>OpaqueAction</td>
<td>«Equations(CodeInDiagram)»</td>
</tr>
</tbody>
</table>
Which UML elements should I create in order to represent a Modelica ... ?

<table>
<thead>
<tr>
<th>In order to create a Modelica ...</th>
<th>create a UML ...</th>
<th>and apply the ModelicaML Stereotype ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm(Code)</td>
<td>OpaqueBehavior</td>
<td>«Algorithm(Code)&gt;&gt;</td>
</tr>
<tr>
<td>AlgorithmStatements(CodeInDiagram)</td>
<td>OpaqueAction</td>
<td>«AlgorithmStatements(CodeInDiagram)&gt;&gt;</td>
</tr>
<tr>
<td>Annotation</td>
<td>Comment</td>
<td>«Annotation»</td>
</tr>
<tr>
<td>Annotation(CodeGeneration)</td>
<td>Comment</td>
<td>«Annotation(CodeGeneration)&gt;&gt;</td>
</tr>
<tr>
<td>Annotation(Documentation)</td>
<td>Comment</td>
<td>«Annotation(Documentation)&gt;&gt;</td>
</tr>
<tr>
<td>Annotation(SimulationExperiment)</td>
<td>Comment</td>
<td>«Annotation(SimulationExperiment)&gt;&gt;</td>
</tr>
<tr>
<td>Annotation(VersionHandling)</td>
<td>Comment</td>
<td>«Annotation(VersionHandling)&gt;&gt;</td>
</tr>
<tr>
<td>Assert</td>
<td>Constraint</td>
<td>«Assert»</td>
</tr>
<tr>
<td>Block</td>
<td>Class</td>
<td>«Block»</td>
</tr>
<tr>
<td>Component</td>
<td>Property</td>
<td>«Component»</td>
</tr>
<tr>
<td>ConditionalAlgorithm(Diagram)</td>
<td>Activity</td>
<td>«ConditionalAlgorithm(Diagram)&gt;&gt;</td>
</tr>
<tr>
<td>ConditionalEquations(Diagram)</td>
<td>Activity</td>
<td>«ConditionalEquations(Diagram)&gt;&gt;</td>
</tr>
<tr>
<td>Connection</td>
<td>Connector</td>
<td>«Connection»</td>
</tr>
<tr>
<td>ConnectionPort</td>
<td>Port</td>
<td>«ConnectionPort»</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>------------------</td>
</tr>
<tr>
<td>Connector</td>
<td>Port</td>
<td>«Connector»</td>
</tr>
<tr>
<td>Equations(Code)</td>
<td>OpaqueBehavior</td>
<td>«Equations(Code)»</td>
</tr>
<tr>
<td>Equations(CodeInDiagram)</td>
<td>OpaqueAction</td>
<td>«Equations(CodeInDiagram)»</td>
</tr>
<tr>
<td>ExtendsRelation</td>
<td>Generalization</td>
<td>«ExtendsRelation»</td>
</tr>
<tr>
<td>Function</td>
<td>FunctionBehavior</td>
<td>«Function»</td>
</tr>
<tr>
<td>FunctionArgument</td>
<td>Parameter</td>
<td>«FunctionArgument»</td>
</tr>
<tr>
<td>If</td>
<td>DecisionNode</td>
<td>«If»</td>
</tr>
<tr>
<td>InitialAlgorithm(Code)</td>
<td>OpaqueBehavior</td>
<td>«InitialAlgorithm(Code)»</td>
</tr>
<tr>
<td>InitialConditionalAlgorithm(Diagram)</td>
<td>Activity</td>
<td>«InitialConditionalAlgorithm(Diagram)»</td>
</tr>
<tr>
<td>InitialConditionalEquations(Diagram)</td>
<td>Activity</td>
<td>«InitialConditionalEquations(Diagram)»</td>
</tr>
<tr>
<td>InitialEquations(Code)</td>
<td>OpaqueBehavior</td>
<td>«InitialEquations(Code)»</td>
</tr>
<tr>
<td>Model</td>
<td>Class</td>
<td>«Model»</td>
</tr>
<tr>
<td>Package</td>
<td>Package</td>
<td>«ModelicaPackage»</td>
</tr>
<tr>
<td>Record</td>
<td>Data</td>
<td>«Record»</td>
</tr>
<tr>
<td>SimulateCommand</td>
<td>Data</td>
<td>«SimulateCommand»</td>
</tr>
<tr>
<td>StateMachine</td>
<td>StateMachine</td>
<td>«ModelicaStateMachine»</td>
</tr>
<tr>
<td>Type</td>
<td>Data</td>
<td>«Type»</td>
</tr>
<tr>
<td>TypeRelation</td>
<td>Generalization</td>
<td>«TypeRelation»</td>
</tr>
<tr>
<td>Variable</td>
<td>Property</td>
<td>«Variable»</td>
</tr>
<tr>
<td>When</td>
<td>DecisionNode</td>
<td>«When»</td>
</tr>
</tbody>
</table>

Where can I set the Modelica keyword ... ?

