



# SysML-Modelica Transformation

*Version 1.0*

---

OMG Document Number: formal/2012-11-09

Normative reference: <http://www.omg.org/spec/SyM/1.0/>

Machine consumable files:

Normative:

<http://www.omg.org/spec/SyM/20120320/SysML4Modelica-Profile.xml>

Non-normative:

<http://www.omg.org/spec/SyM/20120214/Modelica2ModelicaUnparsed.qvto>

<http://www.omg.org/spec/SyM/20120215/ModelicaUnparsed2SysML.qvto>

<http://www.omg.org/spec/SyM/20120216/SysML2ModelicaUnparsed.qvto>

<http://www.omg.org/spec/SyM/20120217/ModelicaUnparsed2Modelica.qvto>

<http://www.omg.org/spec/SyM/20120313/openModelica.emof>

<http://www.omg.org/spec/SyM/20120312/SysML4ModelicaProfile.mdzip>

---

Copyright © 2009-2010, Atego  
Copyright © 2009-2010, Deere & Company  
Copyright © 2009-2010, EADS  
Copyright © 2009-2010, ESA/ESTEC  
Copyright © 2009-2010, Georgia Institute of Technology  
Copyright © 2009-2010, Jet Propulsion Laboratory  
Copyright © 2009-2010, Linköping University  
Copyright © 2009-2010, Lockheed Martin Corporation  
Copyright © 2009-2010, NoMagic Inc.  
Copyright © 2012, Object Management Group, Inc.

## USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

The specification customizes the Unified Modeling Language (UML) specification of the Object Management Group (OMG) to address the requirements of Systems Engineering as specified in the UML for Systems Engineering RFP, OMG document number ad/2003-03-41. This document includes references to and excerpts from the UML 2 Superstructure Specification and UML 2 Infrastructure Specification with copyright holders and conditions as noted in those documents.

## LICENSES

Redistribution and use of this specification, with or without modification, are permitted provided that the following conditions are met: (1) Redistributions of this specification must reproduce the above copyright notice, this list of conditions and disclaimers in the documentation and/or other materials provided with the distribution; (2) The Copyright Holders listed in the above copyright notice may not be used to endorse or promote products derived from this specification without specific prior written permission; (3) All modified versions of this specification must include a prominent notice stating how and when the specification was modified; and (4) No modifications to this OMG SysML™ specification may be published under or identified by that name, except for versions published by OMG and incorporating official changes made through the applicable procedures of OMG. OMG SysML™ is a trademark of OMG, and no unauthorized version or revision of the OMG SysML specification may use the trademark "OMG SysML" or claim any connection with or endorsement by OMG.

In accordance with the above copyright provisions, the companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute OMG SysML and to modify OMG SysML and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification. Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, nonsublicenseable, perpetual, worldwide license, to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of this document in your possession or control.

This document was derived from the “Systems Modeling Language (SysML) Specification, version 1.0 DRAFT,” OMG document (ad/2006-03-01) submitted to OMG in response to the “UML for Systems Engineering RFP” (ad/2003-03-41). Review and editing in the OMG process produced the “OMG SysML Specification Final Adopted Specification” (ptc/2006-05-04). Subsequent changes to the specification are controlled through the OMG process as documented at the OMG Technology Document website - <http://www.omg.org/technology/documents/>.

## PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

## GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

## DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED “AS IS” AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

## RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

## TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™, Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™, IMM™, MOF™, OMG Interface Definition Language (OMG IDL)™, and OMG Systems Modeling Language (OMG SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

## COMPLIANCE

The Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with OMG SysML™. Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

## **OMG's Issue Reporting Procedure**

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page <http://www.omg.org>, under Documents, Report a Bug/Issue ([http://www.omg.org/report\\_issue.htm](http://www.omg.org/report_issue.htm)).



# Table of Contents

Preface .....	V
Part I	
1 Scope .....	3
1.1 OMG SysMLTM .....	3
1.2 SysML-Modelica .....	4
1.3 Conclusion .....	4
1.4 Objective .....	4
2 Conformance .....	4
3 Normative References .....	5
4 Terms and Definitions .....	5
5 Symbols .....	5
6 Additional Information .....	6
6.1 Acknowledgments .....	6
7 Transformation Approach .....	7
7.1 General .....	7
7.2 Overview .....	9
Part II	
8 Class Definition .....	11
8.1 Overview .....	11
8.2 «modelicaClassDefinition» .....	13
8.3 «modelicaClass» and «modelicaModel» .....	14
8.4 «modelicaRecord» .....	15
8.5 «modelicaBlock» .....	16
8.6 «modelicaConnector» .....	17
8.7 «modelicaType» .....	18
8.8 «modelicaPackage» .....	19
8.9 «modelicaFunction» .....	19
8.10 «modelicaExtends» .....	21
8.11 «modelicaDer» .....	23

8.12	«modelicaConstrainedBy» .....	23
8.13	Short Class Definitions .....	24
<b>9</b>	<b>Predefined Types .....</b>	<b>25</b>
9.1	Overview .....	25
9.2	ModelicaReal .....	26
9.3	ModelicaInteger .....	27
9.4	ModelicaBoolean .....	28
9.5	ModelicaString .....	28
9.6	ModelicaStateSelect .....	29
9.7	ModelicaExternalObject .....	29
<b>10</b>	<b>Component Declarations .....</b>	<b>31</b>
10.1	Overview .....	31
10.2	«modelicaValueProperty» .....	33
10.3	«modelicaPart» .....	34
10.4	«modelicaPort» .....	36
10.5	«modelicaFunctionParameter» .....	37
<b>11</b>	<b>Equation and Algorithm Sections .....</b>	<b>39</b>
11.1	Overview .....	39
11.2	«modelicaEquation» .....	39
11.3	«modelicaAlgorithm» .....	40
11.4	«modelicaConnection» .....	41
<b>12</b>	<b>Other Related Constructs .....</b>	<b>43</b>
12.1	«modelicaSimulation» .....	43
12.2	"modelicaAnnotation" .....	44
<b>Part III</b>		
<b>13</b>	<b>Modelica Meta-Modeling Approach .....</b>	<b>47</b>
13.1	General .....	47
<b>14</b>	<b>Modelica Meta-Model Constructs .....</b>	<b>51</b>
14.1	The Model Structure Definition .....	51
14.1.1	Program .....	51
14.1.2	Within .....	51
14.1.3	Path .....	51
14.2	Class Definition .....	51
14.2.1	Class .....	51



14.2.2	Restriction	52
14.2.3	ClassDef	52
14.2.4	TypeSpec	53
14.2.5	EnumDef	54
14.2.6	EnumLiteral	54
14.2.7	ClassPart	54
14.2.8	ExternalDecl	55
14.2.9	ElementItem	55
14.2.10	Element	56
14.2.11	InnerOuter	56
14.2.12	ComponentRef	56
14.2.13	Subscript	57
14.2.14	ConstrainClass	57
14.2.15	ElementSpec	57
14.3	Import	58
14.4	Annotation and Comments	58
14.4.1	Annotation	58
14.4.2	Comment	59
14.5	Component Definition	59
14.5.1	ComponentItem	59
14.5.2	ComponentCondition	59
14.5.3	Component	59
14.5.4	ElementAttributes	59
14.5.5	Variability	60
14.5.6	Direction	60
14.5.7	ArrayDim	60
14.6	Modifications and Redclarations	60
14.6.1	Modification	60
14.6.2	ElementArg	61
14.6.3	RedeclareKeywords	61
14.6.4	Each	61
14.7	Behavior	62
14.7.1	EquationItem	62
14.7.2	AlgorithmItem	62
14.7.3	Equation	62
14.7.4	Algorithm	63
14.8	Expressions	64
14.8.1	Exp	64
14.8.2	FunctionArgs	66
14.8.3	ForIterator	66
14.8.4	ForIterators	66
14.8.5	NamedArg	66
14.8.6	Operator	67

## Part IV

15	Class Definitions	71
----	-------------------	----

15.1	«modelicaClassDefinition»	71
15.2	«modelicaClass»	72
15.3	«modelicaModel»	72
15.4	«modelicaRecord»	72
15.5	«modelicaBlock»	73
15.6	«modelicaConnector»	73
15.7	«modelicaType»	73
15.8	«modelicaPackage»	73
15.9	«modelicaFunction»	74
15.10	«modelicaExtends»	74
15.11	«modelicaDer»	75
15.12	«modelicaConstrainedBy»	75
16	Predefined Types	77
16.1	Overview	77
17	Component Declarations	79
17.1	Overview	79
17.2	«modelicaPart»	79
17.3	«modelicaPort»	80
17.4	«modelicaValueProperty»	80
17.5	«modelicaFunctionParameter»	81
18	Equation and Algorithm Sections	83
18.1	Overview	83
18.2	«modelicaEquation»	83
18.3	«modelicaAlgorithm»	84
18.4	«modelicaConnection»	84
Part V		
Annex A	Examples	87
Annex B	Justification	107
Annex C	QVT Transformation	111

# Preface

## About the Object Management Group

### OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at <http://www.omg.org/>.

## OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from this URL:

<http://www.omg.org/spec>

Specifications are organized by the following categories:

### Business Modeling Specifications

#### Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

#### IDL/Language Mapping Specifications

#### Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

#### Modernization Specifications

## Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- CORBAServices
- CORBAFacilities

## OMG Domain Specifications

## CORBA Embedded Intelligence Specifications

## CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters  
109 Highland Avenue  
Needham, MA 02494  
USA  
Tel: +1-781-444-0404  
Fax: +1-781-444-0320  
Email: [pubs@omg.org](mailto:pubs@omg.org)

Certain OMG specifications are also available as ISO standards. Please consult <http://www.iso.org>

## Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or sub clause headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

**Helvetica/Arial - 10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

**Courier - 10 pt. Bold:** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

**Note** – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

## Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to [http://www.omg.org/report\\_issue.htm](http://www.omg.org/report_issue.htm).

# Part I - Overview

## General Information

OMG SysML™ is a standardized general purpose graphical modeling language for capturing complex system descriptions in terms of their structure, behavior, properties, and requirements. Modelica is a standardized general purpose systems modeling language for analyzing the continuous and discrete time dynamics of complex systems based on solving differential algebraic equations. Integrating the descriptive power of SysML models with the analytic and computational power of Modelica models provides a capability that is significantly greater than SysML or Modelica individually. The objectives of this document are to enable and specify a standardized bi-directional transformation between the two modeling languages that will support implementations to efficiently and automatically transfer the modeling information transfer between SysML and Modelica models without ambiguity.

The transformation approach is to specify first an extension to SysML called the SysML4Modelica profile to represent the Modelica constructs and then to specify the SysML-Modelica Transformation between the profile constructs and the Modelica language. Introducing the profile into the transformation approach is intended to simplify the transformation to Modelica and facilitate model reuse by more directly leveraging existing model libraries within Modelica. In this way, the user first creates the system model in a SysML modeling tool as he would normally do. The user then selects the part of the model to be analyzed by Modelica (e.g., a particular subsystem) and applies the SysML4Modelica profile to create an analytic representation of that part of the model. The SysML modeling tool is expected to include this profile. The analytic representation expressed in the SysML4Modelica profile is then transformed to a Modelica model where it can be executed by a Modelica modeling tool.

The SysML-Modelica transformation leverages the fundamental concepts of the Model-Driven Architecture (MDA). Different transformation implementations can be applied to implement this specification such as the QVT and others. The transformation can leverage an XMI formatted static file transfer or other mechanisms such as APIs that support a dynamic interchange capability.

This specification is organized as follows:

Part I - Introduction (normative)

Part II - SysML4Modelica profile (normative)

Part III - Modelica meta-model (non-normative)

Part IV - SysML-Modelica mapping, a bidirectional mapping between the SysML4Modelica profile and the Modelica meta-model (non-normative)

Part V - Annexes: Examples, Justification, and QVT transformation (non-normative)



# 1 Scope

## 1.1 OMG SysML™

OMG SysML™ is a general-purpose systems modeling language that can be used to create and manage models of systems using well-defined constructs with underlying semantics and a graphical notation. SysML reuses a subset of UML 2 constructs and extends them by adding new modeling elements and two new diagram types. These SysML diagrams are shown in Figure 1.1. The set of behavioral and structural diagrams combined with the requirements diagram and parametric diagram provide an integrated view of a system. But SysML represents much more than just a set of diagrams. Underlying the diagrams, there is an abstract syntax model repository that formally represents all the modeling constructs. The graphical model provides a mechanism to organize, enter, retrieve, and view the system-descriptive data contained in the model repository.

The diagrams provide multiple views of the same system model; these multiple views can be maintained consistently due to the semantic underpinning of the modeling language. In the context of SysML:

- The structure view primarily refers to the hierarchy and interconnections among the parts of the system, and the interconnections between the system and its external systems.
- The behavior view describes how the state of the system changes (or must change) over the time according to its own dynamics and/or to external events.

The requirements diagram captures text requirements in the model and enables them to be linked to other parts of the model to provide unambiguous traceability between the requirements and system design. Parametrics provide a means to specify that interdependencies between values of some system properties and can provide a bridge between the system descriptive model in SysML and other simulation and engineering analysis models. While structure and behavior are heavily based on UML, both requirements and parametrics are unique to SysML. Through these extensions, SysML is capable of representing the specification, analysis, design, verification, and validation of systems.

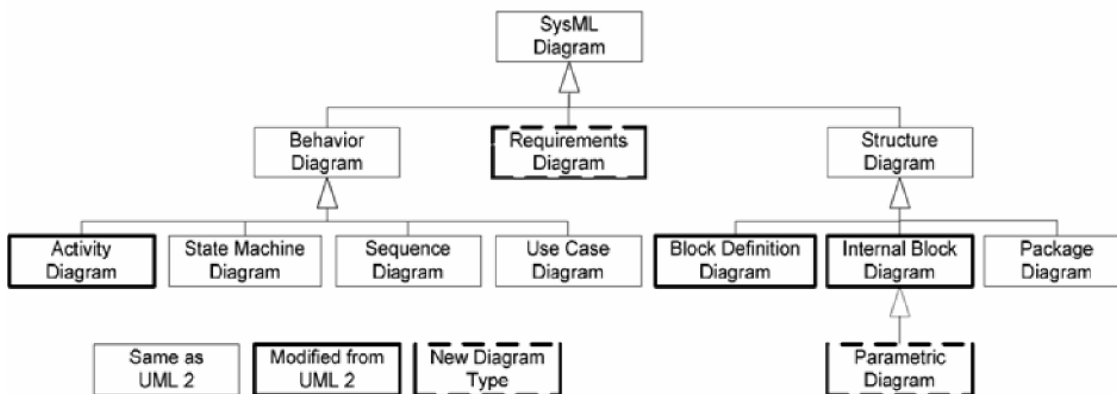


Figure 1.1 - An overview of the SysML diagrams and their relation to UML diagrams

As indicated above, the system behavior in SysML is captured through a combination of activity graphs, state machine, and/or interactions specifications using diagrams and their associated semantics. The Semantics of a Foundational Subset for Executable UML Models (<http://www.omg.org/spec/FUML>) provides more formal semantics to enable SysML activity models to be executed in better compliance with the standard. In addition, SysML includes parametric constructs to capture models of constraint-based behavior, such as continuous-time dynamics in terms of energy flow. The syntax and

semantics of such behavioral descriptions in parametrics have been left open to integrate with other simulation and analysis modeling capabilities to support the execution of these models. Additional information on SysML can be found at <http://www.omg.sysml.org>.

## 1.2 SysML-Modelica

Modelica is an object-oriented language for describing differential algebraic equation (DAE) systems combined with discrete events. Such models are ideally suited for representing the flow of energy, materials, signals, or other continuous interactions between system components. It is similar in structure to SysML 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 ([www.modelica.org](http://www.modelica.org)), which publishes a formal specification [Modelica Association, 2008] but also provides an extensive Modelica Standard Library, which 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 ([www.openmodelica.org](http://www.openmodelica.org)).

## 1.3 Conclusion

In conclusion, SysML and Modelica are two complementary languages supported by two active communities. By integrating SysML and Modelica, we combine the very expressive, formal language for differential algebraic equations and discrete events of Modelica with the very expressive SysML constructs for requirements, structural decomposition, logical behavior and corresponding cross-cutting constructs. In addition, the two communities are expected to benefit from the exchange of multi-domain model libraries and the potential for improved and expanded commercial and open-source tool support.

## 1.4 Objective

The objective of this document is to provide a bi-directional mapping between SysML and Modelica to leverage the benefits from both languages. By integrating SysML and Modelica, SysML's strength in descriptive modeling can be combined with Modelica's DAE solving capability to support analyses and trade studies. The scope of this specification supports the objectives of the bi-directional mapping, and includes the SysML4Modelica profile, and the SysML-Modelica Transformation. Not all Modelica constructs will be represented in this profile. The focus is to include the Modelica language features that are most common and together cover the majority of the Modelica models in the standard library. When certain Modelica constructs are omitted, this will be pointed out explicitly in this document. In the future, it may be desirable to introduce additional SysML constructs into the Modelica Language or additional Modelica constructs in the SysML language; however, this is outside the scope of the current effort.

# 2 Conformance

This specification has a narrow scope, focusing exclusively on the transformation between SysML4Modelica and Modelica. Partial support of this specification is therefore of limited use. Still, it is useful to distinguish between the following two compliance levels:

- Level 0: Compliance with SysML4Modelica profile
- Level 1: Compliance with the SysML-Modelica mapping



Compliance to Level 0: This level entails the support of all the modeling concepts included in the SysML4Modelica profile as defined in Part II of this specification.

Compliance to Level 1: In addition to the capabilities provided by Level 0, Level 1 supports the successful bi-directional transformation between SysML4Modelica models and corresponding Modelica models. The bi-directional transformation is considered successful if the original Modelica model and the Modelica model generated by the round-trip transformation (Modelica->SysML4Modelica->Modelica) are semantically identical, that is, they result in the same simulation results.

### 3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- Systems Modeling Language: Specification, v1 .3 (<http://www.omg.org/spec/SysML/1.3>)
- Modelica Specification, v.3.1 (<http://www.modelica.org/documents/ModelicaSpec31.pdf>)
- QVT, v1.1(<http://www.omg.org/spec/QVT/1.1/>)
- OCL, v2.3. 1 (<http://www.omg.org/spec/OCL/2.3.1/>)

### 4 Terms and Definitions

There are no formal definitions in this specification that are taken from other documents. For an overview of Modelica related terms and definitions, refer to Annex A.

### 5 Symbols

Acronym	Me
MDA	Model Driven Architecture
MOF	Meta Object Facility
OMG	Object Management Group
SysML	System Modeling Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

## 6 Additional Information

### 6.1 Acknowledgments

The following companies submitted this specification:

- Atego
- Deere & Company
- No Magic Inc.

The following companies supported this specification:

- EADS
- ESA/ESTEC
- Georgia Institute of Technology
- Jet Propulsion Laboratory
- Linköping University
- Lockheed Martin Corporation

The following people have contributed significantly to this document either directly or indirectly through discussions and feedback:

- Yves Bernard (EADS)
- Conrad Bock (NIST)
- Roger Burkhart (Deere & Co)
- Hans-Peter De Koning (ESA)
- Sanford Friedenthal (Lockheed Martin)
- Peter Fritzson (Linköping University)
- Nerijus Jankevicius (No Magic Inc)
- Thomas Johnson (Georgia Tech)
- Alek Kerzhner (Georgia Tech)
- Alan Moore (Mathworks)
- Chris Paredis (Georgia Tech)
- Russell Peak (InterCAx, Georgia Tech)
- Axel Reichwein (Georgia Tech)
- Nicolas Rouquette (Jet Propulsion Laboratory)
- Wladimir Schamai (EADS, Linköping University)

# 7 Transformation Approach

## 7.1 General

To develop a transformation between the SysML and Modelica languages, a formal, systematic approach is used, as is illustrated in Figure 7.1. The transformation approach is to specify first an extension to SysML called the SysML4Modelica profile which represents the most common Modelica language constructs. This allows the Modelica concepts to be expressed in an extension of SysML that supports round-trip transformations between SysML and Modelica. The profile extends the UML4SysML subset of UML and the SysML extensions to provide the concept required to capture the relevant Modelica concepts and enable the mapping between the two languages. The “transformation” stereotype is a self-defined stereotype referring to a mapping between the Modelica metamodel and the SysML4Modelica profile. The <<conformsTo>>, <<transformation>>, and <<instanceOf>> stereotypes are purely informal.

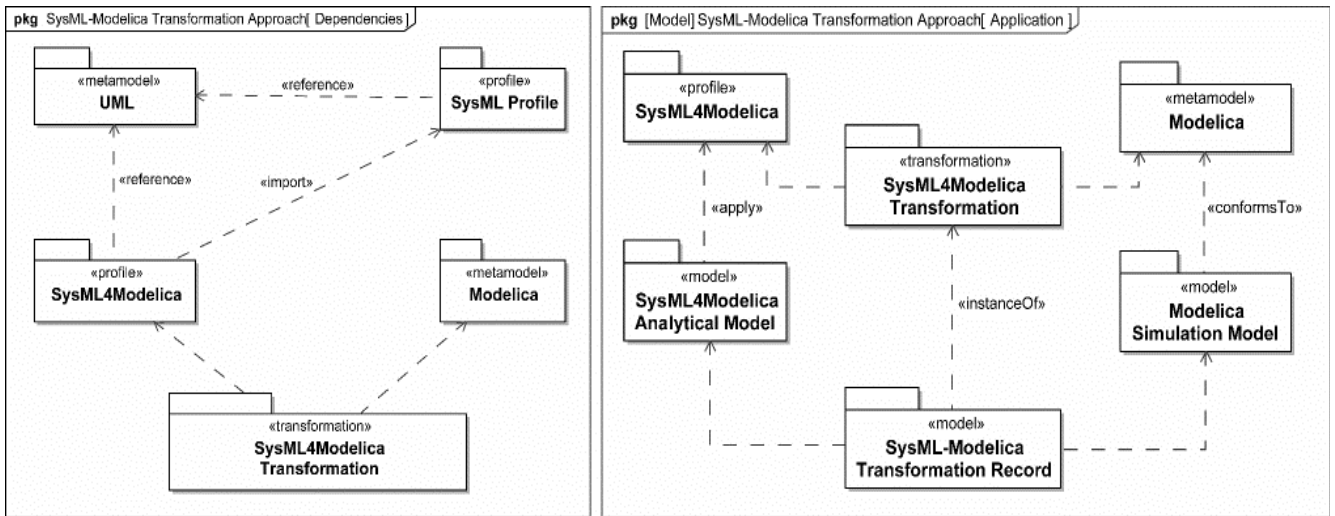


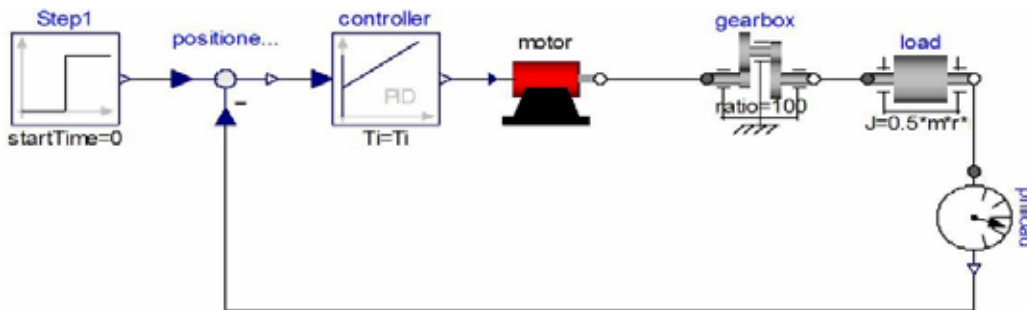
Figure 7.1 - The SysML-Modelica Transformation in relation to SysML and Modelica

To develop the SysML4Modelica profile in a systematic fashion, we start from the Modelica Language Specification and identify for each Modelica language construct an equivalent construct in SysML from a semantic point of view. Where equivalent constructs do not exist, stereotypes are created to extend the SysML language. The following naming convention is used to define a Modelica construct in the SysML4Modelica profile: “modelicaConstruct” where Construct is the name of the Modelica language construct as defined in the Modelica abstract syntax definition (see Part III - Modelica Abstract Syntax).

Even when an equivalent SysML construct exists, it is sometimes necessary to introduce a stereotype in order to distinguish the Modelica construct from the ordinary SysML construct when supporting round-trip transformation. In addition, the textual syntax of Modelica often provides alternative ways to express the exact same semantics. In such cases, the intent is to avoid propagating this redundancy to SysML4Modelica without loss of expressivity. For mapping purposes, one of the redundant textual notations is identified as the primary (most explicit) one, and SysML4Modelica constructs are preferably shown in this primary notation when using Modelica textual syntax. It should also be noted, that

Modelica includes a graphical syntax using iconic representations of block diagrams that maps to its textual syntax. An example of the Modelica graphical syntax is shown in Figure 7.2 for a set of components connected together via Modelica connectors and connections.

Modelica has a very rich representation for modeling differential algebraic equations. Where Modelica has a concept that cannot be directly transformed into SysML, an opaque expression in Modelica syntax is sometimes used to capture the concept in the SysML4Modelica profile. For example, mathematical expressions appearing in Modelica models are represented as opaque expressions in the corresponding SysML4Modelica models.



**Figure 7.2 - A Modelica model of a motor controller consisting of component models and the connections between them. The connections include both causal signal connections (e.g., in and out of the controller) and acausal energy connections (e.g., the rotational mechanical energy connections of the gearbox).**

This specification provides a textual description of the mapping between Modelica and SysML4Modelica (see Part IV - Transformation). In Annex C, this mapping is also (partially) described using QVT. Such a formal definition of the mapping has the advantage that tools can be used to generate executable transformations between SysML and Modelica modeling tools (assuming they support some standardized interface such as JMI<sup>1</sup>). An additional implementation of the mapping is being developed as part of the OpenModelica project.

---

1. Java Metadata Interface (JMI), <http://java.sun.com/products/jmi/>

# Part II - SysML4Modelica Profile

## Overview

This part describes the stereotypes that represent the Modelica modeling constructs in SysML. As illustrated below, the stereotypes, together with the library of predefined types, are organized in sub-packages and profiles in the SysML4Modelica profile. In Clause 8, all the stereotypes related to the Modelica restricted classes are introduced. In Clause 9, the predefined Modelica types and the enumerations used in the SysML4Modelica profile are defined. In Clause 10, the Modelica equivalent of properties are defined - called Component Declarations in Modelica. Finally, in Clause 11, the Equation and Algorithm sub clauses of Modelica models are covered.

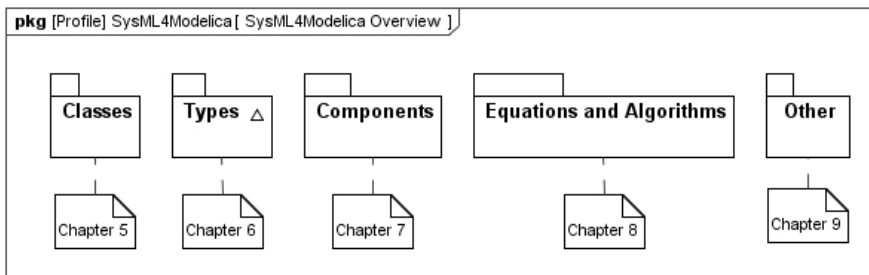


Figure (Part II) - Overview



# 8 Class Definition

## 8.1 Overview

The class concept is the basic structural unit in Modelica. Classes provide the structure for objects and contain equations and algorithms, which ultimately are the basis for the executable simulation code. The most general class is “model.” Specialized classes such as “record,” “type,” “block,” “package,” “function,” and “connector” have most of the properties of a “model” but with restrictions, which need to be preserved in SysML to support round-trip mapping.

The following production rules define the different specialized classes. The reference in parentheses on the right indicates the clause of this document in which the particular language element is discussed in detail.

```
[ within [ name ] ";" ] (9)
{ [ final ] class_definition ";" } (9)

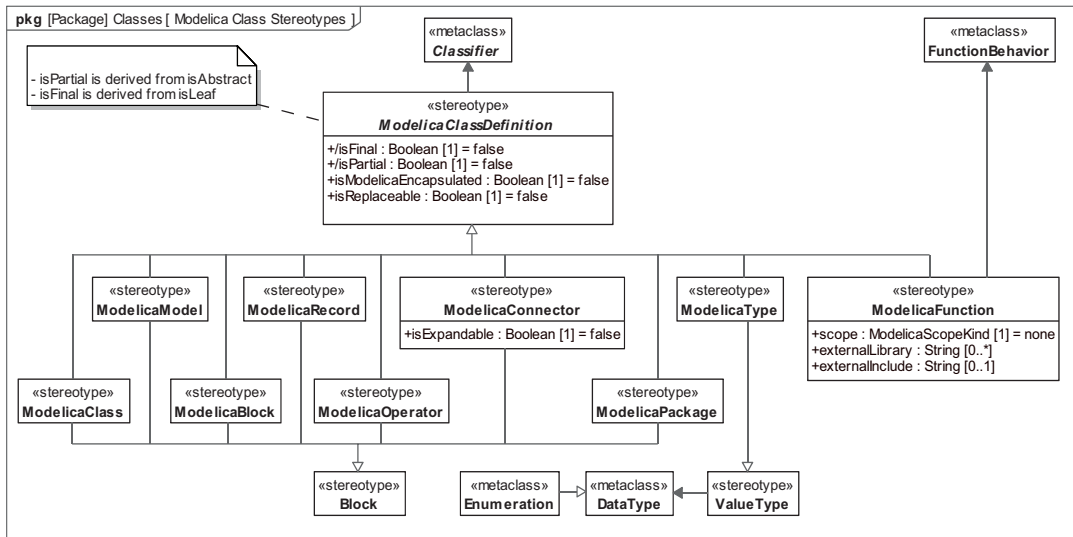
class_definition :
  [ encapsulated ] (9)
  [ partial ] (9)
  ( class
  | model
  | record
  | block
  | [ expandable ] connector (12)
  | type
  | package
  | function ) (14)

class_specifier
class_specifier :
  IDENT string_comment composition (23)
  | IDENT "=" base_prefix name [ array_subscripts ] (13)
  [ class_modification ] comment (23)
  | IDENT "=" enumeration "(" ( [enum_list] | ":" ) ")" comment (13)
  | IDENT "=" der "(" name "," IDENT { "," IDENT } ")" comment (14)
  | extends IDENT [ class_modification ] string_comment composition (16)
end IDENT
```

The following table lists the SysML stereotypes for representing the specialized Modelica classes. Using this approach the modeler only needs to apply the respective stereotype to indicate all the semantics and restrictions of the associated Modelica class. This information is represented graphically in Figure 8.1. In the following sub clauses, the details of each stereotype are described.

**Table 8.1 - Mapping for the Modelica specialized classes**

Modelica Construct	SysML Base Class	SysML4Modelica	
		New Stereotype	See Clause
abstract generalization for all Modelica classes	UML4SysML: :Classifier	«modelicaClassDefinition»	9
Class and Model	SysML::Blocks::Block	«modelicaModel»	10
Record	SysML::Blocks::Block	«modelicaRecord»	11
Block	SysML::Blocks::Block	«modelicaBlock»	12
Connector	SysML::Blocks::Block	«modelicaConnector»	12
Type	SysML::Blocks: :Block SysML::Blocks::ValueType UML4SysML: :Enumeration	«modelicaType»	13
Package	SysML::Blocks::Block	«modelicaPackage»	14
Function	UML4SysML: :FunctionBehavior	«modelicaFunction»	14



**Figure 8.1 - Package diagram with an overview of the stereotypes for Modelica Classes**



## 8.2 «modelicaClassDefinition»

### Stereotypes

- Classifier (from UML4SysML)

### Abstract Syntax

- See Figure 8.1

### Description

A Modelica `class` is the basic structural unit in Modelica. However, because it lacks precise semantics, the `class` construct should never be used in Modelica. Without precise semantics, a Modelica tool cannot easily check whether any restrictions are violated. Therefore, the constructs that are specialized from Modelica `class` should be used instead.

In the context of the SysML4Modelica profile, the Modelica `class` construct is mapped to the stereotype «modelicaClassDefinition», which is abstract and thus cannot be instantiated directly. This choice has been made because it is desirable to have the additional semantics specified by the specialized classes. In addition, as clearly shown in Figure 8.1, the stereotypes associated with the specialized classes derive from different SysML constructs and thus cannot be mapped to a single common construct for a Modelica `class`. The abstract stereotype «modelicaClassDefinition» serves the purpose of grouping the attributes that apply to all the Modelica specialized classes. It stereotypes UML::Classifier, which is a common generalization for the stereotypes of all the specialized classes.

Just like UML Classifiers, a «modelicaClassDefinition» can contain nested class definitions. Such nested definitions can be of any restricted class type derived from «modelicaClassDefinition». For instance, a «modelicaConnector» can contain a «modelicaPackage».

Modelica classes are often defined using a short class definition syntax. For example, the type `Force` could be defined as:

```
type Force = Real [3] (unit={ "N.m" , "N.m" , "N.m" } );
```

Rather than supporting such short class definitions explicitly, the SysML4Modelica profile supports only the longer (but equivalent) form (Note: in the Modelica abstract syntax the two forms are often represented identically):

```
type Force
  extends Real [3] (unit={ "N .m" , "N .m" , "N .m" });
end Force;
```

In the remainder of this sub clause, all the common attributes and associations for all the constructs specialized from Modelica `class` are described. In subsequent clauses for the individual specialized constructs, only the constraints on these attributes and associations will be described in detail.

### Attributes

- `/isFinal` : Boolean [1]

In Modelica, the definition of a class can be qualified to be final (see Modelica, v3.1, sub clause 7.2.6). This means that the declared class cannot be further modified through (local) type modifications. Note that this is identical to the UML attribute `isLeaf` for redefinable elements (see UML Specification 7.3.46) which, if true, indicates that no further redefinitions are possible.

The `isFinal` attribute is true when the final prefix is present in Modelica; false otherwise. Its default value is *false*.

This is derived from *isLeaf*.

- `/isPartial` : Boolean [1]  
The Modelica partial construct has the same semantics as the `isAbstract` attribute in SysML. The `isPartial` attribute is true when the partial prefix is present in Modelica; false otherwise. Its default value is *false*. This is derived from `isAbstract`.
- `isModelicaEncapsulated` : Boolean [1]  
As explained in Modelica Specification 5.3.2, the Modelica encapsulated construct limits the scope of name lookup. An encapsulated package can be moved within the package hierarchy without affecting the local name resolutions. These semantics are different from the `isEncapsulated` attribute of Blocks in SysML (SysML Specification 8.3.2.2). An encapsulated block is treated as a black box; no connections can be made to its internal parts directly. A second difference in semantics is that in Modelica the encapsulated prefix can be applied to *all* classes, although it is most commonly applied to packages. It is therefore necessary to introduce `isModelicaEncapsulated` as a new attribute so that it becomes available also for specialized class stereotypes that do not derive from a SysML Block.  
  
The `isModelicaEncapsulated` attribute is true when the encapsulated prefix is present in Modelica; false otherwise. Its default value is *false*.
- `isReplaceable` : Boolean [1]  
As explained in the Modelica Specification 7.3, the Modelica prefix `replaceable` is most commonly applied to components (see sub clause 23) , but can also be applied to a Modelica class to indicate that a local model definition can be redeclared when the containing model is used. The `isReplaceable` attribute is true when the `replaceable` prefix is present in Modelica; false otherwise. Its default value is *false*.

#### Associations

No additional associations

#### Constraints

- [ 1 ] Any generalization relationship to/from `«modelicaClassDefinition»` must be stereotyped by a `«modelicaExtends»` relationship.
- [ 2 ] A `«modelicaClassDefinition»` can only contain nestedClassifiers stereotyped by a restricted type specializing `«modelicaClassDefinition»`.

#### Additional Notes

The Modelica `within` clause is explained in the Modelica Specification, v 3.1, sub clause 13.2.2.3. It defines where in the package hierarchy the subsequent class definitions are located. This is important in Modelica to allow large package structures to be divided over multiple model files. As long as fully qualified type identifiers are used, the `within` clause is not relevant in SysML4Modelica and is therefore not supported in the SysML4Modelica profile.

## 8.3 «modelicaClass» and «modelicaModel»

#### Generalizations

- `«modelicaClassDefinition»` (from SysML4Modelica: :Classes)
- `«block»` (from SysML)

#### Abstract Syntax

- See Figure 8.1

## Description

The Modelica specialized class model is the most general specialized class; it is equivalent to the general Modelica class construct. All the Modelica class elements are allowed in models: variables, connectors, sub-models, equations, and algorithm sections. A model can also include state variables. Modelica does not differentiate between a model and a class. Although redundant, we therefore include both the equivalent stereotypes «modelicaClass» and «modelicaModel».