<table>
<thead>
<tr>
<th>In order to set the Modelica keyword ...</th>
<th>apply the ModelicaML stereotype ...</th>
<th>to a UML ...</th>
<th>and use the attribute ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>'and' or 'or' or 'not'</td>
<td>«StateDependency»</td>
<td>ControlFlow</td>
<td>logicalOperator (of the «StateDependency» stereotype)</td>
</tr>
<tr>
<td>'continuous' or 'discrete' or 'parameter' or 'constant'</td>
<td>«Variable»</td>
<td>Property</td>
<td>variability (of the «Variable» stereotype)</td>
</tr>
<tr>
<td>'continuous' or 'discrete' or 'parameter' or 'constant'</td>
<td>«FunctionArgument»</td>
<td>Parameter</td>
<td>variability (of the «FunctionArgument» stereotype)</td>
</tr>
<tr>
<td>'Evaluate'</td>
<td>«Annotation(CodeGeneration)&gt;&gt;</td>
<td>Comment</td>
<td>Evaluate (of the «Annotation(CodeGeneration)&gt;&gt; stereotype)</td>
</tr>
<tr>
<td>'expandable'</td>
<td>«Connector&gt;&gt;</td>
<td>DataType</td>
<td>expandable (of the «Connector» stereotype)</td>
</tr>
<tr>
<td>'flow' or 'stream'</td>
<td>«Variable&gt;&gt;</td>
<td>Property</td>
<td>flowFlag (of the «Variable» stereotype)</td>
</tr>
<tr>
<td>'flow' or 'stream'</td>
<td>«FunctionArgument&gt;&gt;</td>
<td>Parameter</td>
<td>flowFlag (of the «FunctionArgument» stereotype)</td>
</tr>
<tr>
<td>'generateInteractiveSimInitFile'</td>
<td>«CodeGeneration&gt;&gt;</td>
<td>Class</td>
<td>generateInteractiveSimInitFile (of the «CodeGeneration» stereotype)</td>
</tr>
<tr>
<td>'HideResult'</td>
<td>«Annotation(CodeGeneration)&gt;&gt;</td>
<td>Comment</td>
<td>HideResult (of the «Annotation(CodeGeneration)&gt;&gt; stereotype)</td>
</tr>
<tr>
<td>'Inline'</td>
<td>«Annotation(CodeGeneration)&gt;&gt;</td>
<td>Comment</td>
<td>Inline (of the «Annotation(CodeGeneration)&gt;&gt; stereotype)</td>
</tr>
<tr>
<td>'inner' or 'outer'</td>
<td>«Variable&gt;&gt;</td>
<td>Property</td>
<td>scope (of the «Variable» stereotype)</td>
</tr>
<tr>
<td>'inner' or 'outer'</td>
<td>«FunctionArgument&gt;&gt;</td>
<td>Parameter</td>
<td>scope (of the «FunctionArgument» stereotype)</td>
</tr>
<tr>
<td>'input' or 'output'</td>
<td>«ConnectionPort&gt;&gt;</td>
<td>Port</td>
<td>causality (of the «ConnectionPort» stereotype)</td>
</tr>
<tr>
<td>'input' or 'output'</td>
<td>«Variable&gt;&gt;</td>
<td>Property</td>
<td>causality (of the «Variable» stereotype)</td>
</tr>
<tr>
<td>'input' or 'output'</td>
<td>«FunctionArgument&gt;&gt;</td>
<td>Parameter</td>
<td>causality (of the «FunctionArgument» stereotype)</td>
</tr>
<tr>
<td>'isSubActivity'</td>
<td>«ConditionalAlgorithm(Diagram)&gt;&gt;</td>
<td>Activity</td>
<td>isSubActivity (of the «ConditionalAlgorithm(Diagram)&gt;&gt; stereotype)</td>
</tr>
<tr>
<td>'isSubActivity'</td>
<td>«ConditionalEquations(Diagram)&gt;&gt;</td>
<td>Activity</td>
<td>isSubActivity (of the «ConditionalEquations(Diagram)&gt;&gt; stereotype)</td>
</tr>
<tr>
<td>'LateInline'</td>
<td>«Annotation(CodeGeneration)&gt;&gt;</td>
<td>Comment</td>
<td>LateInline (of the «Annotation(CodeGeneration)&gt;&gt; stereotype)</td>
</tr>
<tr>
<td>'public' or 'protected'</td>
<td>«Component&gt;&gt;</td>
<td>Property</td>
<td>visibility (of the «Component» stereotype)</td>
</tr>
<tr>
<td>'public' or 'protected'</td>
<td>«Variable&gt;&gt;</td>
<td>Property</td>
<td>visibility (of the «Variable» stereotype)</td>
</tr>
<tr>
<td>'public' or 'protected'</td>
<td>«FunctionArgument&gt;&gt;</td>
<td>Parameter</td>
<td>visibility (of the «FunctionArgument» stereotype)</td>
</tr>
<tr>
<td>'warning' or 'error'</td>
<td>«Assert&gt;&gt;</td>
<td>Constraint</td>
<td>level (of the «Assert» stereotype)</td>
</tr>
</tbody>
</table>

**Where can I define ... ?**

<table>
<thead>
<tr>
<th>You can define a ... using the attribute (with the same name) ...</th>
<th>of the ModelicaML stereotype ...</th>
<th>that is applied to a UML ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition' or 'message'</td>
<td>«Assert»</td>
<td>Constraint</td>
</tr>
<tr>
<td>declarationEquation' or 'instanceModification' or 'conditionalExpression' or 'instanceArraySize'</td>
<td>«Variable»</td>
<td>Property</td>
</tr>
<tr>
<td>declarationEquation' or 'instanceModification' or 'conditionalExpression' or 'instanceArraySize'</td>
<td>«FunctionArgument&gt;&gt;</td>
<td>Parameter</td>
</tr>
<tr>
<td>forIndices'</td>
<td>«Equations(Code)&gt;&gt;</td>
<td>OpaqueBehavior</td>
</tr>
<tr>
<td>forIndices'</td>
<td>«InitialEquations(Code)&gt;&gt;</td>
<td>OpaqueBehavior</td>
</tr>
<tr>
<td>forIndices'</td>
<td>«ConditionalEquations(Diagram)&gt;&gt;</td>
<td>Activity</td>
</tr>
<tr>
<td>forIndices'</td>
<td>«InitialConditionalEquations(Diagram)&gt;&gt;</td>
<td>Activity</td>
</tr>
<tr>
<td>forIndices'</td>
<td>«Equations(CodeInDiagram)&gt;&gt;</td>
<td>OpaqueAction</td>
</tr>
<tr>
<td>forIndices' or 'whileExpression'</td>
<td>«Algorithm(Code)&gt;&gt;</td>
<td>OpaqueBehavior</td>
</tr>
<tr>
<td>forIndices' or 'whileExpression'</td>
<td>«InitialAlgorithm(Code)&gt;&gt;</td>
<td>OpaqueBehavior</td>
</tr>
<tr>
<td>forIndices' or 'whileExpression'</td>
<td>«ConditionalAlgorithm(Diagram)&gt;&gt;</td>
<td>Activity</td>
</tr>
<tr>
<td>forIndices' or 'whileExpression'</td>
<td>«InitialConditionalAlgorithm(Diagram)&gt;&gt;</td>
<td>Activity</td>
</tr>
<tr>
<td>forIndices' or 'whileExpression'</td>
<td>«AlgorithmStatements(CodeInDiagram)&gt;&gt;</td>
<td>OpaqueAction</td>
</tr>
<tr>
<td>forIndices' or 'whileExpression'</td>
<td>«ModelicaStateMachine»</td>
<td>StateMachine</td>
</tr>
<tr>
<td>fullAnnotationString'</td>
<td>«Annotation»</td>
<td>Comment</td>
</tr>
<tr>
<td>info' or 'revisions'</td>
<td>«Annotation(Documentation)&gt;&gt;</td>
<td>Comment</td>
</tr>
<tr>
<td>instanceModification' or 'conditionalExpression' or 'instanceArraySize'</td>
<td>«Component&gt;&gt;</td>
<td>Property</td>
</tr>
<tr>
<td>instanceModification' or 'conditionalExpression' or 'instanceArraySize'</td>
<td>«ConnectionPort»</td>
<td>Port</td>
</tr>
<tr>
<td>shortClassDefinition'</td>
<td>«Type»</td>
<td>DataType</td>
</tr>
<tr>
<td>typeModification'</td>
<td>«ExtendsRelation»</td>
<td>Generalization</td>
</tr>
<tr>
<td>typeModification' or 'typeArraySize'</td>
<td>«TypeRelation»</td>
<td>Generalization</td>
</tr>
<tr>
<td>version' or 'conversion' or 'uses'</td>
<td>«Annotation(VersionHandling)&gt;&gt;</td>
<td>Comment</td>
</tr>
</tbody>
</table>
2.3 List of UML Meta-Classes Used in ModelicaML

The ModelicaML Profile defines the necessary extensions of the UML meta-model. However, some of the UML meta-classes are used without any extension.

The following UML meta-classes (their interrelations are implicit and are not listed here) from [2] can be used in ModelicaML models\(^5\) and are eventually translated into Modelica code. Most of them are extended by the ModelicaML Profile in order to facilitate the capturing of required Modelica constructs. Some are not extended and are used as defined by the UML (e.g. InitialNode, ControlFlow, Enumeration, Comment etc.). The numbers in the title-rows correspond to the chapter-numbers in [2].

<table>
<thead>
<tr>
<th>7 Classes</th>
<th>11 Actions</th>
<th>13 Common Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>CallBehaviorAction</td>
<td>FunctionBehavior</td>
</tr>
<tr>
<td>Comment</td>
<td>OpaqueAction</td>
<td>OpaqueBehavior</td>
</tr>
<tr>
<td>Constraint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataType</td>
<td>12 Activities</td>
<td></td>
</tr>
<tr>
<td>Enumeration</td>
<td>Activity</td>
<td></td>
</tr>
<tr>
<td>Generalization</td>
<td>ControlFlow</td>
<td></td>
</tr>
<tr>
<td>Package</td>
<td>DecisionNode</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>InitialNode</td>
<td></td>
</tr>
<tr>
<td>9 Composite Structures</td>
<td>15 State Machines</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>ConnectionPointReference</td>
<td></td>
</tr>
<tr>
<td>Connector</td>
<td>Pseudostate</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Region</td>
<td></td>
</tr>
<tr>
<td>Port</td>
<td>State</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>StateMachine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transition</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: UML meta-classes used in ModelicaML

2.4 Alternatives for Implementation of ModelicaML in UML Tools

There are different possibilities for implementing the UML extensions, defined by the ModelicaML, in a UML modeling tool.

**Definition of a ModelicaML profile for a UML Tool (user-model level)**

This way the modeler needs to first create a UML element and then apply a stereotype, defined in the profile, in order to represent a specific concept or to introduce (or to precise) the semantics. **Advantage:** This approach will allow creating or reading ModelicaML models using any UML tool. **Disadvantage:** The modeling tool does not directly reflect the Modelica wording\(^6\). The modeler needs to have a basic knowledge of the UML/SysML in order to be able to apply the stereotypes of the ModelicaML profile appropriately. All limitations, constraints and possible inconsistencies will have to be checked and resolved before the Modelica code generation.

**Extension of the UML meta-model and building a new modeling tool (meta-model level)**

The modeling tool is build upon the extended UML meta-model (e.g. see the implementation of the Topcased [10] SysML Editor) and reflects (and allows) only the ModelicaML constructs. **Advantage:** This way a modeler who is used to Modelica does not need to know the UML/SysML. The modeling tool will directly reflect the Modelica wording and appear to be more intuitive to

---

5 Still, any other UML/SysML construct can be used for a descriptive modeling. However, those will not be translated into executable Modelica code.

6 Some of the COTS UML tools (e.g. MagicDraw [18]) provide the capability to reflect a profile as if it were an extension of the meta-model.
the Modelica modeler. Possible inconsistencies can be prevented and constraints can be checked during modeling. Disadvantage: UML tools will probably not be able to read ModelicaML models. However, the readers who are familiar with the graphical notation of the UML/SysML, will still share a common understanding of models.

The current ModelicaML prototype is implemented as a profile (the first approach listed above) that can be used in UML tools that are based on the Eclipse UML2 [19] implementation.
3 Application Example

The ModelicaML profile, as defined in this document was implemented using the PapyrusUML tool, which is an open source tool based on the Eclipse UML2 implementation. This way the ModelicaML profile can be used in other Eclipse UML2-based modeling tools, too (e.g. Topcased [10]).

This chapter presents an example of the application of the ModelicaML profile. The example is explained in the chapter “12.2 Modeling a Tank System” in [3]. The UML tool used for modeling the example is also PapyrusUML.

In order to illustrate the application the next section lists the Modelica code, that would result from the modeling activity; this Modelica code will finally be generated form the ModelicaML model (presented in the sections after).

3.1 Modelica Code Example

```modelica
connector ActSignal "Signal to actuator for setting valve position"
    Real act;
end ActSignal;

partial model BaseController
par Real K = 2 "Gain";
par Real T(unit = "s") = 10 "Time constant";
ReadSignal cIn "Input sensor level, connector";
ActSignal cOut "Control to actuator, connector";
par Real ref "Reference level";
Real error "Deviation from reference level";
Real outCtr "Output control signal";
equation
    error = ref - cIn.val;
cOut.act = outCtr;
end BaseController;

function limitValue
    input Real pMin;
    input Real pMax;
    input Real p;
    output Real pLim;
algorithm
    pLim := if p>pMax then pMax
     else if p<pMin then pMin
      else p;
end limitValue;

connector LiquidFlow "Liquid flow at inlets or outlets"
    Real lflow(unit = "m3/s");
end LiquidFlow;

model LiquidSource
    LiquidFlow qOut;
par Real flowLevel = 0.02;
equation
    qOut.lflow = if time > 150 then 3*flowLevel else flowLevel;
end LiquidSource;

model PIcontinuousController
    extends BaseController(K = 2, T = 10);
    Real x "State variable of continuous PI controller";
equation
    der(x) = error/T;
    outCtr = K*(error + x);
end PIcontinuousController;

connector ReadSignal "Reading fluid level"
    Real val(unit = "m");
end ReadSignal;

model Tank
    ReadSignal tSensor "Connector, sensor reading tank level (m)"
    ActSignal tActuator "Connector, actuator controlling input flow";
    LiquidFlow qIn "Connector, flow (m3/s) through input valve";
    LiquidFlow qOut "Connector, flow (m3/s) through output valve";
par Real area(unit = "m2") = 0.5;
```
parameter Real flowGain(unit = "m²/s") = 0.05;
parameter Real minV= 0, maxV = 10; // Limits for output valve flow
Real h(start = 0.0, unit = "m") "Tank level";
equation
assert(minV>=0,"minV - minimum Valve level must be >= 0 ");
der(h) = (qIn.lflow - qOut.lflow)/area; // Mass balance equation
qOut.lflow = limitValue(minV, maxV, - flowGain*tActuator.act);
tSensor.val = h;
end Tank;
model TanksConnectedPI
  LiquidSource source(flowLevel = 0.02);
  Tank tank1(area = 1);
  Tank tank2(area = 1.3);
  PIcontinuousController piContinuous1(ref = 0.25);
  PIcontinuousController piContinuous2(ref = 0.4);
equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tActuator,piContinuous1.cOut);
  connect(tank1.tSensor,piContinuous1.cIn);
  connect(tank1.qOut,tank2.qIn);
  connect(tank2.tActuator,piContinuous2.cOut);
  connect(tank2.tSensor,piContinuous2.cIn);
end TanksConnectedPI;

3.2 ModelicaML Model Example

Usually, using a UML modeling tool, the model elements can be created either directly in the model browser (a tree-like representation of the classes, etc.) or using diagrams. In any case the model data is stored in the model repository.

Diagrams can be used only for modeling (i.e. capturing the data), and might be deleted after the data is captured. In some cases the modeler may decide to leave some diagrams for documentation or communication purposes. In this case, the modeler will need to select the data that should appear on dedicated diagrams (depending which data can be displayed on a specific diagram type). Most likely, the modeler will also have to arrange the content of the diagrams manually. In the following example diagrams are shown that are left in the model for documentation purposes. The diagrams that were used only for capturing the data are not shown.