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

(All constraints apply to both «modelicaClass» and «modelicaModel»)

- [ 1 ] A «modelicaModel» must have a Name.
- [ 2 ] A «modelicaModel» can only have Properties that are stereotyped by «modelicaPart», «modelicaPort», or «modelicaValueProperty».
- [ 3 ] A «modelicaModel» can only contain Behaviors that are stereotyped by «modelicaFunction», or «modelicaAlgorithm».
- [ 4 ] A «modelicaModel» can only be contained in a «modelicaClassDefinition».
- [ 5 ] A «modelicaModel» can only specialize other classifiers derived from «modelicaBlock», or «modelicaRecord». The stereotype «modelicaExtends» must be applied to the generalization relationship.
- [ 6 ] All other attributes or associations inherited from «block» or Classifier are not relevant and should be set to their default values. This includes the attributes: isActive, isEncapsulated.

## 8.4 «modelicaRecord»

### Generalizations

- «modelicaClassDefinition» (from SysML4Modelica: :Classes)
- «block» (from SysML)

### Abstract Syntax

- See Figure 8.1

### Description

The Modelica specialized class `record` is restricted to contain only public declarations of components that in turn also contain only public declarations. A complete description of `record` is available in the Modelica Specification, v3.1, sub clause 4.6:

---

Only public sections are allowed in the definition or in any of its components (i.e., equation, algorithm, initial equation, initial algorithm and protected sections are not allowed). May not be used in connections. The elements of a record may not have prefixes input, output, inner, outer, or flow. Enhanced with implicitly available record constructor function. Additionally, record components can be used as component references in expressions and in the left hand side of assignments, subject to normal type compatibility rules.

---

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

- [ 1 ] A «modelicaRecord» must have a Name.
- [ 2 ] A «modelicaRecord» can only have Properties that are stereotyped by «modelicaValueProperty».
- [ 3 ] Any «modelicaValueProperty» owned by an instance of «modelicaRecord» must have *visibility=public, flowFlag=nonflow, causality=null, scope=null*.
- [ 4 ] A «modelicaRecord» can only be contained in a «modelicaClassDefinition».
- [ 5 ] A «modelicaRecord» can only specialize other classifiers derived from «modelicaRecord». The stereotype «modelicaExtends» must be applied to the generalization relationship.
- [ 6 ] All other attributes or associations inherited from «block» or Classifier may not be used. This includes the attributes: isActive, isEncapsulated; and the ownedElements: Behavior, Constraint.

## 8.5 «modelicaBlock»

### Generalizations

- «modelicaClassDefinition» (from SysML4Modelica: :Classes)
- «block» (from SysML)

### Abstract Syntax

- See Figure 8.1

### Description

The Modelica specialized class block is very similar to a model except that all its connectors must be either an input or output making it similar to a Simulink block. A complete description of block is available in sub clause 4.6 of the Modelica Specification:

---

Same as model with the restriction that each connector component of a Modelica block must have prefixes input and/or output for all connector variables.

---

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

- [ 1 ] A «modelicaBlock» must have a Name.
- [ 2 ] A «modelicaBlock» can only have Properties that are stereotyped by «modelicaPart», «modelicaPort», or «modelicaValueProperty».
- [ 3 ] Any «modelicaValueProperty» owned by an instance of «modelicaBlock» must have *causality= input* or *output*.
- [ 4 ] A «modelicaBlock» can only contain Behaviors that are stereotyped by «modelicaFunction», «modelicaAlgorithm», or «modelicaInitialAlgorithm».
- [ 5 ] A «modelicaBlock» can only contain Constraints that are stereotyped by «modelicaEquation» or «modelicaInitialEquation».
- [ 6 ] A «modelicaBlock» can only be contained in a «modelicaClassDefinition».
- [ 7 ] A «modelicaBlock» can only specialize other classifiers derived from «modelicaBlock» or «modelicaRecord». The stereotype «modelicaExtends» must be applied to the generalization relationship.
- [ 8 ] All other attributes or associations inherited from «block» or Classifier may not be used. This includes the attributes: *isActive*, *isEncapsulated*.

## 8.6 «modelicaConnector»

### Generalizations

- «modelicaClassDefinition» (from SysML4Modelica: :Classes)
- «block» (from SysML)

### Abstract Syntax

- See Figure 8.1

### Description

The Modelica specialized class `connector` is a `model` that cannot contain equations or algorithms in any of its components. A complete description of `connector` is available in sub clause 4.6 and Clause 9 of the Modelica Specification:

---

No equations or algorithms are allowed in the definition or in any of its components.  
Enhanced to allow `connect(..)` to components of connector classes.

---

### Attributes

- `isExpandable` : Boolean [1]  
As explained in the Modelica Specification, sub clause 9.1.3, the Modelica `expandable` prefix can be applied to a `connector`. The primary purpose of `expandable` connectors is to allow for the convenient modeling of bus interfaces. The `isExpandable` attribute is true when the `expandable` prefix is present in Modelica; false otherwise. The default value is false.

## Associations

No additional associations

## Constraints

- [ 1 ] A «modelicaConnector» must have a Name.
- [ 2 ] A «modelicaConnector» can only have Properties that are stereotyped by «modelicaPart», «modelicaPort», or «modelicaValueProperty».
- [ 3 ] None of the Properties owned by a «modelicaConnector» can be typed to «modelicaClassDefinition»s that contain Behaviors or Constraints (at any level of containment).
- [ 4 ] A «modelicaConnector» can only be contained in a «modelicaClassDefinition».
- [ 5 ] A «modelicaConnector» can only specialize other classifiers derived from «modelicaConnector», «modelicaType», or «modelicaRecord». The stereotype «modelicaExtends» must be applied to the generalization relationship.
- [ 6 ] All other attributes or associations inherited from «block» or Classifier may not be used. This includes the attributes: isActive, isEncapsulated; and the ownedElements: Behavior, Constraint.

## 8.7 «modelicaType»

### Generalizations

- «modelicaClassDefinition» (from SysML4Modelica: :Classes)
- «valueType» (from SysML)

### Abstract Syntax

- See Figure 8.1

### Description

The Modelica specialized class `type` is restricted to predefined types, enumerations, arrays of type or classes extending from type. It is enhanced to allow extension of predefined types. In the SysML4Modelica profile, the extension from predefined types is handled by making the predefined types instances of «modelicaType» (See Clause 18).

Unlike the other Modelica restricted classes, «modelicaType» does not generalize «block». This implies that it is not possible for a «modelicaType» to contain definitions of other modeling elements (e.g., a contained package). Although such containment would, strictly speaking, be allowed by the Modelica language, it is rarely, if ever, used. To avoid unnecessary complications in extending SysML «valueType»s, the SysML4Modelica profile does not support «modelicaType»s that contain definitions of other modeling constructs.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

- [ 1 ] A «modelicaType» must have a Name.
- [ 2 ] A «modelicaType» can only be contained in a «modelicaClassDefinition».

[ 3 ] A «modelicaType» can only specialize other classifiers derived from «modelicaType». The stereotype «modelicaExtends» must be applied to the generalization relationship.

## 8.8 «modelicaPackage»

### Generalizations

- «modelicaClassDefinition» (from SysML4Modelica: :Classes)
- «block» (from SysML)

### Abstract Syntax

- See Figure 8.1

### Description

A Modelica package has broader semantics than just a container for other model elements as in SysML. Although it may only contain declarations of classes and constants, these declarations can be replaceable and can be inherited from parent packages, so that the package itself should be thought of as a model. The corresponding SysML4Modelica construct, «modelicaPackage», therefore generalizes «block» rather than Package. In the Modelica language, a Modelica package is enhanced, as compared to Modelica class, to allow for the import of elements of packages. (See Modelica, v 3.1, Clause 13.)

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

- [ 1 ] A «modelicaPackage» must have a Name.
- [ 2 ] A «modelicaPackage» can only have Properties that are stereotyped by «modelicaValueProperty».
- [ 3 ] Any «modelicaValueProperty» owned by an instance of «modelicaPackage» must have *variability= constant*. (ref. Modelica Specification 4.6, package)
- [ 4 ] A «modelicaPackage» can be contained in a «modelicaClassDefinition» or in a UML4SysML: :Package.
- [ 5 ] A «modelicaPackage» can only specialize other classifiers derived from «modelicaPackage». The stereotype «modelicaExtends» must be applied to the generalization relationship.
- [ 6 ] All other attributes or associations inherited from «block» or Classifier may not be used. This includes the attributes: isActive, isEncapsulated; and the ownedElements: Behavior, Constraint.

## 8.9 «modelicaFunction»

### Extensions

- FunctionBehavior (from UML4SysML)

### Generalizations

- «modelicaClassDefinition» (from SysML4Modelica: :Classes)

## Abstract Syntax

- See Figure 8.1

## Description

The Modelica specialized class function represents a callable section of procedural algorithmic code without side effects. It is similar to a SysML FunctionBehavior. Compared to a general Modelica class, quite a few restrictions and enhancements apply; refer to Modelica, v 3.1, sub clause 12.2 for details.

As described in Modelica, v 3.1, sub clause 12.9, a Modelica function may refer to an external function specifier (e.g., an external C or Fortran function):

```
function IDENT string_comment
{ component _clause ";" }
[ protected { component_clause ";" } ]
external [ language_specification ] [ external_function_call ] [annotation ] ";"
[ annotation ";" ]
end IDENT;
```

Whether a particular function is external or not is determined by the language attribute of the «modelicaFunction» (inherited from OpaqueBehavior). For Modelica native functions, the language should be specified as “Modelica.” For external functions, the language attribute is set to another language. Modelica currently only allows the languages “C,” “FORTRAN,” or “builtin.” For such external functions, the body attribute contains the external `_function_call` from Modelica. The annotation that is part of the external statement in Modelica can contain two types of information: Libraries and Include directives. In SysML4Modelica, this information is captured in the two additional attributes: *externalLibrary* and *externalInclude*.

Several additional attributes are included in «modelicaFunction» to capture such semantics.

At this point, SysML4Modelica only allows for function definitions; functions cannot be “called” explicitly – they can only be referred to in opaque Modelica syntax portions of the model.

## Attributes

- `externalLibrary`: String [0..\*]  
A list of external libraries that need to be linked in to resolve the references to the external function (see Modelica, v 3.1, sub clause 12.9 for details). It should only be defined when *language* = “C” or “FORTRAN”.
- `externalInclude`: String [0.. 1]  
An optional string containing include directives to be considered when compiling and linking the external function. It should only be defined when *language* = “C” or “FORTRAN.”

## Associations

No additional associations

## Constraints

- [ 1 ] A «modelicaFunction» must have a Name.
- [ 2 ] A «modelicaFunction» can only have Parameters that are stereotyped by «modelicaFunctionParameter».
- [ 3 ] Any «modelicaFunctionParameter» (owned by an instance of «modelicaPackage») for which *causality*= *input* may not be assigned values in the body of the function (i.e., it is read-only).
- [ 4 ] A «modelicaFunction» can only have zero or one *body* attribute.
- [ 5 ] A «modelicaFunction» must have *language*= “Modelica,” “builtin,” “C,” or “FORTRAN.”



- [ 6] If *language*= “*Modelica*,” then the *body* of the function must be represented in the Modelica syntax and must constitute a valid Modelica algorithm section.
- [ 7] If *language*= “*C*” or “*FORTRAN*,” then the *body* of the function must be represented a valid functional call in the respective language (as specified in Modelica, v 3.1, sub clause 12.9.4).
- [ 8] The optional attributes, *externalLibrary* and *externalInclude*, can only be used when *language*= “*C*” or “*FORTRAN*.” [9] A «modelicaFunction» definition can only be contained in a «modelicaClassDefinition».
- [ 9] A «modelicaFunction» can only specialize other classifiers derived from «modelicaFunction». The stereotype «modelicaExtends» must be applied to the generalization relationship.
- [ 10] All other attributes or associations inherited from FunctionBehavior or Classifier may not be used.

## 8.10 «modelicaExtends»

### Extensions

- Generalization (from UML4SysML)

### Abstract Syntax

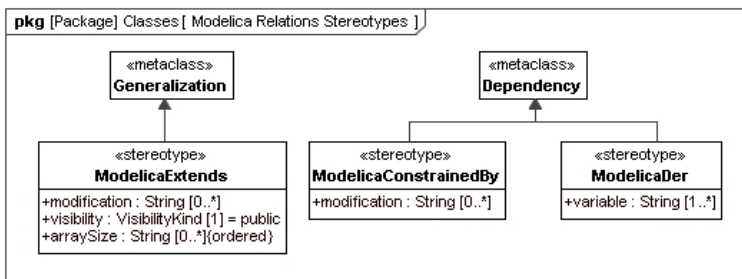


Figure 8.2 - Modelica Relations stereotype definitions

### Description

The extends clause of Modelica is equivalent to a SysML Generalization. The only difference is that in Modelica the type being extended can be locally modified (see Modelica, v 3.1, sub clause 7.1):

```

extends_clause :
  extends name [ class_modification ] [annotation]

constraining_clause :
  extends name [ class_modification ]

```

Similar local type modifications can be used when defining usages (i.e., Modelica components – see Clause 23). In both cases the SysML4Modelica mapping currently captures the local modifications only as a text string in Modelica syntax. A separate modification can be defined for every component of a «modelicaClassDefinition»; in Modelica these modifications are grouped, separated by commas, and surrounded by parentheses. Each such modification is represented in SysML4Modelica as a separate string. It corresponds thus to an argument as defined in the following extract of the Modelica EBNF (see Modelica, v 3.1, sub clause 7.2):

```

class_modification :
  "(" [ argument_list ] ")"

argument_list :
  argument { "," argument }

argument :
  element_modification_or_replaceable |
  element_redeclaration

element_modification_or_replaceable:
  [ each ] [ final ] ( element_modification | element_replaceable)

element_modification :
  component_reference [ modification ] string_comment

element_redeclaration :
  redeclare [ each ] [ final ]
  ( ( class_definition | component_clause1) | element_replaceable )

element_replaceable:
  replaceable ( class_definition | component_clause1)
  [constraining_clause]

component_clause1 :
  type_prefix type_specifier component_declaration1

component_declaration1 :
  declaration comment

```

Multiple inheritance is supported in Modelica. Therefore, more than one «modelicaExtends» relationship is allowed for a single «modelicaClassDefinition». The extends clause can be applied to any of the restricted classes (including packages).

If the extends clause appears in a protected section of the Modelica model, then all the elements of the base class become protected elements of the specialized class. It is therefore important to specify whether the «modelicaExtends» relation is *public* or *protected*.

Not every restricted class can inherit from every other restricted class. Refer to Modelica, v 3.1, sub clause 7.1.3 for an overview table.

### Attributes

- **visibility:** VisibilityKind [1]  
When an extends statement appears in a protected section of a «modelicaClassDefinition», then all components of the parent class are protected. Default value is *public*.
- **modification:** String [0..\*]  
An inherited Modelica class can be locally modified. The modifications are defined by this attribute in Modelica syntax. Each modification (as specified in the Modelica concrete syntax as a comma-separated expression) is specified as a separate instance of this attribute.
- **arraySize:** String [0..\*] {ordered}  
One can specify an array size for an inherited Modelica class. This attribute is an ordered list of strings, each of which must be a Modelica expression that evaluates to an integer. The  $i^{th}$  element in the ordered list corresponds to size of the multidimensional array in the  $i^{th}$  dimension.

## Associations

No additional associations

## Constraints

- [ 1] Both the *source* and *target* of a «modelicaExtends» relation must be typed to instances of a specialization of «modelicaClassDefinition».
- [ 2] The *visibility* attribute of «modelicaExtends» can only take on values of *public* or *protected*.

## 8.11 «modelicaDer»

### Extensions

- Dependency (from UML4SysML)

### Abstract Syntax

- See Figure 8.2

### Description

The `der` clause in Modelica identifies a function as a partial derivative of another function (Modelica Specification 3.1, sub clause 12.7.2). It establishes a relationship between two functions and is therefore modeled as an extension of Dependency in SysML4Modelica. It requires as attributes a list of variables with respect to which the partial derivative is taken.

### Attributes

- `variable: String [1..*]`  
A list of variables with respect to which the partial derivative is taken. At least one variable must be specified. No default value is specified.

### Associations

No additional associations

### Constraints

- [ 1] Both the *source* and *target* of a «modelicaDer» relation must be typed to instances of «modelicaFunction».

## 8.12 «modelicaConstrainedBy»

### Extensions

- Dependency (from UML4SysML)

### Abstract Syntax

- See Figure 8.2

### Description

In a replaceable declaration in Modelica, one can specify a constrained by clause. The semantics of this construct are explained in more detail in Modelica, v 3.1, sub clause 7.3.2.

### Attributes

- **modification:** String [0..\*]  
A Modelica class that constrains a replaceable declaration can be locally modified. The modifications are defined by this attribute in Modelica syntax. Each modification (as specified in the Modelica concrete syntax as a comma-separated expression) is specified as a separate instance of this attribute. Default value is *null*.

### Associations

No additional associations

### Constraints

- [ 1 ] Both the *source* and *target* of a «modelicaConstrainedBy» relation must be typed to instances of a specialization of «modelicaClassDefinition».

## 8.13 Short Class Definitions

Modelica provides a short-hand notation for definition of classes. It is equivalent to an inheritance construct, and is therefore redundant and not supported separately in the SysML4Modelica profile.

# 9 Predefined Types

## 9.1 Overview

The following predefined types are available in the Modelica language (Modelica. v 3.1, sub clause 4.8): Real Type, Integer Type, Boolean Type, String Type, Enumeration Types, StateSelect, ExternalObject, and Graphical Annotation Type. These primitive types are defined as predefined types in SysML4Modelica::Types::ModelicaPredefinedTypes.

Although these types have direct counterparts in SysML, they are redefined to account for the additional attributes associated with them in Modelica. Note that in Modelica, the properties such as “start,” “quantity,” etc., are not really equivalent to user- defined complex data-types. For instance, if one defines “Real x;” then one cannot refer to “x.min” in an equation. The only way one can specify a value for these special properties is as part of a type definition or local modification: e.g., “Real x(start=1, unit=“m”).