Figure 4 shows the TwoTanksSystemExample in the model browser of an Eclipse-based UML2 tool (in this case Papyrus UML). It depicts how packages, classes and their properties (variables or components), ports (instances of type connector classes), behavior (equations or statements), and diagrams, are shown in the model browser.
3.2.1 Packages, Classes, Inheritance

As mentioned above the modeler can create model elements, e.g. packages or classes, in the model browser or using diagrams. Moreover, the modeler can decide which information is shown on a diagram. Figure 5 shows how packages and classes are represented graphically. Note that the respective stereotypes from the ModelicaML profile are applied to the elements created. For example, the stereotype «Package» is applied to the package “DesignModels” and the stereotype «Model» is applied to the TanksConnectedPI class.
Figure 5: Example of ModelicaML package and classes representation on a diagram

Figure 6 shows an example of a UML class diagram presenting the extends relation (inheritance) between classes and possible type modifications. Such a graphical representation is in particular useful when there are multiple levels of inheritance between classes.

Figure 6: Example of inheritance
3.2.2 Components, Variables, Ports

There are different types of properties that can be defined in a class. The type of property is indicated by the used stereotype. For example, a property of a composite type is represented by the stereotype «Component» as depicted in Figure 7.

A property of a primitive type is represented by the stereotype «Variable» (see Figure 8). The stereotypes have attributes that are characteristic for each type of property. For example, the following figure shows how instance modification of a variable is captured.
Instances of connector classes (referred to as ConnectionPorts in ModelicaML) are defined in a similar way (See Figure 9).
3.2.3 Connections
The UML Composite Diagram, as depicted in Figure 10, is similar to the Modelica Connection Diagram. This figure shows the enclosing class and its components. The components have ports (components of type “connector”) and they are connected (resulting in the Modelica connection clauses, see the generated Modelica code shown in the Figure 11).

![Figure 10: UML Composite Diagram (similar to the Modelica Connection Diagram)](image1)

3.2.4 Equations or Algorithm Statements
Non-conditional statements or equations are captured using the body attribute of the OpaqueBehavior (having the respective stereotype applied, in this case «Equations(Code)>>), as illustrated in Figure 12.

![Figure 11: Example of Modelica code generated from the model](image2)
The following Figure 13 highlights the possibility to define loops (for/while) for equations or statements respectively. In ModelicaML any OpaqueBehavior, Activity or OpaqueAction can be enclosed by a loop construct.

3.2.5 Asserts

The stereotype «Assert» extends the UML Constraint by the properties condition, message and assertion-level. The following figure depicts the possible graphical appearance on diagrams.
3.2.6 Conditional Equations or Algorithm Statements

Conditional equations or algorithm statements are modeled using the UML Activity Diagram. Any Activity must have a stereotype applied, which indicates whether the contents shall go into “algorithm” or into “equation” sections. Depending on this the actual equations or statements are captured using the body attribute of OpaqueActions, as illustrated in the following figure.
The following figure addresses the fact that there are 2 possible types of conditional constructs in Modelica, “if” and “when”, that can be modeled in UML using the DecisionNode. Therefore, the stereotypes «if» or «when» must be applied respectively.

The “if”/“when” conditions are captured using “Guards” of the ControlFlow as depicted in Figure 17.
The execution semantics of such Activity Diagrams are the same as for the conditional statements or equations. The conditions are evaluated at each time instance. The actions, presented on the diagram are not time-consuming activities; their execution does not take any simulation time.

Figure 17: Conditions for “if” or “when”

In the previous section the original code of the Modelica function “limitValue” is given using the Modelica shorthand notation:

```
algorithm
pLim := if p>pMax then pMax
else if p<pMin then pMin
else p;
```

Figure 18 shows how the Modelica code that is generated from the activity diagram above looks like. Any further DecisionNode and ControlFlows will result in nested “if” or “when” Constructs respectively.

```plaintext
1 within TwoInputs:
2 function limitValue:
3   input Real pMin;
4   input Real pMax;
5   output Real pLim;
6   input Real p;
7
8 algorithm
9   // code generated from the Activity "algorithm (Diagram)"
10  // Activity.Name: "algorithm (Diagram)"
11  if p < pMin then
12    pLim := pMin; // OpaqueAction.Name: "pLim := pMin;"
13  elseif p > pMax then
14    pLim := pMax; // OpaqueAction.Name: "pLim := pMax;"
15  else
16    pLim := p; // OpaqueAction.Name: "pLim := p;"
17  end if;
18
19 end limitValue;
```

Figure 18: Modelica code generated from an Activity Diagram
Conditional equations are defined similarly to the conditional statements.

Figure 19: Example of conditional equations

### 3.2.7 Initial Equations or Algorithm
For the definition of initial equations or algorithm the same constructs are used as presented in the previous sections. The only difference is the kind of stereotype that needs to be applied to a UML OpaqueBehavior or Activity.

### 3.2.8 Hierarchical Modeling of Statements/Equations
It is also possible to model statements or equations hierarchically using CallBehaviorActions, which only gives the reference to the sub-Activity diagram. Using CallBehaviorActions, the modeler can model hierarchically, which will improve the readability and structure of the model. The names of the CallBehaviorActions will go into comments inside the generated Modelica code.

### 3.2.9 Calculation Models and Calculated Properties
In ModelicaML the stereotypes «Component» is usually used to represent a physical part (a property of composite type) and «Variable» to represent a system attribute (a property of primitive type). For example, the model TanksConnectedPI has physical system parts: 2 tanks, 2 controller and 1 liquid source, all are properties of composite type as indicated by the «Component» stereotype.
Calculation Models (instances of the Calculation Models are referred to as Calculated Properties) are also of composite type. However, they do not represent a physical system part, they are used for calculating system properties. For example, the “areaCal” or volume “volumeCal” Calculated Properties calculate the area and the volume of the tank (see Figure 20). This way the Calculation Models are means for capturing reusable equation sets.
3.2.10 Explicit States or Modes

UML State Machines are typically used for modeling the reactive (event-based) behavior of objects. In ModelicaML the State Machines are used for modeling explicit states or modes of a system or its components. The behavior defined by a State Machine is translated into Modelica algorithm code. Following the principles of a synchronous language the following restrictions are imposed on the semantic of the State Machines as used in ModelicaML:

- The system is at any time in a defined state (note, that the state machines include composite and parallel states, which means that it can be in multiple sub-states at the same time).
- Events and transitions between states take no simulation time. For that reason the effect actions on transitions are not allowed.
- Any behavior that is executed when the state is entered or exited takes no simulation time as well.
- Even though the system will stay in certain states for a time the Do-behavior of a state is also not time-consuming.

Consider the State Machine defined for the tank. (as illustrated in Figure 21). Depending on the level of liquid in the tank (represented by the variable “h”) we can define that the tank is empty, partially filled or even in an overflow state.
The following figure shows the states/modes of the controller. It defines that only of the controller is in the state “on” (i.e. it is powered), it will monitor or control the tank actuator depending on the sensor value received.