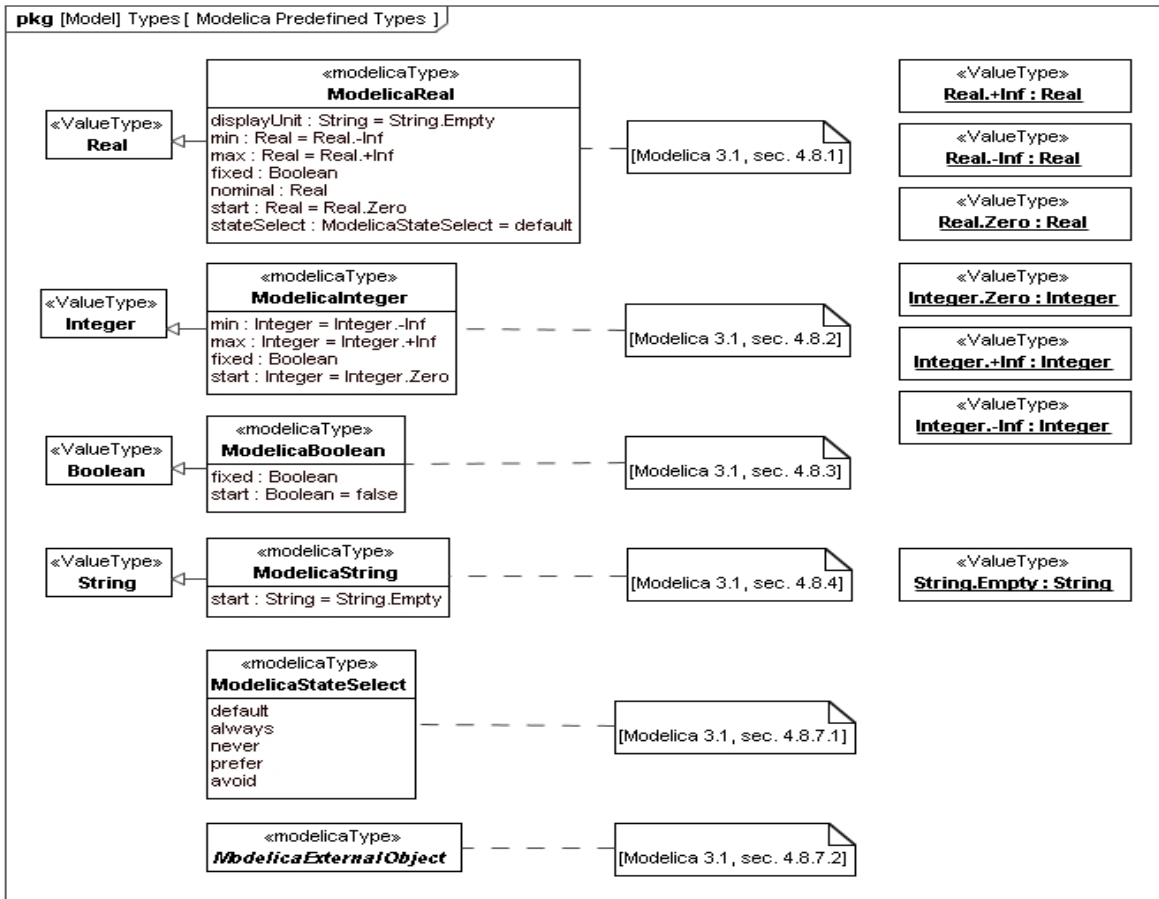


Figure 9.1 - Package diagram with an overview of the Predefined Modelica Types

## 9.2 ModelicaReal

### Instantiation

- SysML4Modelica::Classes::ModelicaType

### Generalizations

- SysML::Blocks::Real

### Abstract Syntax

- See Figure 9.1

### Description

The predefined type `Real` in Modelica includes a variety of attributes besides its actual value (Modelica 3.1, sub clause 4.8.1). In SysML4Modelica, these attributes are defined in `ModelicaReal`, a specialization of the primitive type `SysML::Blocks::Real`. As a result of this specialization, `ModelicaReal`, inherits the attributes: `quantityKind` and `unit`, which correspond to the Modelica attributes `quantity` and `unit`, respectively. Additional attributes are listed below.

### Attributes

- `displayUnit`: String [0.. 1]  
In addition to the actual units, a `ModelicaReal` can have a units used for display in a tool's graphical user interface or in plots. These units are defined in this attribute as a string.
- `min`: Real [1]  
The minimum value the `ModelicaReal` variable can take on. Default value is *-Inf*.
- `max`: Real [1]  
The maximum value the `ModelicaReal` variable can take on. Default value is *+Inf*.
- `start`: Real [1]  
The value of the `ModelicaReal` variable at the beginning of a simulation. The meaning of this variable depends on the value of the attribute `fixed`. If `fixed= false`, then it is to be interpreted as an initial guess from which may be deviated in order to satisfy all the algebraic constraints. If `fixed=true`, then the variable is required to equal its start value. Default value is *0*.
- `fixed`: Boolean [1]  
This attribute qualifies the meaning of the attribute `start`. If `fixed=false`, then `start` is to be interpreted as an initial guess from which may be deviated in order to satisfy all the algebraic constraints. If `fixed=true`, then the variable is required to equal its `start` value. Default value is *true* for parameters and constants, and *false* for all other variables.
- `nominal`: Real [0.. 1]  
The value of this attribute may be used by the solver for scaling purposes.
- `stateSelect`: StateSelect [1]  
The value of this attribute determines how a Modelica solver should select state variables for the system of Differential Algebraic Equations (Modelica. v 3.1, sub clause 4.8.7.1). Default value is *StateSelect.default*.

### Associations

No additional associations

## Constraints

No additional constraints

## 9.3 ModelicaInteger

### Instantiation

- SysML4Modelica::Classes::ModelicaType

### Generalizations

- SysML: :Blocks::Integer

### Abstract Syntax

- See Figure 9.1

### Description

The predefined type `Integer` in Modelica includes a variety of attributes besides its actual value (Modelica, v 3.1, sub clause 4.8.2). In SysML4Modelica, these attributes are defined in `ModelicaInteger`, a specialization of the primitive type `SysML::Blocks::Integer`. As a result of this specialization, `ModelicaInteger`, inherits the attribute: `quantityKind`, which correspond to the Modelica attribute `quantity`. Additional attributes are listed below.

### Attributes

- `min: Integer [1]`  
The minimum value the `ModelicaInteger` variable can take on. Default value is `-Inf`.
- `max: Integer [1]`  
The maximum value the `ModelicaInteger` variable can take on. Default value is `+Inf`.
- `start: Integer [1]`  
The value of the `ModelicaInteger` variable at the beginning of a simulation. The meaning of this variable depends on the value of the attribute `fixed`. If `fixed=false`, then it is to be interpreted as an initial guess from which may be deviated in order to satisfy all the algebraic constraints. If `fixed=true`, then the variable is required to equal its start value. Default value is `0`.
- `fixed: Boolean [1]`  
This attribute qualifies the meaning of the attribute `start`. If `fixed=false`, then `start` is to be interpreted as an initial guess from which may be deviated in order to satisfy all the algebraic constraints. If `fixed=true`, then the variable is required to equal its `start` value. Default value is `true` for parameters and constants, and `false` for all other variables.

### Associations

No additional associations

### Constraints

No additional constraints

## 9.4 ModelicaBoolean

### Instantiation

- SysML4Modelica::Classes::ModelicaType

### Generalizations

- SysML: :Blocks::Boolean

### Abstract Syntax

- See Figure 9.1

### Description

The predefined type `Boolean` in Modelica includes a variety of attributes besides its actual value (see Modelica, v 3.1, sub clause 4.8.3). In SysML4Modelica, these attributes are defined in `ModelicaBoolean`, a specialization of the primitive type `SysML: :Blocks::Boolean`. As a result of this specialization, `ModelicaBoolean`, inherits the attribute, `quantityKind`, which correspond to the Modelica attribute `quantity`. Additional attributes are listed below.

### Attributes

- `start: Boolean [1]`  
The value of the `ModelicaBoolean` variable at the beginning of a simulation. The meaning of this variable depends on the value of the attribute `fixed`. If `fixed=false`, then it is to be interpreted as an initial guess from which may be deviated in order to satisfy all the algebraic constraints. If `fixed=true`, then the variable is required to equal its start value. Default value is *false*.
- `fixed: Boolean [1]`  
This attribute qualifies the meaning of the attribute `start`. If `fixed=false`, then `start` is to be interpreted as an initial guess from which may be deviated in order to satisfy all the algebraic constraints. If `fixed=true`, then the variable is required to equal its start value. Default value is *true* for parameters and constants, and *false* for all other variables.

### Associations

No additional associations

### Constraints

No additional constraints

## 9.5 ModelicaString

### Instantiation

- SysML4Modelica::Classes::ModelicaType

### Generalizations

- SysML: :Blocks::String

### Abstract Syntax

- See Figure 9.1



## Description

The predefined type `String` in Modelica includes a variety of attributes besides its actual value (see Modelica, v3.1, sub clause 4.8.4). In SysML4Modelica, these attributes are defined in `ModelicaString`, a specialization of the primitive type `SysML::Blocks::String`. As a result of this specialization, `ModelicaString` inherits the attribute, `quantityKind`, which corresponds to the Modelica attribute `quantity`. In addition, a start value can be specified.

## Attributes

- `start: String [1]`  
The value of the `ModelicaString` variable at the beginning of a simulation. Default value is `String.Empty`.

## Associations

No additional associations

## Constraints

No additional constraints

## 9.6 ModelicaStateSelect

### Instantiation

- `SysML4Modelica::Classes::ModelicaType`

### Generalizations

No generalizations

### Abstract Syntax

- See Figure 9.1

### Description

The predefined type `ModelicaStateSelect` is the type of the attribute `stateSelect` of `ModelicaReal`. It is an enumeration used to provide guidance to the Modelica solver tool for selecting appropriate state variables (See Modelica, v 3.1, sub clause 4.8.7.1).

### Associations

No additional associations

### Constraints

No additional constraints

## 9.7 ModelicaExternalObject

### Instantiation

- `SysML4Modelica::Classes::ModelicaType`

### Generalizations

No generalizations

### **Abstract Syntax**

- See Figure 9.1

### **Description**

The predefined type `ModelicaExternalObject` is an abstract type used to indicate that a `ModelicaType` that specializes it refers to an object defined in an external language such as C or FORTRAN (See Modelica, v 3.1, sub clause 12.9.7 for details).

### **Associations**

No additional associations

### **Constraints**

[ 1] The value of the attribute *isAbstract* (and hence *isPartial*) must be *true*.

# 10 Component Declarations

## 10.1 Overview

In the Modelica language, instances (or usages) of a class are referred to as “Components.” In SysML, these can be mapped to Block Properties, such as Value Property, Part Property, or Port.<sup>1</sup> Modelica does not distinguish explicitly between Value Properties, Parts, or Ports. Instead, whether a component is interpreted as a Value Property, Part, or Port depends on the restricted type to which the usage has been typed:

- if the usage is of restricted type `class`, `model`, or `block`, then it is mapped to a «modelicaPart»,
- if it is of restricted type `connector`, then it is mapped to a «modelicaPort», and
- if it is of restricted type `record` or `type`, then it is mapped to «modelicaValueProperty».

In addition, the stereotype «modelicaFunctionParameter» is introduced to represent components of restricted type `record` or `type` that are used in a `function` (this is necessary because a Modelica function is mapped to a SysML FunctionBehavior that has parameters rather than properties). The restricted types `package` and `function` are not considered here because they cannot be instantiated.

Depending on the type of restricted type, a Modelica Component declaration allows for a variety of options (modifications or additional specifications). These additional options are captured as attributes of the corresponding SysML4Modelica stereotypes, as shown in Figure 10.2. To define the possible values these options can assume, several enumerations are defined, as shown in Figure 10.3. The following production rules define Modelica Components declarations:

```
component_clause:
    type_prefix type_specifier [ array_subscripts ] component_list

type_prefix :
    [ flow ]
    [ discrete | parameter | constant ] [ input | output ]

type_specifier : name

component_list :
    component_declaration { "," component_declaration }

component_declaration :
    declaration [ conditional_attribute ] comment

conditional_attribute: if expression

declaration :
IDENT [ array_subscripts ] [ modification ]
```

---

1. Note that Modelica does not have the equivalent of a reference property — properties are never shared.

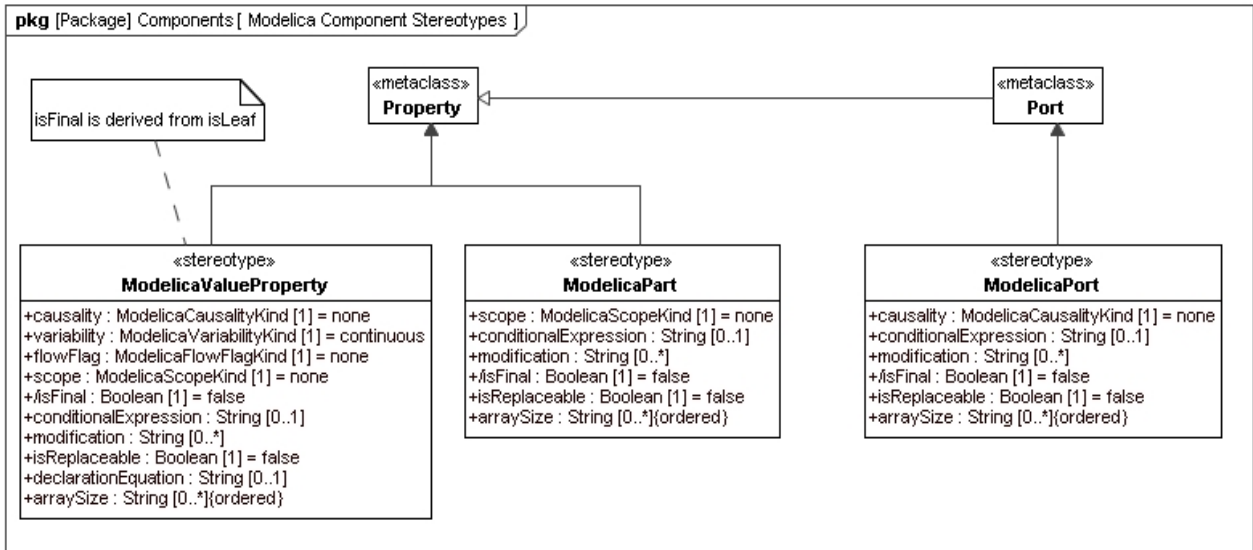


Figure 10.1 - Package diagram with an overview of the stereotypes for Modelica Components

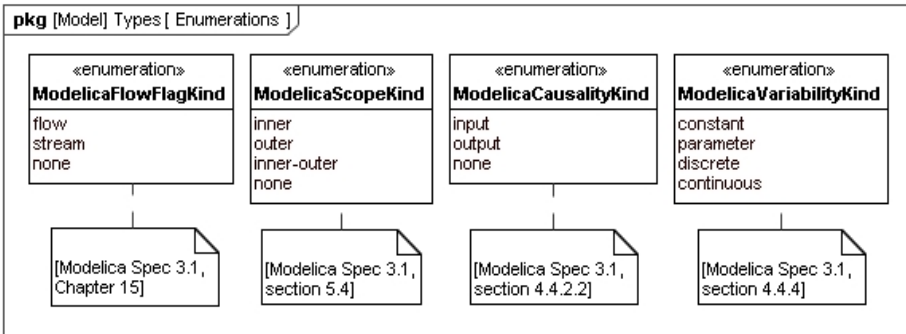


Figure 10.2 - Package diagram with enumerations used in Modelica Component definitions

**Table 10.1 - The applicable attributes for Modelica Components**

Attribute Name	<<modelicaValueProperty>>	<<modelicaPart>>	<<modelicaPort>>
visibility	•	•	
causality	•		•
variability	•		
flowFlag	•		
scope	•	•	
conditionalExpression	•	•	•
isFinal	•	•	•
modification	•	•	•
isReplaceable	•	•	•
declarationEquation	•		
arraySize	•	•	•

## 10.2 «modelicaValueProperty»

### Extensions

- Property (from UML4SysML)

### Abstract Syntax

- See Figure 10.2

### Description

If a Modelica Component is of restricted type *record* or *type*, then it is mapped to a «modelicaValueProperty» that is the equivalent of a Value Property in SysML.

### Attributes

- **visibility:** VisibilityKind [1]  
This attribute is inherited from the meta-class Property. In the context of the SysML4Modelica profile, it is limited to the values *public* or *protected*. A protected «modelicaValueProperty» cannot be modified or replaced in specializations or modifications. The members of a protected «modelicaValueProperty» cannot be accessed using the dot-notation. Default value is *public*.
- **causality:** ModelicaCausalityKind [1]  
A «modelicaValueProperty» can be defined as being an input or output (see Modelica, v3.1, sub clause 4.4.2.2). Default value is *none*, which means that the property is neither an input or output.
- **variability:** ModelicaVariabilityKind [1]  
A «modelicaValueProperty» can be defined as being constant, parameter, discrete, or continuous (see Modelica, v 3.1, sub clauses 4.4.3 and 4.4.4). Default value is *continuous*.

- **flowFlag**: ModelicaFlowFlagKind [1]  
This attribute can only be applied to variables that are a subtype of ModelicaReal. It can only be used inside «modelicaConnector» or to define a Type. The attribute *causality* must be *null* when *flowFlag=flow* or *stream*. Default value is *none*.
- **scope**: ModelicaScopeKind [1]  
A Modelica element declared with the prefix *outer* references an element instance with the same name but using the prefix *inner*, which is nearest in the enclosing instance hierarchy of the outer element declaration (see Modelica, v3.1, sub clause 5.4). Default value is *none*.
- **conditionalExpression**: String [0..1]  
When defined, this attribute contains an expression in Modelica syntax that must evaluate to *true* or *false*. Only if the expression evaluates to *true* is the the corresponding «modelicaValueProperty» instantiated (see Modelica v 3.1, sub clause 4.4.5).
- **smodification**: String [0..\*]  
A «modelicaValueProperty» may have a type that is locally modified. Rather than capturing the detailed semantics of such modifications in the SysML4Modelica profile, currently, the modifications are only captured as a set of strings in the Modelica syntax; each string corresponds to a single modification of a component declaration of the modified class (see Modelica, v 3.1, sub clause 7.2). Default value is *null*.
- **sisReplaceable**: Boolean [1]  
A «modelicaValueProperty» may be defined as *replaceable*. One can then *redeclare* such a «modelicaValueProperty» in extended classes or in modifications (see Modelica, v 3.1, sub clause 7.3). Default value is *false*.
- **s declarationEquation**: String [0.. 1]  
When defined, this attribute contains an expression in Modelica syntax that must evaluate to the same type as the «modelicaValueProperty» itself. A declaration equation refers to the shorthand notation in Modelica in which an equation corresponding to a component is defined in the equation section. The value of the attribute is the right-hand-expression of the equations. The “=” sign is omitted, i.e., it is implicit.
- **s/isFinal**: Boolean [1]  
A Modelica element declared with the prefix *final* cannot be modified in redeclarations or modifications (see Modelica, v 3.1, sub clause 7.2.6). Default value is *false*. This is derived from *isLeaf*.
- **sarraySize**: String [0..\*] {ordered}  
This attribute is an ordered list of strings, each of which must be a Modelica expressions that evaluates to an integer. The  $i^{th}$  element in the ordered list corresponds to size of the multi-dimensional array in the  $i^{th}$  dimension.