3.2.11 Textual Requirements, Measure of Effectiveness

Inspired by the SysML, ModelicaML reuses the concept of textual requirements within models. As in the SysML it is possible to include textual requirements into ModelicaML models and link requirements to model artifacts. This enables traceability between textual requirements and design artifacts, and supports impact analysis when requirements and/or the model change. Figure 23 illustrates how textual requirements appears graphically on diagrams.
In the ModelicaML, a requirement is also a sub-class of the UML Class. As for any UML Class, it is possible to define properties and behavior (e.g. assertions) for a requirement. In doing so it is possible to evaluate if a requirement is violated or not during system simulation. Our current research in this field aims at finding ways to formalize different types of requirements and to find a flexible way to associate requirements with design models. The following examples present some ideas.

Assume the following requirements to be imposed on the two tanks system:

- **Req. 001**: The level of liquid in a tank shall never exceed 80% of the tank-height.
- **Req. 002**: The volume of tank 1 shall be 0.8 m³.

The first requirement specifies a type: Tank in this case. In order to establish the traceability between the textual requirement and the design artifact the class Tank is referenced from the requirement inside the model using the requirement property “specifiesType”. It implies that any instance of a tank must meet this requirement. In contrast, the second requirement is a design requirement defining the required volume of tank 1. This requirement is imposed only on a particular instance of the type Tank. Therefore, the dot-notation in the requirement property “specifiesObject” is used to reference the respective instance. The “specifies…” relations are descriptive only. They are not translated into Modelica code and do not impact the simulation. In order to be able to evaluate these requirements during the system simulation requirements need to be formalized. In the following one possible way to do so is presented.

From the textual statement of the requirement 001 we can identify measurable properties such as: level (current level in a tank), maxLevel (80 % max. allowed level), tank_height (the height of a tank). Moreover, we can define a property indicating if the requirement is violated or not by evaluating: level > maxLevel * tank_height. Consider the following state machine specifying if the requirement 001 is violated or not. The second requirement is modeled in a similar way; it is not presented here.
The modeled requirements can now be instantiated and their properties can be bound to the values within the corresponding design model (TanksConnectedPI in that case). In this example, the declarations for the r001_tank2 (Figure 25) are:

- \( \text{level} = dm.\text{tank1}.h \)
- \( \text{tank_height} = dm.\text{tank1}.\text{tank_height} \)
Note that requirement 001, which specifies the type Tank, is instantiated two times (because there are two tanks in the system).

Figure 26 shows the results of the evaluation (the tank_height is 0.6m in this example). The requirement 001 evaluated for the tank2 (r001_tank2) was violated two times during the simulation.

![Figure 26: Example of requirements evaluation during system simulation](image)

**Requirement Representatives**

In the examples presented above there is only one design of the two tank system that is validated against the defined requirements. If there are multiple design alternatives, and hence multiple design models exist that have to meet the same requirements, Requirement Representatives are used to bind the requirements to multiple design alternatives. Therefore, a Requirement Representative class typically inherits from exactly one existing Requirement class but does not redefine it. Thus, a Requirement Representative is not a new Requirement, it represents an existing Requirement.

As Requirements, the Requirement Representatives are instantiated in a simulation model and Requirement assertions are evaluated during the simulation of one specific system design model. For example, assume that different types or a different number of tanks are used in different designs. The Requirements are still the same. However, the traceability links (defined using the “specifies…” property) are different as well as the accessible values inside the design model are different in each alternative. In order to avoid the duplication of Requirements, the Requirements Representatives are used in the context of one particular design alternative.

**Measure of Effectiveness**

Similar to the concept of textual requirements, the modeller can define measure of effectiveness which are used to record measurable properties of system models during simulations and to compare them according to predefined metrics in order to select the best potential design alternative.

A measure of effectiveness model bound to multiple design models can be used to compare their performance of different system solutions. For example, assume, there are multiple tanks systems with different types of controllers or different controller configurations. A measure of effectiveness model could compare how long the different design alternative models need to adjust the level of liquid in tanks to the defined reference level each time the input flow changes.

---

9 A measure of effectiveness is a model on its own.
4 ModelicaML Profile - Full List Of Stereotypes

Profile Version: 0.0.70
Note: In each section the stereotypes are sorted alphabetically.

4.1 ModelicaClassConstructs

4.1.1 Stereotype: «Block»
Description: This stereotype is used to represent the Modelica "block". Modelica "block" is the same as Modelica "model", but the ports (instances of "connector" classes) must have fixed causality (i.e. "input" or "output").

Extends the UML meta-class: Class
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

4.1.2 Stereotype: «CalculationModel» (is not a Modelica concept)
Description: A CalculationModel is a sub-type of Modelica "model". It is defined in order to enable reusing of equations. It does not represent a physical system part and cannot have external connectors.
When instantiated, it calculates values that are used by other components.

Inherits from the Stereotype: «Model» the properties: encapsulated, partial

4.1.3 Stereotype: «Connector»
Description: This stereotype is used to represent the Modelica restricted class "connector". A "connector" cannot contain equation or algorithm sections, it can have causality and flow specification. All ports must be typed by a "connector".

Extends the UML meta-class: DataType
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

Property: expandable (multiplicity of entries: 1..1), type: Boolean.

4.1.4 Stereotype: «Function»
Description: Modelica "function" resembles the Modelica "block" but it:
- can only have inputs and outputs; no connectors
- does not have a connectable interface but instead needs to be called
- can be called recursively
- is more dynamic; the arguments are computed at run-time
- does not have internal states
- can have only one algorithm section and no equation section; is always imperative
- becomes part of the equation system as black boxes only; the internals are not manipulated symbolically by the solver

Extends the UML meta-class: FunctionBehavior
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial
4.1.5 Stereotype: «Model»

Description: This stereotype is used to represent the Modelica restricted class "model". All the following Modelica class elements are allowed in models: Both equation and algorithm sections; connectors, inputs, outputs; states;

Extends the UML meta-class: Class
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

4.1.6 Stereotype: «ModelicaClass» (is abstract)

Description: This stereotype has attributes that are common to all Modelica restricted classes.

Property: encapsulated (multiplicity of entries: 1..1), type: Boolean.
Property: partial (multiplicity of entries: 1..1), type: Boolean.

4.1.7 Stereotype: «ModelicaPackage»

Description: This stereotype is used to represent the Modelica restricted class "package".

Extends the UML meta-class: Package
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

4.1.8 Stereotype: «Record»

Description: This stereotype is used to represent the Modelica "record". A Modelica "record" cannot contain equation or algorithm sections and does not contain causality and flow specification.

Extends the UML meta-class: DataType
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

Property: shortClassDefinition (multiplicity of entries: 0..1), type: String. Property description: This is the short-hand class declaration. Except for Modelica "type" the same can be expressed using inheritance construct.

4.1.9 Stereotype: «Type»

Description: This stereotype is used to represent the Modelica restricted class "type".

Extends the UML meta-class: DataType
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

4.2 ModelicaCompositeConstructs

4.2.1 Stereotype: «CalculatedProperty» (is not a Modelica concept)

Description: Does not represent a physical system part. This component supplies other components with calculated values.
Inherits from the Stereotype: «Component» the properties: visibility, final, redeclaration, instanceModification, conditionalExpression, instanceArraySize

4.2.2 Stereotype: «Component»
Description: This stereotype is used to represent a Modelica component of a composite type.