#### Associations

No additional associations

#### Constraints

No additional constraints

### 10.3 «modelicaPart»

#### Extensions

- Property (from UML4SysML)

## Abstract Syntax

- See Figure 10.2

## Description

If a Modelica Component is of restricted type `class`, `model`, or `block`, it is mapped to a «modelicaPart», which is the equivalent of a Part Property in SysML.

## Attributes

- `svisibility`: `VisibilityKind` [1]  
This attribute is inherited from the meta-class `Property`. In the context of the SysML4Modelica profile, it is limited to the values *public* or *protected*. A protected «modelicaPart» cannot be modified or replaced in specializations or modifications. The members of a protected «modelicaPart» cannot be accessed using the dot- notation. Default value is *public*.
- `s scope`: `ModelicaScopeKind` [1]  
A Modelica element declared with the prefix `outer` references an element instance with the same name but using the prefix `inner`, which is nearest in the enclosing instance hierarchy of the outer element declaration (see Modelica, v 3.1, sub clause 5.4). Default value is *none*.
- `sconditionalExpression`: `String` [0..1]  
When defined, this attribute contains an expression in Modelica syntax that must evaluate to *true* or *false*. Only if the expression evaluates to *true* is the the corresponding «modelicaPart» instantiated (see Modelica, v 3.1, sub clause 4.4.5). Default value is *null*.
- `smodification`: `String` [0..\*]  
A «modelicaPart» may have a type that is locally modified. Rather than capturing the detailed semantics of such modifications in the SysML4Modelica profile, currently, the modifications are only captured as a set of strings in the Modelica syntax; each string corresponds to a single modification of a component declaration of the modified class (see Modelica, v 3.1, sub clause 7.2). Default value is *null*.
- `sisReplaceable`: `Boolean` [1]  
A «modelicaPart» may be defined as `replaceable`. One can then `redeclare` such a «modelicaPart» in extended classes or in modifications (see Modelica, v 3.1, sub clause 7.3). Default value is *false*.
- `s/isFinal`: `Boolean` [1]  
A Modelica element declared with the prefix `final` cannot be modified in redeclarations or modifications (see Modelica, v 3.1, sub clause 7.2.6). Default value is *false*. This is derived from *isLeaf*.
- `sarraySize`: `String` [0..\*] {ordered}  
This attribute is an ordered list of strings, each of which must be a Modelica expression that evaluates to an integer.
  - The  $i^{th}$  element in the ordered list corresponds to size of the the multi-dimensional array in the  $i^{th}$  dimension. The default value is *null*.

## Associations

No additional associations

## Constraints

No additional constraints

## 10.4 «modelicaPort»

### Extensions

- sPort (from UML4SysML)

### Abstract Syntax

- See Figure 10.2

### Description

If a Modelica Component is of restricted type `connector`, it is mapped to a «modelicaPort», which is the equivalent of a Port Property in SysML.

### Attributes

- s causality: ModelicaCausalityKind [1]  
A «modelicaPort» can be defined as being an input or output (see Modelica, v 3.1, sub clause 4.4.2.2). Default value is *null*, which means that the property is neither an input or output. Default value is *none*.
- s scope: ModelicaScopeKind [1]  
A Modelica element declared with the prefix `outer` references an element instance with the same name but using the prefix `inner`, which is nearest in the enclosing instance hierarchy of the outer element declaration (see Modelica, v 3.1, sub clause 5.4). Default value is *none*.
- sconditionalExpression: String [0..1]  
When defined, this attribute contains an expression in Modelica syntax that must evaluate to *true* or *false*. Only if the expression evaluates to *true* is the the corresponding «modelicaPort» instantiated (see Modelica, v 3.1, sub clause 4.4.5).
- s/isFinal: Boolean [1]  
A Modelica element declared with the prefix `final` cannot be modified in redeclarations or modifications (Modelica, v 3.1, sub clause 7.2.6). Default value is *false*. This is derived from *isLeaf*.
- smodification: String [0..\*]  
A «modelicaPort» may have a type that is locally modified. Rather than capturing the detailed semantics of such modifications in the SysML4Modelica profile, currently, the modifications are only captured as a set of strings in the Modelica syntax; each string corresponds to a single modification of a component declaration of the modified class (see Modelica, v 3.1, sub clause 7.2).
- sisReplaceable: Boolean [1]  
A «modelicaPort» may be defined as `replaceable`. One can then `redeclare` such a «modelicaPort» in extended classes or in modifications (see Modelica, v 3.1, sub clause 7.3). Default value is *false*.
- sarraySize: String [0..\*] {ordered}  
This attribute is an ordered list of strings, each of which must be a Modelica expressions that evaluates to an integer. The  $i^{th}$  element in the ordered list corresponds to size of the multi-dimensional array in the  $i^{th}$  dimension.

### Associations

No additional associations

### Constraints

No additional constraints



## 10.5 «modelicaFunctionParameter»

### Extensions

- sParameter (from UML4SysML)

### Abstract Syntax

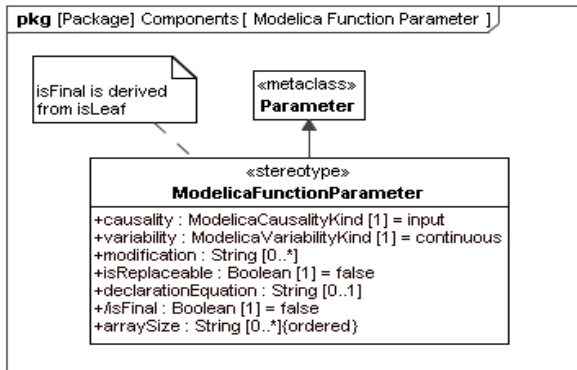


Figure 10.3 - Definition of the «modelicaFunctionParameter» stereotype

### Description

A Modelica restricted class function, can also contain can contain Modelica component declarations. These declarations must be of either restricted type «modelicaType» or «modelicaRecord». Because «modelicaFunction» does not derive from «block» (as all the other restricted classes do), the stereotype «modelicaValueProperty» cannot be applied here. Instead, an equivalent (but more restricted) stereotype for functions is created: «modelicaFunctionParameter».

### Attributes

- scausality: ModelicaCausalityKind [1]  
A «modelicaFunctionParameter» can be defined as being an input or output (see Modelica, v 3.1, sub clause 4.4.2.2). Default value is *input*.
- s/isFinal: Boolean [1]  
A Modelica element declared with the prefix *final* cannot be modified in redeclarations or modifications (see Modelica, v 3.1, sub clause 7.2.6). Default value is *false*. This is derived from *isLeaf*.
- smodification: String [0..\*]  
A «modelicaFunctionParameter» may have a type that is locally modified. Rather than capturing the detailed semantics of such modifications in the SysML4Modelica profile, currently, the modifications are only captured as a set of strings in the Modelica syntax; each string corresponds to a single modification of a component declaration of the modified class (see Modelica, v 3.1, sub clause 7.2).
- sisReplaceable: Boolean [1]  
A «modelicaFunctionParameter» may be defined as *replaceable*. One can then *redeclare* such a «modelicaPort» in extended classes or in modifications (see Modelica, v 3.1, sub clause 7.3). Default value is *false*.
- s declarationEquation: String [0.. 1]  
When defined, this attribute contains an expression in Modelica syntax that must evaluate to the same type as the «modelicaFunctionParameter» itself. A declaration equation refers to the shorthand notation in Modelica in which an

equation corresponding to a component is defined in the equation section. The value of the attribute is the right-hand-expression of the equations. The “:=” sign is omitted, i.e., it is implicit.

- `sarraySize`: String [0..\*] {ordered}  
This attribute is an ordered list of strings, each of which must be a Modelica expressions that evaluates to an integer.
  - The  $i^{th}$  element in the ordered list corresponds to size of the the multi-dimensional array in the  $i^{th}$  dimension.

#### **Associations**

No additional associations

#### **Constraints**

No additional constraints

# 11 Equation and Algorithm Sections

## 11.1 Overview

Equations and Algorithms are the main Modelica constructs for defining behavior of Modelica classes. Modelica distinguishes between declarative equations, which are organized in equation sections (see Modelica, v 3.1, Clause 8), and imperative algorithms, which are organized in algorithm sections (see Modelica, v 3.1, Clause 11). The Modelica restricted classes, class, model, and block can each have zero or more equation and algorithm sections. Modelica functions can only have one single algorithm sections (and no equations).

The equations and expressions in equation and algorithm sections are enforced by the solver in every time step --- they must hold at every moment in time. In addition, one can specify equations or expressions that only need do hold at the start of the simulation; they are organized in initial equation and initial algorithm sections.

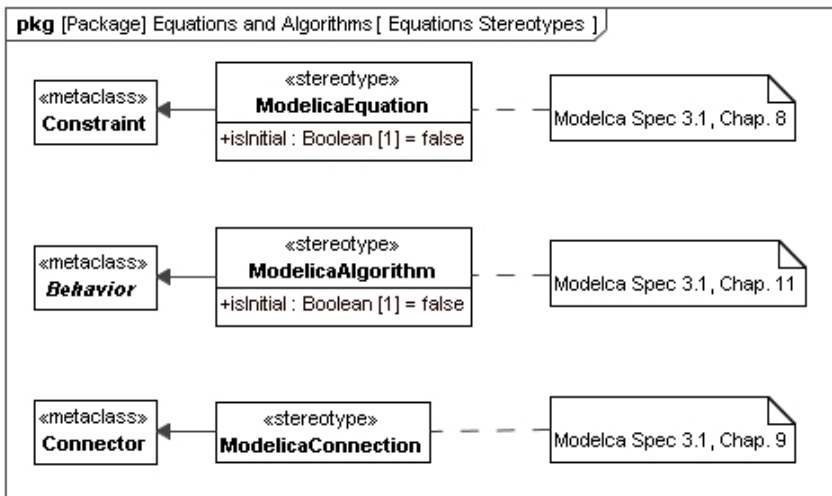


Figure 11.1 - Package diagram with Equation and Algorithm definitions

## 11.2 «modelicaEquation»

### Extensions

- Constraint (from UML4SysML)

### Abstract Syntax

- See Figure 11.1

## Description

Modelica `equation` sections contain declarative equations that must hold at every moment in time. Each model (of restricted class types `class`, `model`, or `block`) may contain zero or more equation sections. Given that the equations in these equation sections are declarative, they could be combined into a single section (note: the order in which declarative equations are defined does not matter). However, the SysML4Modelica mapping allows for each equation section to be modeled by a separate «modelicaEquation».

Modelica `equation` sections may also contain `connect` statements (see Modelica, v 3.1, Clause 9). Although `connect` statements are treated just like other equations in Modelica, they require special attention in SysML4Modelica. Refer to sub clause 31 from details on «modelicaConnection».

## Attributes

- `isInitial`: Boolean [1]  
This attribute is *true* when the «modelicaEquation» represents an *initial equation* sub clause in Modelica. The default value is *false*.

## Associations

No additional associations

## Constraints

No additional constraints

## 11.3 «modelicaAlgorithm»

### Extensions

- Behavior (from UML4SysML)

### Abstract Syntax

- See Figure 11.1

## Description

Modelica `algorithm` sections contain imperative statements that are executed at every moment in time. Each model (of restricted class types `class`, `model`, or `block`) may contain zero or more algorithm sections. In addition, a `function` contains at most one algorithm section. Each algorithm section is modeled by a separate «modelicaAlgorithm». To capture the imperative nature of algorithm sections, a «modelicaAlgorithm» extends UML4SysML: `Behavior`. Only opaque behaviors are currently supported and the algorithm statements are expressed in Modelica syntax in the *Body* of the «modelicaAlgorithm».

## Attributes

- `isInitial`: Boolean [1]  
This attribute is *true* when the «modelicaAlgorithm» represents an *initial algorithm* sub clause in Modelica. The default value is *false*.

## Associations

No additional associations

## Constraints

No additional constraints

## 11.4 «modelicaConnection»

### Extensions

- Connector (from UML4SysML)

### Abstract Syntax

- See Figure 11.1

### Description

In Modelica, a `connection` between two ports typically has Kirchhoff semantics (i.e., across variables are equal, through variables sum to zero), or an output-to-input binding in the case of a signal connection (See Modelica Spec. 3.1, Clause 9). To capture these same semantics succinctly, a «modelicaConnection» is used. The two arguments of the connect statement correspond to the two ends of the «modelicaConnection». Note that the use of a «modelicaConnection» is optional. The alternative is to represent the connection using a connect statement in Modelica syntax in a «modelicaEquation». If a «modelicaConnection» is used, then the corresponding connect statement must be removed from the «modelicaEquation».

As for all equations, Modelica allows `connect` statements to be used in a parametric fashion, for instance, inside a for loop. Since the parameter values are only resolved at the time of compilation of the Modelica model, a parametrically defined `connect` statement cannot be modeled explicitly in SysML4Modelica. The alternative is to represent such connect statements in Modelica syntax in a «modelicaEquation».

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

[1]The start and end of a «modelicaConnection» must be a «modelicaPort».



# 12 Other Related Constructs

## 12.1 «modelicaSimulation»

### Generalizations

- Block (from SysML)

### Abstract Syntax

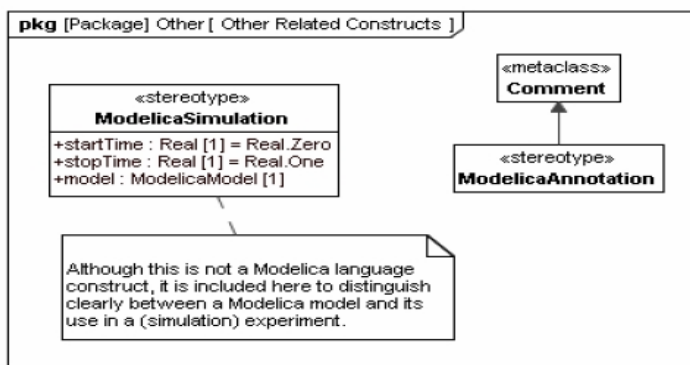


Figure 12.1 - Package diagram with definitions of Modelica-related constructs

### Description

A “modelicaSimulation” is not a Modelica language construct. However, it is introduced in order to distinguish between the model and its simulation. A simulation refers to the solution of the initial value problem: the integration of the model over a particular time period starting from a particular initial condition. Since the initial conditions are already defined in the model itself, the only additional information that needs to be provided is the time over which to integrate and the properties of the solver to be used.

### Attributes

- startTime: Real [1]  
The time at which the simulation starts. Default value is 0.
- stopTime: Real [1]  
The time at which the simulation stops. Default value is 1.
- model: "modelicaClassDefinition" [1]  
The instance of a specialization of “modelicaClassDefinition” that is to be solved. Default value is null.

### Associations

No additional associations

### Constraints

No additional constraints

## 12.2 "modelicaAnnotation"

### Extension

- Comment (from UML4SysML)

### Abstract Syntax

- See Figure 12.1

### Description

Any Modelica language construct can be annotated with information about its graphical representation. In addition, guidelines for the compiler can be specified. In SysML4Modelica, these annotations are represented in Modelica syntax as "modelicaAnnotation."

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints



# Part III - Modelica Abstract Syntax

## Introduction

Part III of the SysML-Modelica Transformation Specification is non-normative and contains the following Clauses:

- Clause 13 - Modelica Meta-Modeling Approach
- Clause 14 - Modelica Meta-Model Constructs



# 13 Modelica Meta-Modeling Approach

## 13.1 General

The abstract syntax (AST = abstract syntax tree) of Modelica is not standardized by the Modelica Association, only the textual syntax is. The abstract syntax described in this document is therefore only one possible definition, defined in an extended subset of Modelica (also known as MetaModelica<sup>1</sup>) and used in the OpenModelica specification/implementation of Modelica which originated as a Structural Operational Semantics/Natural Semantics specification (first version from 1998).

The abstract syntax used in OpenModelica has been designed with several goals in mind:

- Complete representation of all Modelica language constructs.
- Reconstruction of the source code from the AST.
- Use for semantic specification, type checking, and compilation.

Syntax type classes are defined using the uniontype construct. A union type is the union of all the record types it contains. Recursive references to a union type are allowed. Components with optional values are declared at instances of the Option<...> parametrized type constructor. In a few cases the tuple<type1 ,type2,...> type constructor is used. A tuple type can be described as an anonymous record type, where the record type name and the field names are not defined.

In the following all MetaModelica classes (including a short textual description) are listed (version Oct.2009<sup>2</sup> from the OpenModelica SVN). This definition is translated into an OMG MOF-based description (see <http://www.omg.org/mof/>) using the Eclipse EMF (<http://www.eclipse.org/emf/>) implementation of a subset of the OMG MOF standard. Please see the .ecore file for details.

The mapping between MetaModelica and EMOF is defined as follows:

- MetaModelica package is translated to Package.
- MetaModelica uniontype is translated to Class (isAbstract).
- MetaModelica record is translated to Class which inherits from the respective Class that represents the uniontype) - MetaModelica record attributes of primitive type are translated to Class attributes of primitive type -MetaModelica record attributes of composite type are translated to Class Property to the respective Class -MetaModelica types are expanded and translated into Classes.
- MetaModelica tuples are expanded and translated into Classes with the prefix “tuple\_”.
- MetaModelica Option<...> implies the multiplicity 0.. 1.
- MetaModelica list<...> implies the multiplicity 0..\*.
- MetaModelica type I dent = String; is not translated. String is used directly.
- In order to avoid name clashes between Classes representing uniontype or record each Class that represents a union-type was given the prefix “u.”

---

1. MetaModelica corresponds to OMG MOF

2. Note that all MetaModelica-specific classes that are not used for the definition of Modelica language are removed.



“uClassDef.” Figure 13.2 shows the PROGRAM meta-class having a list of class definitions declared at the top level in the source file, combined with a within statement that indicates the hierarchical position of the program. Figure 13.3 shows the COMPONENTS meta-class and its related meta-classes including the COMPONENT meta-class.