Extends the UML meta-class: Property
Inherits from the Stereotype: «InstanceOfAClass» the properties: final, redeclaration, instanceModification, conditionalExpression, instanceArraySize

Property: visibility (multiplicity of entries: 1..1), type: ModelicaAccess (public, protected).

4.2.3 Stereotype: «Connection»
Description: This stereotype is used to represent the Modelica "connection" clause and implies its associated semantics (Kirchhoff's laws).

Extends the UML meta-class: Connector

4.2.4 Stereotype: «ConnectionPort»
Description: This stereotype is used to represent a Modelica class instance of the type "connector".

Extends the UML meta-class: Port
Inherits from the Stereotype: «InstanceOfAClass» the properties: final, redeclaration, instanceModification, conditionalExpression, instanceArraySize

Property: causality (multiplicity of entries: 0..1), type: ModelicaCausality (input, output).

4.2.5 Stereotype: «FunctionArgument»
Description: This stereotype is used to represent variables of functions.

Extends the UML meta-class: Parameter
Inherits from the Stereotype: «InstanceOfAClass» the properties: final, redeclaration, instanceModification, conditionalExpression, instanceArraySize

Property: flowFlag (multiplicity of entries: 0..1), type: ModelicaFlowFlag (flow, stream).
Property: visibility (multiplicity of entries: 1..1), type: ModelicaAccess (public, protected).
Property: scope (multiplicity of entries: 0..1), type: ModelicaScope (inner, outer).
Property: causality (multiplicity of entries: 0..1), type: ModelicaCausality (input, output).
Property: variability (multiplicity of entries: 1..1), type: ModelicaVariability (continuous, discrete, parameter, constant).
Property: declarationEquation (multiplicity of entries: 0..1), type: String. Property description: Declaration Equation refers to the shorthand notation in Modelica. The result must be of the same type as the Modelica component itself. In this case the value of the tag is the right-hand-expression of the equation. The “=” sign is omitted, i.e., it is implicit.
4.2.6 **Stereotype: «InstanceOfAClass» (is abstract)**

*Description:* This stereotype has attributes that are common to all Modelica class instances.

Property: **final** (multiplicity of entries: 1..1), type: Boolean.

Property: **redeclaration** (multiplicity of entries: 0..1), type: ModelicaRedeclaration (replaceable, redeclare).

Property: **instanceModification** (multiplicity of entries: 0..*), type: String. *Property description:* Each entry represents one instance modification of type "instance name = modification expression". The parenthesis "(" and ")" and the commas are omitted.

Property: **conditionalExpression** (multiplicity of entries: 0..1), type: String. *Property description:* Only if the condition is true then the Modelica component is created; the condition is only evaluated at compilation time and is often used for configuration changes. The result of expression must be of type Boolean. This string contains the expression. The keyword "if" is omitted.

Property: **instanceArraySize** (multiplicity of entries: 0..*), type: String. *Property description:* Each entry string contains the full specification of the array. For example, ":[1,3,;]".

4.2.7 **Stereotype: «Variable»**

*Description:* This stereotype is used to represent a Modelica class instance of a primitive type (Real, String, Boolean, Integer) or a sub-type of Modelica restricted class "type".

*Extends the UML meta-class: Property*

*Inherits from the Stereotype: «InstanceOfAClass»* the properties: final, redeclaration, instanceModification, conditionalExpression, instanceArraySize

Property: **flowFlag** (multiplicity of entries: 0..1), type: ModelicaFlowFlag (flow, stream).

Property: **visibility** (multiplicity of entries: 1..1), type: ModelicaAccess (public, protected).

Property: **scope** (multiplicity of entries: 0..1), type: ModelicaScope (inner, outer).

Property: **causality** (multiplicity of entries: 0..1), type: ModelicaCausality (input, output).

Property: **variability** (multiplicity of entries: 1..1), type: ModelicaVariability (continuous, discrete, parameter, constant).

Property: **declarationEquation** (multiplicity of entries: 0..1), type: String. *Property description:* Declaration Equation refers to the shorthand notation in Modelica. The result must be of the same type as the Modelica component itself. In this case the value of the tag is the right-hand-expression of the equation. The "=" sign is omitted, i.e., it is implicit.

4.3 **ModelicaBehaviorConstructs**

4.3.1 **Stereotype: «Algorithm(Code)»**

*Description:* This stereotype is used to represent a set of statements in an algorithm section.

*Extends the UML meta-class: OpaqueBehavior*

*Inherits from the Stereotype: «AlgorithmLoopProperties»* the properties: loopKind, forIndices, whileExpression
4.3.2  **Stereotype: «AlgorithmLoopProperties» (is abstract)**

*Description:* This stereotype has properties that are inherited by other stereotypes in order to be able to capture the "for" or a "while" loop with the "for-indices" or "while expression" respectively.

Property: loopKind (multiplicity of entries: 0..1), type: ModelicaAlgorithmLoopKind (for, while).
Property: forIndices (multiplicity of entries: 0..1), type: String.
Property: whileExpression (multiplicity of entries: 0..1), type: String.

4.3.3  **Stereotype: «AlgorithmStatements(CodeInDiagram)»**

*Description:* This stereotype is used to represent a set of algorithm statements in a diagram.

*Extends the UML meta-class: OpaqueAction*
* Inherits from the Stereotype: «AlgorithmLoopProperties»* the properties: loopKind, forIndices, whileExpression

4.3.4  **Stereotype: «Assert»**

*Description:* This stereotype represents the Modelica assert() function.

*Extends the UML meta-class: Constraint*

Property: condition (multiplicity of entries: 1..1), type: String.
Property: message (multiplicity of entries: 1..1), type: String.
Property: level (multiplicity of entries: 1..1), type: ModelicaAssertionLevel (warning, error).

4.3.5  **Stereotype: «ConditionalAlgorithm(Diagram)»**

*Description:* This stereotype is used to represent an algorithm section. An Activity Diagram is used to model the conditional constructs (loop, if, when). Any OpaqueActions, in such an Activity Diagram, must contain Modelica statements in the "body" attribute.

*Extends the UML meta-class: Activity*
* Inherits from the Stereotype: «AlgorithmLoopProperties»* the properties: loopKind, forIndices, whileExpression

Property: isSubActivity (multiplicity of entries: 1..1), type: Boolean. *Property description:* This attribute indicates that this activity is a sub-activity. It is invoked by another activity using CallBehaviorAction.

4.3.6  **Stereotype: «ConditionalEquations(Diagram)»**

*Description:* This stereotype is used to represent an equation section. An Activity Diagram is used to model the conditional constructs (loop, if, when). Any OpaqueActions, in such an Activity Diagram, must contain Modelica equations in the "body" attribute.

*Extends the UML meta-class: Activity*
* Inherits from the Stereotype: «EquationsLoopProperties»* the properties: loopKind, forIndices

Property: isSubActivity (multiplicity of entries: 1..1), type: Boolean. *Property description:* This attribute indicates that this activity is a sub-activity. It is invoked by another activity using CallBehaviorAction.
4.3.7 Stereotype: «Equations(Code)»
Description: This stereotype is used to represent a set of equations in an equation section.