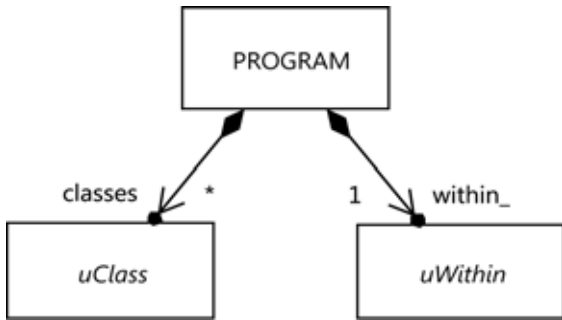


Figure 13.2 - PROGRAM meta-class and relationships

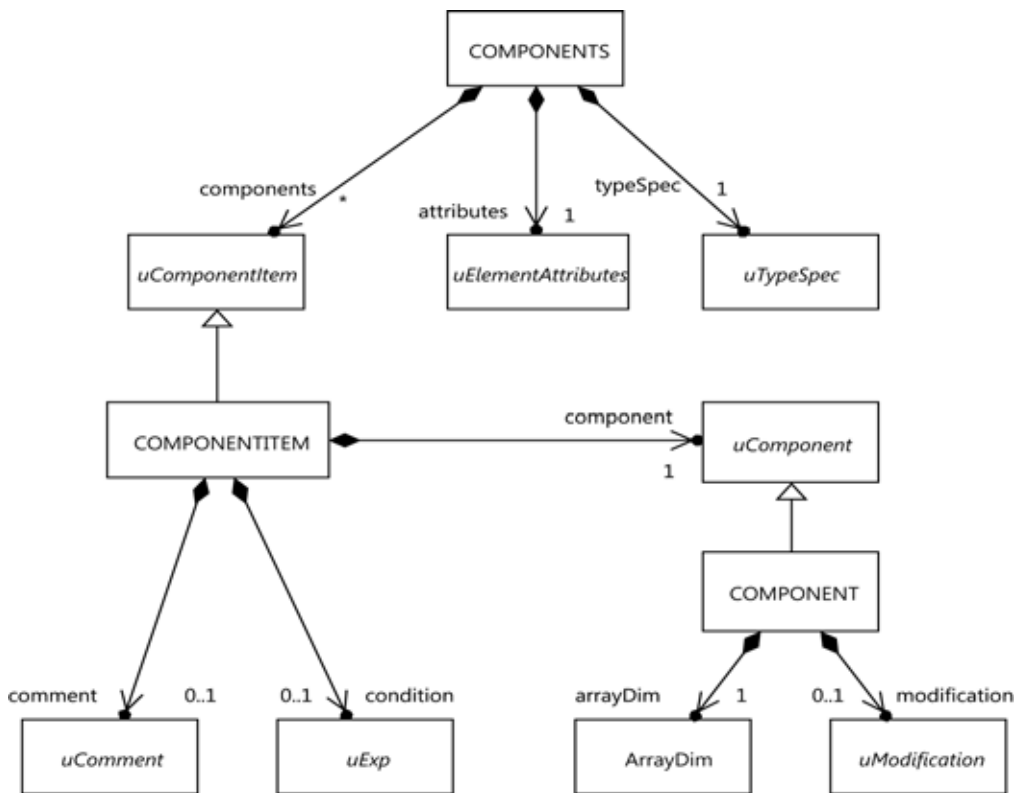


Figure 13.3 - COMPONENTS meta-class and relationships



# 14 Modelica Meta-Model Constructs

## 14.1 The Model Structure Definition

### 14.1.1 Program

```
public uniontype Program
```

“A program is a list of class definitions declared at top level in the source file, combined with a within statement that indicates the hierarchical position of the program.”

```
    record PROGRAM "PROGRAM, the top level construct"  
        list<Class> classes "List of classes" ;  
        Within      within_ "Within clause" ;  
    end PROGRAM;  
end Program;
```

### 14.1.2 Within

```
public uniontype Within "Within Clauses"
```

//See Modelica specification 3.1 sub clause 13.2.2.3 The within Clause.

```
    record WITHIN "the within clause"  
        Path path "the path for within";  
    end WITHIN;  
    record TOP end TOP;  
end Within;
```

### 14.1.3 Path

```
uniontype Path
```

“Path’ is used to store references to class names, or names inside class definitions.”

```
    record QUALIFIED  
        Ident name "name" ;  
        Path path "path" ;  
    end QUALIFIED;  
  
    record IDENT  
        Ident name "name" ;  
    end IDENT;  
end Path;
```

## 14.2 Class Definition

### 14.2.1 Class

```
public uniontype Class
```

“A class definition consists of a name, a flag to indicate if this class is declared as partial, the declared class restriction, and the body of the declaration.”

See Modelica specification 3.1 sub clause 4.5 Class Declarations.

```
record CLASS
  Ident name;
  Boolean   partialPrefix "true if partial" ;
  Boolean   finalPrefix "true if final" ;
  Boolean   encapsulatedPrefix "true if encapsulated" ;
  Restriction restriction "Restriction" ;
  ClassDef  body;
end CLASS;
end Class;
```

## 14.2.2 Restriction

**uniontype Restriction**

“These constructors each correspond to a different kind of class declaration in Modelica, except the last four, which are used for the predefined types.” See Modelica specification 3.1 sub clause 4.6 Specialized Classes.

```
record R_CLASS end R_CLASS;
record R_MODEL end R_MODEL;
record R_RECORD end R_RECORD;
record R_BLOCK end R_BLOCK;
record R_CONNECTOR "connector class" end R_CONNECTOR;
record R_EXP_CONNECTOR "expandable connector class" end R_EXP_CONNECTOR;
record R_TYPE end R_TYPE;
record R_PACKAGE end R_PACKAGE;
record R_FUNCTION end R_FUNCTION;
record R_ENUMERATION end R_ENUMERATION;
record R_PREDEFINED_INT end R_PREDEFINED_INT;
record R_PREDEFINED_REAL end R_PREDEFINED_REAL;
record R_PREDEFINED_STRING end R_PREDEFINED_STRING;
record R_PREDEFINED_BOOL end R_PREDEFINED_BOOL;
record R_PREDEFINED_ENUM end R_PREDEFINED_ENUM;
end Restriction;
```

## 14.2.3 ClassDef

**public uniontype ClassDef**

“The ClassDef type contains the definition part of a class declaration. The definition is either explicit, with a list of parts (public, protected, equation, and algorithm), or it is a definition derived from another class or an enumeration type. For a derived type, the type contains the name of the derived class and an optional array dimension and a list of modifications.”

See Modelica specification 3.1 sub clause 4.5 Class Declarations.

```
record PARTS
  list<ClassPart> classParts;
  Option<String> comment;
end PARTS;
```



```
record DERIVED
```

See Modelica specification 3.1 sub clause 4.5.1 Short Class Definitions.

```
    TypeSpec    typeSpec "typeSpec specification includes array dimensions";  
    ElementAttributes attributes;  
    list<ElementArg> arguments;  
    Option<Comment> comment;  
end DERIVED;
```

```
record ENUMERATION
```

See Modelica specification 3.1 sub clause 4.8.5 Enumeration Types.

```
    EnumDef          enumLiterals;  
    Option<Comment> comment;  
end ENUMERATION;
```

```
record OVERLOAD
```

See Modelica specification 3.1 Clause 14 Overloaded Operators.

```
    list<Path>        functionNames;  
    Option<Comment> comment;  
end OVERLOAD;
```

```
record CLASS_EXTENDS
```

See Modelica specification 3.1 sub clause 7.1 Inheritance -Extends Clause.

```
    Ident          baseClassName "name of class to extend" ;  
    list<ElementArg> modifications "modifications to be applied to the base  
class";  
    Option<String> comment        "comment";  
    list<ClassPart> parts         "class parts";  
end CLASS_EXTENDS;
```

```
record PDER
```

See Modelica specification 3.1 sub clause 4.5 Class Declarations.

```
    Path          functionName;  
    list<Ident> vars "derived variables" ;  
end PDER;  
end ClassDef;
```

## 14.2.4 TypeSpec

```
public uniontype TypeSpec
```

```
    record TPATH  
        Path path;  
        Option<ArrayDim> arrayDim;  
    end TPATH;
```

```
    record TCOMPLEX  
        Path          path;  
        list<TypeSpec> typeSpecs;  
        Option<ArrayDim> arrayDim;
```

```
    end TCOMPLEX;
end TypeSpec;
```

### 14.2.5 EnumDef

```
public uniontype EnumDef
```

“The definition of an enumeration is either a list of literals or a colon, \':\' which defines a supertype of all enumerations.” See Modelica specification 3.1 sub clause 4.8.5 Enumeration Types.

```
    record ENUMLITERALS
        list<EnumLiteral> enumLiterals;
    end ENUMLITERALS;

    record ENUM_COLON end ENUM_COLON;
end EnumDef;
```

### 14.2.6 EnumLiteral

```
public uniontype EnumLiteral
```

“EnumLiteral, which is a name in an enumeration and an optional Comment.” See Modelica specification 3.1 sub clause 4.8.5 Enumeration Types.

```
    record ENUMLITERAL
        Ident          literal;
        Option<Comment> comment;
    end ENUMLITERAL;
end EnumLiteral;
```

### 14.2.7 ClassPart

```
public uniontype ClassPart
```

“A class definition contains several parts. There are public and protected component declarations, type definitions and ‘extends’ clauses, collectively called elements. There are also equation sections and algorithm sections. The EXTERNAL part is used only by functions which can be declared as external C or FORTRAN functions.”

```
    record PUBLIC
```

See Modelica specification 3.1 sub clause 4.1 Access Control - Public and Protected Elements.

```
        list<ElementItem> contents;
    end PUBLIC;
```

```
    record PROTECTED
```

See Modelica specification 3.1 sub clause 4.1 Access Control - Public and Protected Elements.

```
        list<ElementItem> contents;
    end PROTECTED;
```

```
    record EQUATIONS
```

See Modelica specification 3.1 Clause 8 Equations.

```
        list<EquationItem> contents;
    end EQUATIONS;
```

```
record INITIALEQUATIONS
```

See Modelica specification 3.1 sub clause 8.6 Initialization, initial equation, and initial algorithm.

```
list<EquationItem> contents;  
end INITIALEQUATIONS;
```

```
record ALGORITHMS
```

See Modelica specification 3.1 Clause 11 Statements and Algorithms.

```
list<AlgorithmItem> contents;  
end ALGORITHMS;
```

```
record INITIALALGORITHMS
```

See Modelica specification 3.1 sub clause 8.6 Initialization, initial equation, and initial algorithm.

```
list<AlgorithmItem> contents;  
end INITIALALGORITHMS;
```

```
record EXTERNAL
```

See Modelica specification 3.1 sub clause 12.9 External Function Interface.

```
ExternalDecl externalDecl "externalDecl" ;  
Option<Annotation> annotation_ "annotation" ;  
end EXTERNAL;
```

```
end ClassPart;
```

## 14.2.8 ExternalDecl

```
public uniontype ExternalDecl
```

“Declaration of an external function call.”

See Modelica specification 3.1 sub clause 12.9 External Function Interface.

```
record EXTERNALDECL  
  Option<Ident>      funcName "The name of the external function" ;  
  Option<String>    lang      "Language of the external function" ;  
  Option<ComponentRef> output_ "output parameter as return value" ;  
  list<Exp>         args      "only positional arguments, i.e. expression  
list" ;  
  Option<Annotation> annotation_ ;  
end EXTERNALDECL;  
end ExternalDecl;
```

## 14.2.9 ElementItem

```
public uniontype ElementItem
```

“An element item is either an element or an annotation.”

```
record ELEMENTITEM  
  Element element;  
end ELEMENTITEM;
```

```

    record ANNOTATIONITEM
        Annotation annotation_ ;
    end ANNOTATIONITEM;
end ElementItem;

```

## 14.2.10 Element

```
public uniontype Element
```

“Elements: The basic element type in Modelica”

```

    record ELEMENT
        Boolean                finalPre fix;
        Option<RedeclareKeywords> redeclareKeywords "replaceable, redeclare" ;
        InnerOuter             innerOuter "inner/outer" ;
        Ident                  name;
        ElementSpec            specification "Actual element specification" ;
        Option<ConstrainClass> constrainClass "constrainClass ; only valid for
classdef and component" ;
    end ELEMENT;

    record DEFINEUNIT
        Ident name;
        list<NamedArg> args;
    end DEFINEUNIT;

    record TEXT
        Option<Ident> optName "optName : optional name of text, e.g. model with
syntax error. We need the name to be able to browse it. . ." ;
        String string;
        Info info;
    end TEXT;
end Element;

```

## 14.2.11 InnerOuter

```
public uniontype InnerOuter
```

See Modelica specification 3.1 sub clause 5.4 “Instance Hierarchy Name Lookup of Inner Declarations” for explanations of inner/outer.

```

    record INNER end INNER; record
        OUTER end OUTER; record
        INNEROUTER end INNEROUTER;

    record UNSPECIFIED end UNSPECIFIED;
end InnerOuter;

```

## 14.2.12 ComponentRef

```
uniontype ComponentRef
```

“A component reference is the fully or partially qualified name of a component.”

```
    record CREF_QUAL
```

```

    Ident name "name" ;
    list<Subscript> subScripts "subScripts" ;
    ComponentRef componentRef "componentRef" ;
end CREF_QUAL;

record CREF IDENT
    Ident name "name" ;
    list<Subscript> subscripts "subscripts" ;
end CREF IDENT;
end ComponentRef;

```

### 14.2.13 Subscript

uniontype Subscript

“The Subscript uniontype is used both in array declarations and component references. The NOSUB constructor means that the dimension size is undefined when used in a declaration, and when it is used in a component reference it means a slice of the whole dimension.”

See Modelica specification 3.1 sub clause 10.5 Array Indexing.

```

record NOSUB end NOSUB;

record SUBSCRIPT
    Exp subScript "subScript" ;
end SUBSCRIPT;
end Subscript;

```

### 14.2.14 ConstrainClass

public uniontype ConstrainClass

See Modelica specification 3.1 sub clause 7.3.2 Constraining Type.

```

record CONSTRAINCLASS
    ElementSpec elementSpec "elementSpec ; must be extends" ;
    Option<Comment> comment "comment" ;
end CONSTRAINCLASS;
end ConstrainClass;

```

### 14.2.15 ElementSpec

public uniontype ElementSpec

“An element is something that occurs in a public or protected section in a class definition. There is one constructor in the ‘ElementSpec’ type for each possible element type. There are class definitions (‘CLASSDEF’), ‘extends’ clauses (‘EXTENDS’) and component declarations (‘COMPONENTS’). As an example, if the element ‘extends TwoPin;’ appears in the source, it is represented in the AST as ‘EXTENDS(IDENT(“TwoPin”),{ })’.”

```

record CLASSDEF
    Boolean replaceable_ "replaceable" ;
    Class class_ "class" ;
end CLASSDEF;

record EXTENDS

```

See Modelica specification 3.1 sub clause 7.1 Inheritance - Extends Clause.

```
    Path path "path" ;
    list<ElementArg> elementArg "elementArg" ;
    Option<Annotation> annotationOpt "optional annotation";
end EXTENDS;

record IMPORT
```

See Modelica specification 3.1 sub clause 13.2.1 Importing Definitions from a Package.

```
    Import import_ "import" ;
    Option<Comment> comment "comment" ;
end IMPORT;

record COMPONENTS
    ElementAttributes attributes "attributes" ;
    TypeSpec typeSpec "typeSpec" ;
    list<ComponentItem> components "components" ;
end COMPONENTS;
end ElementSpec;
```

## 14.3 Import

```
public uniontype Import
```

See Modelica specification 3.1 sub clause 13.2.1 Importing Definitions from a Package.

```
    record NAMED_IMPORT
        Ident name "name" ;
        Path path "path" ;
    end NAMED_IMPORT;

    record QUAL_IMPORT
        Path path "path" ;
    end QUAL_IMPORT;

    record UNQUAL_IMPORT
        Path path "path" ;
    end UNQUAL_IMPORT;
end Import;
```

## 14.4 Annotation and Comments

### 14.4.1 Annotation

```
public uniontype Annotation
```

See Modelica specification 3.1 Clause 17 Annotations.

```
    record ANNOTATION
        list<ElementArg> elementArgs "elementArgs" ;
    end ANNOTATION;
end Annotation;
```

## 14.4.2 Comment

```
public uniontype Comment
```

See Modelica specification 3.1 sub clause 2.2 Comments.

```
    record COMMENT
      Option<Annotation> annotation_ "annotation" ;
      Option<String> comment "comment" ;
    end COMMENT;
end Comment;
```

## 14.5 Component Definition

### 14.5.1 ComponentItem

```
public uniontype ComponentItem
```

See Modelica specification 3.1 sub clause 4.4.1 Syntax and Examples of Component Declarations.

```
    record COMPONENTITEM
      Component component "component" ;
      Option<ComponentCondition> condition "condition" ;
      Option<Comment> comment "comment" ;
    end COMPONENTITEM;
end ComponentItem;
```

### 14.5.2 ComponentCondition

```
public type ComponentCondition = Exp
```

“A componentItem can have a condition that must be fulfilled if the component should be instantiated.”

### 14.5.3 Component

```
public uniontype Component
```

“Some kind of Modelica entity (object or variable)”

```
    record COMPONENT
      Ident name "name" ;
      ArrayDim arrayDim "arrayDim ; Array dimensions, if any" ;
      Option<Modification> modification "modification ; Optional modification" ;
    end COMPONENT;
end Component;
```

### 14.5.4 ElementAttributes

```
public uniontype ElementAttributes
```

See Modelica specification 3.1 sub clause 4.4.1 Syntax and Examples of Component Declarations.

```
    record ATTR
      Boolean flowPrefix "flow" ;
      Boolean streamPrefix "stream" ;
    end ATTR;
```

```

    Variability variability "variability ; parameter, constant etc." ;
    Direction direction "direction" ;
    ArrayDim arrayDim "arrayDim" ;
end ATTR;
end ElementAttributes;

```

## 14.5.5 Variability

```
public uniontype Variability
```

See Modelica specification 3.1 sub clause 3.8 Variability of Expressions.

```

    record VAR end VAR;
    record DISCRETE end DISCRETE;
    record PARAM end PARAM;
    record CONST end CONST;
end Variability;

```

## 14.5.6 Direction

```
public uniontype Direction
```

See Modelica specification 3.1 sub clauses 4.4.1 Syntax and Examples of Component Declarations and 4.4.2.2 Prefix Rules.

```

    record INPUT end INPUT;
    record OUTPUT end OUTPUT;
    record BIDIR end BIDIR;
end Direction;

```

## 14.5.7 ArrayDim

```
public type ArrayDim = list<Subscript>
```

“Component attributes are properties of components that are applied by type prefixes. As an example, declaring a component as ‘input Real x;’ will give the attributes ‘ATTR( { } ,false,VAR,INPUT)’. Components in Modelica can be scalar or arrays with one or more dimensions. This type is used to indicate the dimensionality of a component or a type definition. Array dimensions”

See Modelica specification 3.1 sub clause 10.5 Array Indexing.

## 14.6 Modifications and Redeclarations

### 14.6.1 Modification

```
public uniontype Modification
```

“There are two forms of modifications: redeclarations and component modifications.”

See Modelica specification 3.1 sub clause 7.2 Modifications.

```

    record CLASSMOD
        list<ElementArg> elementArgLst;
        Option<Exp> expOption;
    end CLASSMOD;
end Modification;

```



## 14.6.2 ElementArg

```
public uniontype ElementArg
  record MODIFICATION
```

See Modelica specification 3.1 Chapter 7.2 Modifications.

```
    Boolean finalItem "finalItem" ;
    Each each_ "each" ;
    ComponentRef componentRef "componentRef" ;
    Option<Modification> modification "modification" ;
    Option<String> comment "comment" ;
  end MODIFICATION;

  record REDECLARATION
```

See Modelica specification 3.1 Chapter 7.3 Redeclaration.

```
    Boolean finalItem "finalItem" ;
    RedeclareKeywords redeclareKeywords "redeclare or replaceable " ;
    Each each_ "each" ;
    ElementSpec elementSpec "elementSpec" ;

    Option<ConstrainClass> constrainClass "class definition or declaration" ;
  end REDECLARATION;
end ElementArg;
```

## 14.6.3 RedeclareKeywords

```
public uniontype RedeclareKeywords
```

“The keywords redeclare and replaceable can be given in three different combinations, each one by themselves or the both combined.”

See Modelica specification 3.1 sub clause 7.3 Redeclaration.

```
  record REDECLARE end REDECLARE;
  record REPLACEABLE end REPLACEABLE;
  record REDECLARE_REPLACEABLE end REDECLARE_REPLACEABLE;
end RedeclareKeywords;
```

## 14.6.4 Each

```
public uniontype Each
```

“The each keyword can be present in both: MODIFICATIONS and REDECLARATIONS.”

See Modelica specification 3.1 sub clause 7.2.5 Modifiers for Array Elements.

```
  record EACH end EACH;
  record NON_EACH end NON_EACH;
end Each;
```

## 14.7 Behavior

### 14.7.1 EquationItem

```
public uniontype EquationItem
```

“Several component declarations can be grouped together in one ‘ElementSpec’ by writing them on the same line in the source. This type contains the information specific to one component.”

See Modelica specification 3.1 Clause 8 “Equations.”

```
    record EQUATIONITEM
      Equation equation_ "equation" ;
      Option<Comment> comment "comment" ;
    end EQUATIONITEM;

    record EQUATIONITEMANN
      Annotation annotation_ "annotation" ;
    end EQUATIONITEMANN;
end EquationItem;
```

### 14.7.2 AlgorithmItem

```
public uniontype AlgorithmItem
```

“Information for an algorithm item.”

See Modelica specification 3.1 Clause 11 “Statements and Algorithms.”

```
    record ALGORITHMITEM
      Algorithm algorithm_ "algorithm" ;
      Option<Comment> comment "comment" ;
    end ALGORITHMITEM;

    record ALGORITHMITEMANN
      Annotation annotation_ "annotation" ;
    end ALGORITHMITEMANN;
end AlgorithmItem;
```

### 14.7.3 Equation

```
public uniontype Equation
```

“Information on one (kind) of equation, different constructors for different kinds of equations.”

See Modelica specification 3.1 Clause 8 “Equations.”

```
    record EQ_IF
      Exp ifExp "ifExp ; Conditional expression" ;
      list<EquationItem> equationTrueItems "equationTrueItems ; true branch" ;
      list<tuple<Exp, list<EquationItem>>> elseIfBranches "elseIfBranches" ;
      list<EquationItem> equationElseItems "equationElseItems Standard 2-side
eqn" ;
    end EQ_IF;

    record EQ_EQUALS
```

```

    Exp leftSide "leftSide" ;
    Exp rightSide "rightSide Connect stmt" ;
end EQ_EQUALS;

record EQ_CONNECT
    ComponentRef connector1 "connector1" ;
    ComponentRef connector2 "connector2" ;
end EQ_CONNECT;

record EQ_FOR
    ForIterators iterators;
    list<EquationItem> forEquations "forEquations" ;
end EQ_FOR;

record EQ_WHEN_E
    Exp whenExp "whenExp" ;
    list<EquationItem> whenEquations "whenEquations" ;
    list<tuple<Exp, list<EquationItem>>> elseWhenEquations "elseWhenEquations" ;
end EQ_WHEN_E;

record EQ_NORETCALL
    ComponentRef functionName "functionName" ;
    FunctionArgs functionArgs "functionArgs; fcalls without return value" ;
end EQ_NORETCALL;

record EQ_FAILURE
    EquationItem equ;
end EQ_FAILURE;
end Equation;

```

#### 14.7.4 Algorithm

```
public uniontype Algorithm
```

“The Algorithm type describes one algorithm statement in an algorithm section. It does not describe a whole algorithm. The reason this type is named like this is that the name of the grammar rule for algorithm statements is ‘algorithm’.”

See Modelica specification 3.1 Clause 11 “Statements and Algorithms.”

```

    record ALG_ASSIGN
        Exp assignComponent "assignComponent" ;
        Exp value "value" ;
    end ALG_ASSIGN;

    record ALG_IF
        Exp ifExp "ifExp" ;
        list<AlgorithmItem> trueBranch "trueBranch" ;
        list<tuple<Exp, list<AlgorithmItem>>> elseIfAlgorithmBranch
"elseIfAlgorithmBranch" ;
        list<AlgorithmItem> elseBranch "elseBranch" ;
    end ALG_IF;

    record ALG_FOR
        ForIterators iterators;
        list<AlgorithmItem> forBody "forBody" ;

```

```

end ALG_FOR;

record ALG_WHILE
  Exp boolExpr "boolExpr" ;
  list<AlgorithmItem> whileBody "whileBody" ;
end ALG_WHILE;

record ALG_WHEN_A
  Exp boolExpr "boolExpr" ;
  list<AlgorithmItem> whenBody "whenBody" ;
  list<tuple<Exp, list<AlgorithmItem>>> elseWhenAlgorithmBranch
"elseWhenAlgorithmBranch" ;
end ALG_WHEN_A;

record ALG_NORETCALL
  ComponentRef functionCall "functionCall" ;
  FunctionArgs functionArgs "functionArgs; general fcalls without return
value";
end ALG_NORETCALL;

record ALG_RETURN
end ALG_RETURN;

record ALG_BREAK
end ALG_BREAK;
end Algorithm;

```

## 14.8 Expressions

### 14.8.1 Exp

```
public uniontype Exp
```

“The Exp uniontype is the container of a Modelica expression.”

See Modelica specification 3.1 Clause 3 Operators and Expressions.

```

record INTEGER
  Integer value;
end INTEGER;

record REAL
  Real value;
end REAL;

record CREF
  ComponentRef componentRef;
end CREF;

record STRING
  String value;
end STRING;

record BOOL
  Boolean value;

```

```

end BOOL;

record BINARY
“Binary operations, e.g., a*b”
  Exp exp1;
  Operator op;
  Exp exp2;
end BINARY;

record UNARY
“Unary operations, e.g., -(x)”
  Operator op "op" ;
  Exp exp "exp Logical binary operations: and, or" ;
end UNARY;

record LBINARY
  Exp exp1 "exp1" ;
  Operator op "op" ;
  Exp exp2 ;
end LBINARY;

record LUNARY
“Logical unary operations: not”
  Operator op "op" ;
  Exp exp "exp Relations, e.g. a >= 0" ;
end LUNARY;

record RELATION
  Exp exp1 "exp1" ;
  Operator op "op" ;
  Exp exp2 ;
end RELATION;

record IFEXP
  Exp ifExp "ifExp" ;
  Exp trueBranch "trueBranch" ; Exp elseBranch "elseBranch" ;
  list<tuple<Exp, Exp>> elseIfBranch "elseIfBranch Function calls" ;
end IFEXP;

record CALL
  ComponentRef function_ "function" ;
  FunctionArgs functionArgs ;
end CALL;

record PARTEVALFUNCTION "Partially evaluated function"
  ComponentRef function_ "function" ;
  FunctionArgs functionArgs ;
end PARTEVALFUNCTION;

record ARRAY "Array construction using {, }, or array"
  list<Exp> arrayExp ;

```

```

end ARRAY;

record MATRIX "Matrix construction using {, } "
  list<list<Exp>> matrix ;
end MATRIX;

record RANGE "Range expressions, e.g. 1:10 or 1:0.5:10"
  Exp start "start" ;
  Option<Exp> step "step" ;
  Exp stop "stop";
end RANGE;

record TUPLE "Tuples used in function calls returning several values"
  list<Exp> expressions "comma-separated expressions" ;
end TUPLE;

record END "array access operator for last element, e.g. a{end}:=1;"
end END;
end Exp;

```

## 14.8.2 FunctionArgs

```
uniontype FunctionArgs
```

“The FunctionArgs uniontype consists of a list of positional arguments followed by a list of named arguments.”

See Modelica specification 3.1 sub clause 12.4 Function Call.

```

record FUNCTIONARGS
  list<Exp> args "args" ;
  list<NamedArg> argNames "argNames" ;
end FUNCTIONARGS;

record FOR_ITER_FARG
  Exp exp "iterator expression";
  ForIterators iterators;
end FOR_I TER_FARG;

end FunctionArgs;

```

## 14.8.3 ForIterator

```
public type ForIterator = tuple<Ident, Option<Exp>>
```

See Modelica specification 3.1 sub clause 11.2.2 For-statement and sub clauses 8.3.2 For-Equations - Repetitive Equation Structures, 10.4.1 Array Constructor with Iterators.

## 14.8.4 ForIterators

```
public type ForIterators = list<ForIterator>
```

## 14.8.5 NamedArg

```
uniontype NamedArg
```

“The NamedArg uniontype consist of an identifier for the argument and an expression giving the value of the argument.”

```
record NAMEDARG
  Ident argName "argName" ;
  Exp argValue "argValue" ;
end NAMEDARG;
end NamedArg;
```

## 14.8.6 Operator

uniontype Operator

See Modelica specification 3.1 Clause 3 Operators and Expressions.

```
/* arithmetic operators */
record ADD      "addition"          "end ADD;
record SUB      "subtraction"       "end SUB;
record MUL      "multiplication"    "end MUL;
record DIV      "division"          "end DIV;
record POW      "power"             "end POW;
record UPLUS    "unary plus"        "end UPLUS;
record UMINUS   "unary minus"       "end UMINUS;
/* element-wise arithmetic operators */
record ADD_EW   "element-wise addition" "end ADD_EW;
record SUB_EW   "element-wise subtraction" "end SUB_EW;
record MUL_EW   "element-wise multiplication" "end MUL_EW;
record DIV_EW   "element-wise division" "end DIV_EW;
record POW_EW   "element-wise power" "end POW_EW;
record UPLUS_EW "element-wise unary minus" "end UPLUS_EW;
record UMINUS_EW "element-wise unary plus" "end UMINUS_EW;
/* logical operators */
record AND      "logical and"       "end AND;
record OR       "logical or"        "end OR;
record NOT      "logical not"       "end NOT;
/* relational operators */
record LESS     "less than"         "end LESS;
record LESSEQ   "less than or equal" "end LESSEQ;
record GREATER  "greater than"      "end GREATER;
record GREATEREQ "greater than or equal" "end GREATEREQ;
record EQUAL    "relational equal"  "end EQUAL;
record NEQUAL   "relational not equal" "end NEQUAL;
end Operator;
```





# Part IV - Transformation

## General Information

This part of the document defines the mapping between the SysML4Modelica profile defined in Part II and the Modelica abstract syntax defined in Part III. The mapping is in tables relating elements in the SysML4Modelica profile to elements of the Modelica abstract syntax as well as in QVT. The QVT code is included in Annex C; it includes explicit references to each of the mapping rule numbers included in the tables.

Each mapping table may consist of 4 sections:

1. A general statement describing which element in the SysML profile is being mapped to which element of the Modelica abstract syntax.
2. A Required section describing the required conditions necessary to make the transformation valid
3. A Conditional section describing possible links between attributes based on conditional expressions
4. An Attributes section describing the mapping between any additional attributes

Part IV of the SysML-Modelica Transformation specification is non-normative and contains the following Clauses:

- 15 - Class Definitions
- 16 - Predefined Types
- 17 - Component Declarations
- 18 - Equation and Algorithm Sections



# 15 Class Definitions

## 15.1 «modelicaClassDefinition»

SysML4Modelica	Modelica		Attributes
	Abstract Syntax	Concrete Syntax	
15.1.1:Classes::ModelicaClassDefinition	Absyn.Class.Class	N/A	See below
<b>Specializations:</b>			
15.1.2:Classes::ModelicaClass	Absyn.Class.Class	Class	See 8.3
15.1.3:Classes::ModelicaModel	Absyn.Class.Class	Model	See 8.3
15.1.4:Classes::ModelicaRecord	Absyn.Class.Class	Record	See 8.4
15.1.5:Classes::ModelicaBlock	Absyn.Class.Class	Block	See 8.5
15.1.6:Classes::ModelicaConnector	Absyn.Class.Class	Connector	See 8.6
15.1.7:Classes::ModelicaType	Absyn.Class.Class	Type	See 8.7
15.1.8:Classes::ModelicaPackage	Absyn.Class.Class	Package	See 8.8
15.1.9:classes::ModelicaFunction	Absyn.Class.Class	Function	See 8.9

SysML4Modelica		Modelica
15.1.10:Classes::ModelicaClassDefinition	maps to	Absyn.Class.Class
<b>Attributes</b>		
15.1.11: isFinal	always maps to	• finalPrefix
15.1.12: IsModelicaEncapsulated	always maps to	• EncapsulatedPrefix
15.1.13: IsAbstract	always maps to	• PartialPrefix

## 15.2 «modelicaClass»

<b>SysML4Modelica</b>		<b>Modelica</b>
15.2.1:Classes::ModelicaClass	maps to	Absyn.Class.Class
<b>Required</b>		
15.2.2:		• restriction <b>equal to</b> Restriction.R_Class
15.2.3: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.3 «modelicaModel»

<b>SysML4Modelica</b>		<b>Modelica</b>
15.3.1:Classes::ModelicaModel	maps to	Absyn.Class.Class
<b>Required</b>		
15.3.2:		• restriction <b>equal to</b> Restriction.R_Model
15.3.3: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.4 «modelicaRecord»

<b>SysML4Modelica</b>		<b>Modelica</b>
15.4.1:Classes::ModelicaRecord	maps to	Absyn.Class.Class
<b>Required</b>		
15.4.2:		• restriction <b>equal to</b> Restriction.R_Record
15.4.3: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.5 «modelicaBlock»

SysML4Modelica		Modelica
15.5.1:Classes::ModelicaBlock	maps to	Absyn.Class.Class
<b>Required</b>		
15.5.2:		• restriction <b>equal to</b> Restriction.R_Block
15.5.3: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.6 «modelicaConnector»

SysML4Modelica		Modelica
15.6.1:Classes::ModelicaConnector	maps to	Absyn.Class.Class
<b>Conditional:</b>		
15.6.2: IsExpandable equal to false	maps to	• restriction <b>equal to</b> Restriction.R_CONNECTOR
15.6.3: IsExpandable equal to true	maps to	• restriction <b>equal to</b> Restriction.R_EXP_CONNECTOR
15.6.4: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.7 «modelicaType»

SysML4Modelica		Modelica
15.7.1:Classes::ModelicaType	maps to	Absyn.Class.Class
<b>Required</b>		
15.7.2:		• restriction <b>equal to</b> Restriction.R_Type
15.7.3: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.8 «modelicaPackage»

SysML4Modelica		Modelica
15.8.1:Classes::ModelicaPackage	maps to	Absyn.Class.Class
<b>Required</b>		
15.8.2:		• restriction <b>equal to</b> Restriction.R_Package
15.8.3: Attributes same as SysML4Modelica: :Classes::ModelicaClassDefinition		

## 15.9 «modelicaFunction»

SysML4Modelica		Modelica
15.9.1:Classes::ModelicaFunction	maps to	Absyn.Class.Class
<b>Required:</b>		
15.9.2:		<ul style="list-style-type: none"> <li>restriction <b>equal to</b> Restriction.R_Function</li> </ul>
<b>Conditional:</b>		
15.9.3: language = "C" or "FORTRAN"		<ul style="list-style-type: none"> <li><b>Type of</b> Class.body <b>equal to</b> ClassDef.PARTS</li> <li><b>Type of</b> Class.body.classParts <b>equal to</b> ClassPart.EXTERNAL</li> <li>Type of Class.body.classParts.externalDecl equal to ExternalDecl.EXTERNALDECL</li> </ul>
15.9.4: language		<ul style="list-style-type: none"> <li>Class.body.classParts.externalDecl.lang</li> </ul>
15.9.5: name		<ul style="list-style-type: none"> <li>Class.body.classParts.externalDecl.func</li> </ul>
15.9.6: scope	always maps to	<ul style="list-style-type: none"> <li><b>Type of</b> innerOuter</li> </ul>
15.9.7: externalLibrary	parsed to	<ul style="list-style-type: none"> <li>Class.body.classParts.externalDecl.annotation <b>_with</b> Library={externalLibrary...}</li> </ul>
15.9.8: externalInclude	parsed to	<ul style="list-style-type: none"> <li>Class.body.classParts.externalDecl.annotation <b>_with</b> Include={externalInclude..}</li> </ul>

## 15.10 «modelicaExtends»

There are multiple representations for the `extends` clause within the Modelica abstract syntax. Only one of the possible mappings is addressed here, the one that most closely resembles the SysML4Modelica definition.