Extends the UML meta-class: OpaqueBehavior
Inherits from the Stereotype: «EquationsLoopProperties» the properties: loopKind, forIndices

4.3.8 Stereotype: «Equations(CodeInDiagram)»
Description: This stereotype is used to represent a set of equations in a diagram.

Extends the UML meta-class: OpaqueAction
Inherits from the Stereotype: «EquationsLoopProperties» the properties: loopKind, forIndices

4.3.9 Stereotype: «EquationsLoopProperties» (is abstract)
Description: This stereotype has properties that are inherited by other stereotypes in order to be able to capture the "for" loop with the "for-indices".

Property: loopKind (multiplicity of entries: 0..1), type: ModelicaEquationsLoopKind (for).
Property: forIndices (multiplicity of entries: 0..1), type: String.

4.3.10 Stereotype: «If»
Description: This stereotype is used to represent the Modelica "if" keyword.

Extends the UML meta-class: DecisionNode

4.3.11 Stereotype: «InitialAlgorithm(Code)»
Description: This stereotype is used to represent a set of statements in an initial algorithm section.

Extends the UML meta-class: OpaqueBehavior
Inherits from the Stereotype: «AlgorithmLoopProperties» the properties: loopKind, forIndices, whileExpression

4.3.12 Stereotype: «InitialConditionalAlgorithm(Diagram)»
Description: This stereotype is used to represent an initial algorithm section. An Activity Diagram is used to model the conditional constructs (loop, if, when). Any OpaqueActions, in such an Activity Diagram, must contain Modelica statements in the "body" attribute.

Extends the UML meta-class: Activity
Inherits from the Stereotype: «AlgorithmLoopProperties» the properties: loopKind, forIndices, whileExpression
4.3.13 **Stereotype: «InitialConditionalEquations(Diagram)»**

*Description:* This stereotype is used to represent an initial equation section. An Activity Diagram is used to model the conditional constructs (loop, if, when). Any OpaqueActions, in such an Activity Diagram, must contain Modelica equations in the "body" attribute.

*Extends the UML meta-class:* Activity
*Inherits from the Stereotype:* «EquationsLoopProperties» the properties: loopKind, forIndices

4.3.14 **Stereotype: «InitialEquations(Code)»**

*Description:* This stereotype is used to represent a set of equations in an initial equation section.

*Extends the UML meta-class:* OpaqueBehavior
*Inherits from the Stereotype:* «EquationsLoopProperties» the properties: loopKind, forIndices

4.3.15 **Stereotype: «ModelicaStateMachine»**

*Description:* This stereotypes indicates that this a special case of the UML StateMachine. It uses a limited set of the UML StateMachine meta-model and is translated into executable Modelica code.

*Extends the UML meta-class:* StateMachine
*Inherits from the Stereotype:* «AlgorithmLoopProperties» the properties: loopKind, forIndices, whileExpression

4.3.16 **Stereotype: «StateDependency» (is not a Modelica concept)**

*Description:*

*Extends the UML meta-class:* ControlFlow

Property: isInState (multiplicity of entries: 1..*), type: State. *Property description:* Reference to a state in one of the state machines of the class.

Property: logicalOperator (multiplicity of entries: 1..1), type: LogicalOperators (and, or, not). *Property description:* The logical operator to be used between the multiple states entries.

4.3.17 **Stereotype: «When»**

*Description:* This stereotype is used to represent the Modelica "when" keyword.

*Extends the UML meta-class:* DecisionNode

4.4 **ModelicaRelationsConstructs**

4.4.1 **Stereotype: «ExtendsRelation»**

*Description:* This stereotype has attributes which allow the capturing of type modifications.
Extends the UML meta-class: Generalization

Property: typeModification (multiplicity of entries: 0..*), type: String. Property description: Each entry represents a Modelica type modification. Commas and the parenthesis "(" and ")" are omitted.

4.4.2 Stereotype: «TypeRelation»
Description: This stereotype has attributes which allow the capturing of type modifications. It can only be used between classes of primitive type.

Extends the UML meta-class: Generalization

Property: typeModification (multiplicity of entries: 0..*), type: String. Property description: Each entry represents a Modelica type modification. Commas and the parenthesis "(" and ")" are omitted.

Property: typeArraySize (multiplicity of entries: 0..*), type: String. Property description: Each string contains the full array specification (e.g. "[1,3,:"]

4.5 ModelicaRequirementConstructs

4.5.1 Stereotype: «MeasureOfEffectiveness» (is not a Modelica concept)
Description: This stereotype is used to represent a model that measures performances of other system models and compares them in order to determine the best potential design solution.

Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

4.5.2 Stereotype: «Requirement» (is not a Modelica concept)
Description: The concept of textual requirements is reused from the OMG SysML. This stereotype is used to represent a requirement with properties such as id, text (representing the textual statement), etc.
Moreover, a requirement can have variables and behavior defined which can be evaluated during simulations.

Extends the UML meta-class: Class
Inherits from the Stereotype: «ModelicaClass» the properties: encapsulated, partial

Property: id (multiplicity of entries: 1..1), type: String. Property description: Requirement ID.


Property: specifiesType (multiplicity of entries: 0..*), type: ModelicaClass. Property description: This property implies that all instantiated objects of the referenced types shall fulfill this requirement.

Property: specifiesObject (multiplicity of entries: 0..*), type: String. Property description: This property implies that only the referenced objects shall fulfill this requirement.
4.5.3  **Stereotype: «RequirementInstance» (is not a Modelica concept)**

*Description:* This stereotype indicates that this component is an instantiation of a requirement.

*Inherits from the Stereotype:* «Component» the properties: visibility, final, redeclaration, instanceModification, conditionalExpression, instanceArraySize

4.5.4  **Stereotype: «RequirementRepresentative» (is not a Modelica concept)**

*Description:* The requirement observer represents a requirement. It is used to bind same requirements to multiple design models.

*Inherits from the Stereotype:* «ModelicaClass» the properties: encapsulated, partial

- **Property:** specifiesType (multiplicity of entries: 0..*), type: ModelicaClass.
- **Property:** specifiesObject (multiplicity of entries: 0..*), type: String.
- **Property:** designModelReference (multiplicity of entries: 1..1), type: ModelicaClass. *Property description:* A reference to the system desining model.

4.5.5  **Stereotype: «Traceability» (is not a Modelica concept)**

*Description:* This stereotype can be applied to any model element. It has properties that can be used for setting up the traceability between requirements and design artifacts.

*Extends the UML meta-class:* NamedElement

- **Property:** specifiedByRequirement (multiplicity of entries: 0..*), type: Requirement.

4.6  **ModelicaSimulationConstructs**

4.6.1  **Stereotype: «CodeGeneration» (is not a Modelica concept)**

*Description:* This stereotype is used for code generation specific information.

*Extends the UML meta-class:* Class

- **Property:** generateInteractiveSimInitFile (multiplicity of entries: 1..1), type: Boolean. *Property description:* If this property is set the code generator will generate an init-file that is required for the interactive simulation (OMC specific).

4.6.2  **Stereotype: «ExcludeFromCGIfUserInteractive» (is not a Modelica concept)**

*Description:* This stereotypes is used to indicate that this behavior shall be excluded from the code generation if the referenced variable is set to be user interactive (variability is set to "parameter").

*Extends the UML meta-class:* StateMachine, OpaqueBehavior, Activity

- **Property:** variableReference (multiplicity of entries: 1..1), type: Variable. *Property description:* Reference to the variable that is user interactive (must have the stereotype <> applied) and thus
4.6.3 **Stereotype: «SimulateCommand»**

*Description:* This stereotype contains the properties which are used in the "simulate()" command in Modelica tools.

*Extends the UML meta-class: **DataType***

Property: `startTime` (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: `stopTime` (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: `model` (multiplicity of entries: 1..1), type: ModelicaClass.

4.6.4 **Stereotype: «UserInteractive» (is not a Modelica concept)**

*Description:* This stereotype is used to indicate that the value of this variable shall be manipulated by the user. Therefore it is necessary to set the variability to "parameter" (this requirement is OMC-specific). Moreover, the modeler can define declaration value for the parameter if necessary.

*Extends the UML meta-class: **Property***

Property: `declaration` (multiplicity of entries: 0..1), type: String. *Property description:* Declaration value for the parameter.

4.7 **ModelicaAnnotationConstructs**

4.7.1 **Stereotype: «Annotation(CodeGeneration)»**

*Description:* See the Modelica specification.

*Extends the UML meta-class: **Comment***

Property: `Evaluate` (multiplicity of entries: 0..1), type: Boolean.
Property: `HideResult` (multiplicity of entries: 0..1), type: Boolean.
Property: `Inline` (multiplicity of entries: 0..1), type: Boolean.
Property: `LateInline` (multiplicity of entries: 0..1), type: Boolean.
Property: `smoothOrder` (multiplicity of entries: 0..1), type: Integer.

4.7.2 **Stereotype: «Annotation(Documentation)»**

*Description:* See the Modelica specification.

*Extends the UML meta-class: **Comment***

Property: `info` (multiplicity of entries: 0..1), type: String.
Property: `revisions` (multiplicity of entries: 0..1), type: String.

4.7.3 **Stereotype: «Annotation(SimulationExperiment)»**

*Description:* The experiment annotation defines the default start time (StartTime) in [s], the default stop time (StopTime) in [s], and the default relative integration tolerance (Tolerance) for simulation experiments to be carried.
out with the model or block at hand.

*Extends the UML meta-class: Comment*

Property: **StartTime** (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: **StopTime** (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: **Tolerance** (multiplicity of entries: 0..1), type: UnlimitedNatural.

### 4.7.4 Stereotype: «Annotation(VersionHandling)»

*Description:* A top-level package or model can specify the version of top-level classes it uses, its own version number, and if possible how to convert from previous versions. This can be used by a tool to guarantee that consistent versions are used, and if possible to upgrade usage from an earlier version to a current one.

*Extends the UML meta-class: Comment*

Property: **version** (multiplicity of entries: 0..*), type: String. *Property description:* Defines the version number of the model or package. All classes within this top-level class have this version number.

Property: **conversion** (multiplicity of entries: 0..*), type: String. *Property description:* The first "(" and the last ")" are omitted.

Property: **uses** (multiplicity of entries: 1..1), type: String. *Property description:* The first "(" and the last ")" are omitted.

### 4.7.5 Stereotype: «Annotation»

*Description:* Annotations are intended for storing extra information about a model, such as graphics, documentation or versioning, etc. A Modelica tool is free to define and use other annotations, in addition to those defined here, according to section 14.1. of the Modelica specification.

The only requirement is that any tool shall save files with all annotations from this chapter and all vendor-specific annotations intact. To ensure this, annotations must be represented with constructs according to the Modelica grammar. The specification in this document defines the semantic meaning if a tool implements any of these annotations.

*Extends the UML meta-class: Comment*

Property: **fullAnnotationString** (multiplicity of entries: 0..1), type: String. *Property description:* This is a qualified annotation string. The first "(" and the last ")" are omitted.

### 4.8 ModelicaPredefinedTypes

#### 4.8.1 Primitive Type: ModelicaBoolean

Property: **quantity** (multiplicity of entries: 0..1), type: String.
Property: **fixed** (multiplicity of entries: 0..1), type: Boolean.
Property: **start** (multiplicity of entries: 0..1), type: Boolean.
4.8.2 Primitive Type: ModelicaInteger
Property: quantity (multiplicity of entries: 0..1), type: String.
Property: min (multiplicity of entries: 0..1), type: Integer.
Property: max (multiplicity of entries: 0..1), type: Integer.
Property: fixed (multiplicity of entries: 0..1), type: Boolean.
Property: start (multiplicity of entries: 0..1), type: Integer.

4.8.3 Primitive Type: ModelicaReal
Property: quantity (multiplicity of entries: 0..1), type: String.
Property: unit (multiplicity of entries: 0..1), type: String.
Property: displayUnit (multiplicity of entries: 1..1), type: String.
Property: min (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: max (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: fixed (multiplicity of entries: 0..1), type: Boolean.
Property: nominal (multiplicity of entries: 0..1), type: UnlimitedNatural.
Property: stateSelect (multiplicity of entries: 0..1), type: ModelicaStateSelect (default, never, avoid, prefer, always).

4.8.4 Primitive Type: ModelicaString
Property: quantity (multiplicity of entries: 0..1), type: String.
Property: start (multiplicity of entries: 0..1), type: String.
5 Conclusion and Outlook

Currently, the ModelicaML does not yet support the complete Modelica specification [1]. Furthermore, some constructs, such as Modelica instance- or type-modification, the actual equations or the algorithm statements, are still captured as strings. This leads to refactoring effort when, for example, the name of classes or components has changed, and prevents an efficient syntax- or semantics-analysis of the model. Current research activities aim at improving this by means of modeling tools features (e.g. code completion) or by extending the meta-model for an adequate support of Modelica expressions.

Moreover, ModelicaML uses a limited set of the UML/SysML concepts, in order to facilitate the creation of executable models by transforming them into Modelica code. Some formalisms that are used in UML/SysML for modeling the behavior of classes are not supported yet. For example, it is not yet defined how to translate a more complete set of Activity Diagrams or the UML State Machines into valid Modelica code, taking into account the properties and limitations of a synchronous language. Interactions (Sequence Diagrams) are typically used in the context of UML/SysML for capturing (modeling) scenarios or test cases. The support for this concept is also subject to future research work for ModelicaML.

Furthermore, “allocation” is a key concept in systems engineering. It facilitates the definition of functional solutions and the allocation of the resulting, required behavior to multiple physical structure designs. In doing so, the system engineer can perform trade-off studies in order to select the best potential design. SysML support the allocation as a descriptive means for modeling separate structures or behaviors and allocating them in a flexible way. The question “how can the allocation concepts help to gain flexibility in modeling system alternatives?” is also being investigated in the work on ModelicaML.

---

10 For example, nested connectors are not supported yet.
References