SysML4Modelica		Modelica
15.10.1:Classes::ModelicaExtends	maps to	ClassDef.CLASS_EXTENDS
<b>Required:</b>		
15.10.2:Specific	maps to	<ul style="list-style-type: none"> <li>Class <b>with property</b> Class.body <b>equal to</b> ClassDef.CLASS_EXTENDS</li> </ul>
15.10.3:General	maps to	<ul style="list-style-type: none"> <li>ClassDef.Class_EXTENDS.baseClassName</li> </ul>
<b>Attributes:</b>		
15.10.4: modification	parsed to	<ul style="list-style-type: none"> <li>ClassDef.CLASS_EXTENDS.modifications</li> </ul>
15.10.5: visibility	parsed to	<ul style="list-style-type: none"> <li>ClassDef.CLASS_EXTENDS.modifications</li> </ul>
15.10.6: arraySize	parsed to	<ul style="list-style-type: none"> <li>ClassDef.CLASS_EXTENDS.modifications</li> </ul>

## 15.11 «modelicaDer»

SysML4Modelica		Modelica
15.11.1:Classes::ModelicaDer	maps to	Absyn.Class.Class
<b>Required:</b>		
15.11.2:		<ul style="list-style-type: none"> <li>• <b>Type of body equal to</b> ClassDef.PDER</li> </ul>
<b>Attributes:</b>		
15.11.3: name	maps to	<ul style="list-style-type: none"> <li>• body.PDER.functionName</li> </ul>
15.11.4: variables	maps to	<ul style="list-style-type: none"> <li>• body.PDER.vars</li> </ul>

## 15.12 «modelicaConstrainedBy»

SysML4Modelica		Modelica
15.12.1:Classes::ModelicaConstrainedBy	maps to	ConstrainClass.CONSTRAINCLASS.
<b>Required:</b>		
15.12.2:		<ul style="list-style-type: none"> <li>• Referenced by ElementArg.REDECLARATION.constrain Class</li> <li>• Type of CONSTRAINCLASS.elementSpec equal to ElementSpec.-CLASSDEF</li> </ul>
<b>Attributes:</b>		
15.12.3: client	always maps to	<ul style="list-style-type: none"> <li>• CONSTRAINCLASS.elementSpec.-class_</li> </ul>
15.12.4: modification	parsed to	<ul style="list-style-type: none"> <li>• ElementArg.REDECLARATION.redeclareKey words</li> </ul>





# 16 Predefined Types

## 16.1 Overview

The following primitive types are available in the Modelica language: Real Type, Integer Type, Boolean Type, String Type, Enumeration Types, StateSelect, ExternalObject, Graphical Annotation Types. These primitive types are defined as predefined types in SysML4Modelica::BasicTypes. Although these types have direct counterparts in SysML, they are defined again to account for the additional attributes associated with them in Modelica.

<b>SysML4Modelica</b>	<b>Modelica</b>
<b>Basic Types</b>	<b>Predefined Type</b>
16.1.1: ModelicaReal	Real
16.1.2: ModelicaInteger	Integer
16.1.3: ModelicaBoolean	Boolean
16.1.4: ModelicaString	String
16.1.5: ModelicaEnumeration	Enumeration
16.1.6: ModelicaStateSelect	StateSelect
16.1.7: ModelicaExternalObject	ExternalObject
16.1.8: ModelicaAnnotation	Annotation



# 17 Component Declarations

## 17.1 Overview

SysML4Modelica	Modelica	Attributes
17.1.1: Component::ModelicaPart	Absyn.Element.Element	See Sub clause 10.3
17.1.2: Component::ModelicaPort	Absyn.Element.Element	See Sub clause 10.4
17.1.3: Component::ModelicaValueProperty	Absyn.Element.Element	See Sub clause 10.2
17.1.4: Component::ModelicaFunctionParameter	Absyn.Element.Element	See Sub clause 10.5

## 17.2 «modelicaPart»

SysML4Modelica		Modelica
17.2.1:Component::ModelicaPart	maps to	Absyn.Element.Element
<b>Required</b>		
17.2.2:		<ul style="list-style-type: none"> <li><b>Type of specification equal to</b> ElementSpec.COMPONENTS</li> </ul>
17.2.3:		<ul style="list-style-type: none"> <li>Absyn.Class.Class <b>referenced by</b> specification.typeSpec has Restriction <b>equal to</b> R_Block or R_Class or R_Model</li> </ul>
17.2.4:		<ul style="list-style-type: none"> <li><b>Type of specification.components equal to</b> ComponentItem.COMPONENTITEM</li> </ul>
17.2.5:		<ul style="list-style-type: none"> <li><b>Type of</b> specification.components.component <b>equal to</b> Component.COMPONENT</li> </ul>
<b>Attributes</b>		
17.2.6:name	always maps to	<ul style="list-style-type: none"> <li>name</li> <li>specification.components.component.name</li> </ul>
17.2.7:scope	always maps to	<ul style="list-style-type: none"> <li><b>Type of</b> innerOuter</li> </ul>
17.2.8:conditionalExpression	always maps to	<ul style="list-style-type: none"> <li>specification.components.condition</li> </ul>
17.2.9:modification	always maps to	<ul style="list-style-type: none"> <li>specification.components.component.modification</li> </ul>
17.2.10: isFinal	always maps to	<ul style="list-style-type: none"> <li>finalPrefix</li> </ul>
17.2.11: isReplaceable	always maps to	<ul style="list-style-type: none"> <li>redeclareKeywords</li> </ul>
17.2.12: arraySize	always maps to	<ul style="list-style-type: none"> <li>specification.components.component.arrayDim</li> </ul>

## 17.3 «modelicaPort»

SysML4Modelica		Modelica
17.3.1:Component::ModelicaPort	maps to	Absyn.Element.Element
<b>Required</b>		
17.3.2:		<ul style="list-style-type: none"> <li><b>Type of specification equal to</b> ElementSpec.COMPONENTS</li> </ul>
17.3.3:		<ul style="list-style-type: none"> <li>Absyn.Class.Class <b>referenced by</b> specification.typeSpec has restriction <b>equal to</b> R_Connector</li> </ul>
17.3.4:		<ul style="list-style-type: none"> <li><b>Type of specification.components equal to</b> ComponentItem.COMPONENTITEM</li> </ul>
17.3.5:		<ul style="list-style-type: none"> <li><b>Type of</b> specification.components.component <b>equal to</b> Component.COMPONENT</li> </ul>
<b>Attributes</b>		
17.3.6:name	always maps to	<ul style="list-style-type: none"> <li>name</li> <li>specification.components.component.name</li> </ul>
17.3.7:causality	always maps to	<ul style="list-style-type: none"> <li><b>Type of</b> specification.attributes.direction</li> </ul>
17.3.8:conditionalExpression	always maps to	<ul style="list-style-type: none"> <li>specification.components.condition</li> </ul>
17.3.9:modification	always maps to	<ul style="list-style-type: none"> <li>specification.components.component.modification</li> </ul>
17.3.10: isFinal	always maps to	<ul style="list-style-type: none"> <li>finalPrefix</li> </ul>
17.3.11: isReplaceable	always maps to	<ul style="list-style-type: none"> <li>redeclareKeywords</li> </ul>
17.3.12: arraySize	always maps to	<ul style="list-style-type: none"> <li>specification.attributes.arrayDim</li> </ul>

## 17.4 «modelicaValueProperty»

SysML4Modelica		Modelica
17.4.1:Component::ModelicaValueProperty	maps to	Absyn.Element.Element
<b>Required</b>		
17.4.2:		<ul style="list-style-type: none"> <li><b>Type of specification equal to</b> ElementSpec.COMPONENTS</li> </ul>
17.4.3:		<ul style="list-style-type: none"> <li>Absyn.Class.Class <b>referenced by</b> specification.typeSpec has restriction <b>equal to</b> R_Type or R_Record</li> </ul>
17.3.4:		<ul style="list-style-type: none"> <li><b>Type of</b> specification.components <b>equal to</b> ComponentItem.COMPONENTITEM</li> </ul>

SysML4Modelica		Modelica
17.4.5:		<ul style="list-style-type: none"> <li><b>Type of</b> specification.components.component <b>equal to</b> Component.COMPONENT</li> </ul>
17.4.6:		<ul style="list-style-type: none"> <li><b>Type of</b> specification.attributes <b>equal to</b> ElementAttributes.ATTR</li> </ul>
<b>Attributes</b>		
17.4.7:name	always maps to	<ul style="list-style-type: none"> <li>name</li> <li>specification.components.component.name</li> </ul>
17.4.8:visibility		<ul style="list-style-type: none"> <li><b>Type of ClassPart of owning</b> Absyn.Class.Class</li> </ul>
17.4.9:causality	always maps to	<ul style="list-style-type: none"> <li><b>Type of</b> specification.components.component.attributes.direction</li> </ul>
17.4.10:variability	always maps to	<ul style="list-style-type: none"> <li><b>Type of</b> specification.components.component.attributes.variability</li> </ul>
17.4.11: flowFlag	always maps to	<ul style="list-style-type: none"> <li>specification.components.component.attributes.flowPrefix</li> </ul>
17.4.12: scope	always maps to	<ul style="list-style-type: none"> <li><b>Type of</b> innerOuter</li> </ul>
17.4.13: conditionalExpression	always maps to	<ul style="list-style-type: none"> <li>specification.components.condition</li> </ul>
17.4.14: modification	always maps to	<ul style="list-style-type: none"> <li>specification.components.component.modification</li> </ul>
17.4.15: isReplaceable	always maps to	<ul style="list-style-type: none"> <li>redeclareKeywords</li> </ul>
17.4.16: declarationEquation	always maps to	<ul style="list-style-type: none"> <li>redeclareKeywords</li> </ul>
17.4.17: isFinal	always maps to	<ul style="list-style-type: none"> <li>finalPrefix</li> </ul>
17.4.19: arraySize	always maps to	<ul style="list-style-type: none"> <li>specification.components.component.arrayDim</li> </ul>

## 17.5 «modelicaFunctionParameter»

SysML4Modelica		Modelica
17.5.1:Component::ModelicaFunctionParameter	maps to	Absyn.Element.Element
<b>Required</b>		
17.5.2:		<ul style="list-style-type: none"> <li><b>Type of</b> specification <b>equal to</b> ElementSpec.COMPONENTS</li> </ul>
17.5.3:		<ul style="list-style-type: none"> <li>Absyn.Class.Class <b>referenced by</b> specification.type. Spec has restriction <b>equal to</b> R_Type or R_Record.</li> </ul>
17.5.4:		<ul style="list-style-type: none"> <li><b>Component of</b> Absyn.Class.Class <b>with</b> restriction <b>equal to</b> R_Function.</li> </ul>

<b>SysML4Modelica</b>		<b>Modelica</b>
17.5.5:		<ul style="list-style-type: none"> <li>• <b>Type of</b> specification.components <b>equal to</b> ComponentItem.COMPONENTITEM</li> </ul>
17.5.6:		<ul style="list-style-type: none"> <li>• <b>Type of</b> specification.components.component <b>equal to</b> Component.COMPONENT</li> </ul>
17.5.7:		<ul style="list-style-type: none"> <li>• <b>Type of</b> specification.attributes <b>equal to</b> ElementAttributes.ATTR</li> </ul>
<b>Attributes</b>		
17.5.8:name	always maps to	<ul style="list-style-type: none"> <li>• name</li> <li>• specification.components.component.name</li> </ul>
17.5.9:causality	always maps to	<ul style="list-style-type: none"> <li>• <b>Type of</b> specification.attributes.direction</li> </ul>
17.5.10:variability	always maps to	<ul style="list-style-type: none"> <li>• <b>Type of</b> specification.attributes.variability</li> </ul>
17.5.11:isFinal	always maps to	<ul style="list-style-type: none"> <li>• finalPrefix</li> </ul>
17.5.12: modification	always maps to	<ul style="list-style-type: none"> <li>• specification.components.component.modification</li> </ul>
17.5.13: isReplaceable	parsed to	<ul style="list-style-type: none"> <li>• redeclareKeywords</li> </ul>
17.5.14: declarationEquation	parsed to	<ul style="list-style-type: none"> <li>• redeclareKeywords</li> </ul>
17.5.15: arraySize	always maps to	<ul style="list-style-type: none"> <li>• specification.attributes.arrayDim</li> </ul>

# 18 Equation and Algorithm Sections

## 18.1 Overview

SysML4Modelica	Modelica Abstract Syntax	Attributes
18.1.1: Equations and Algorithms::Modelica Equation	Absyn.EquationItem.EQUATIONITEM	See Sub clause 11.2
18.1.2: Equations and Algorithms::ModelicaConnection	Absyn.Equation.EQ_CONNECT	See Sub clause 11.4
18.1.3: Equations and Algorithms::ModelicaAlgorithm	Absyn.EquationItem.ALGORITHMITEM	See Sub clause 11.3

## 18.2 «modelicaEquation»

SysML4Modelica		Modelica
18.2.1: Equations and Algorithms::ModelicaEquation	maps to	Absyn.EquationItem.EQUATIONITEM
<b>Required</b>		
18.2.2: specification.body	parsed to	<ul style="list-style-type: none"> <li>equation</li> </ul>
<b>Conditionals</b>		
18.2.3: If isInitial <b>equal to</b> false		<ul style="list-style-type: none"> <li>EQUATIONITEM <b>contained in record typed to</b> ClassPart. EQUATIONS</li> </ul>
18.2.4: If isInitial <b>equal to</b> true		<ul style="list-style-type: none"> <li>EQUATIONITEM <b>contained in record typed to</b> ClassPart. INITIALEQUATIONS</li> </ul>

### 18.3 «modelicaAlgorithm»

SysML4Modelica		Modelica
18.3.1: Equations and Algorithms::ModelicaAlgorithm	maps to	Absyn.AlgorithmItem.ALGORITHMITEM
<b>Required</b>		
18.3.2: constraint.specification	parsed to	<ul style="list-style-type: none"> <li>algorithm</li> </ul>
<b>Conditionals</b>		
18.3.3: If isInitial <b>equal to</b> false		<ul style="list-style-type: none"> <li>ALGORITHMITEM <b>contained in record typed to</b> ClassPart. ALGORITHMS</li> </ul>
18.3.4: If isInitial <b>equal to</b> true		<ul style="list-style-type: none"> <li>A;GORITHMITEM <b>contained in record typed to</b> ClassPart. INITIALALGORITHMS</li> </ul>

### 18.4 «modelicaConnection»

SysML4Modelica		Modelica
18.4.1: Equations and Algorithms::ModelicaConnection	maps to	Absyn.Equation.EQ_CONNECTOR
<b>Required</b>		
<ul style="list-style-type: none"> <li>ConnectorEndA.Role</li> </ul>	maps to	<ul style="list-style-type: none"> <li>connector1</li> </ul>
<ul style="list-style-type: none"> <li>ConnectorEndB.Role</li> </ul>	maps to	<ul style="list-style-type: none"> <li>connector2</li> </ul>



# Part V - Annexes

## Contents

This part of the document introduces the Annexes, as follows:

- Annex A - Examples
- Annex B - Justification
- Annex C - QVT Transformation



# Annex A: Examples

(non-normative)

## A.1 A Car Suspension Model

The following example is intended to illustrate the concepts of how the transformation approach can be used to provide a context for the normative specification in Part II of this specification. Consider the design of a car suspension. As illustrated in Figure A.1, the suspension can be described in the context of a car using a descriptive SysML model, expressed in a BDD and corresponding IBD.

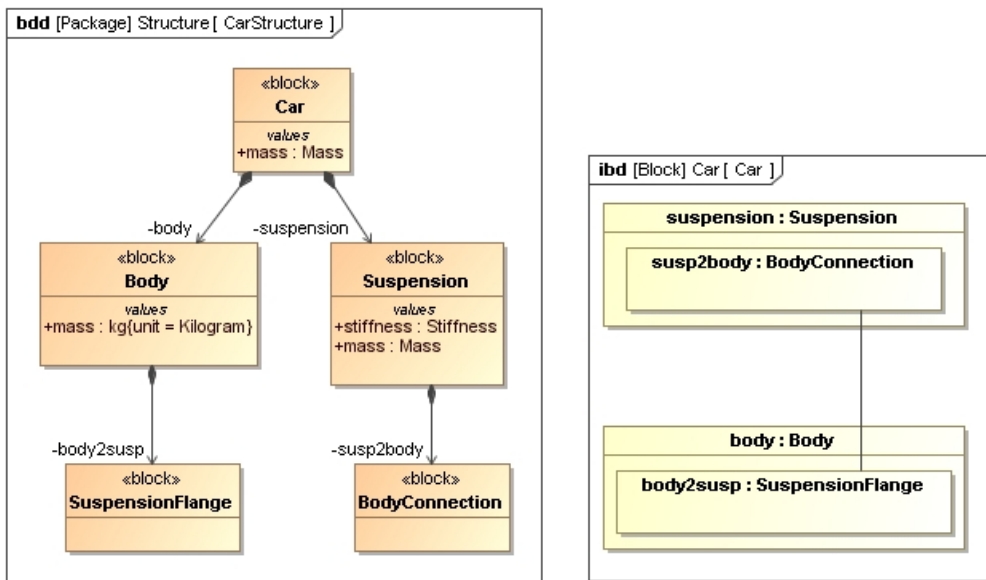
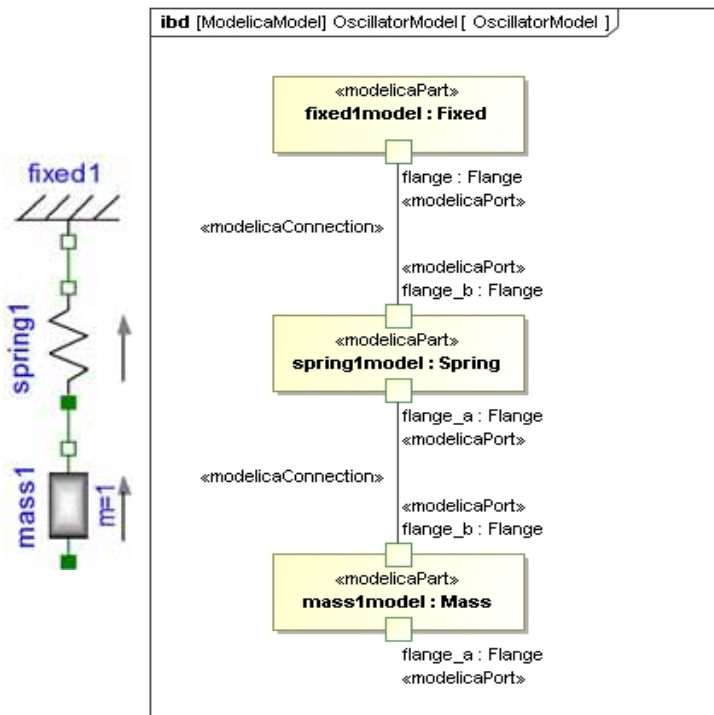


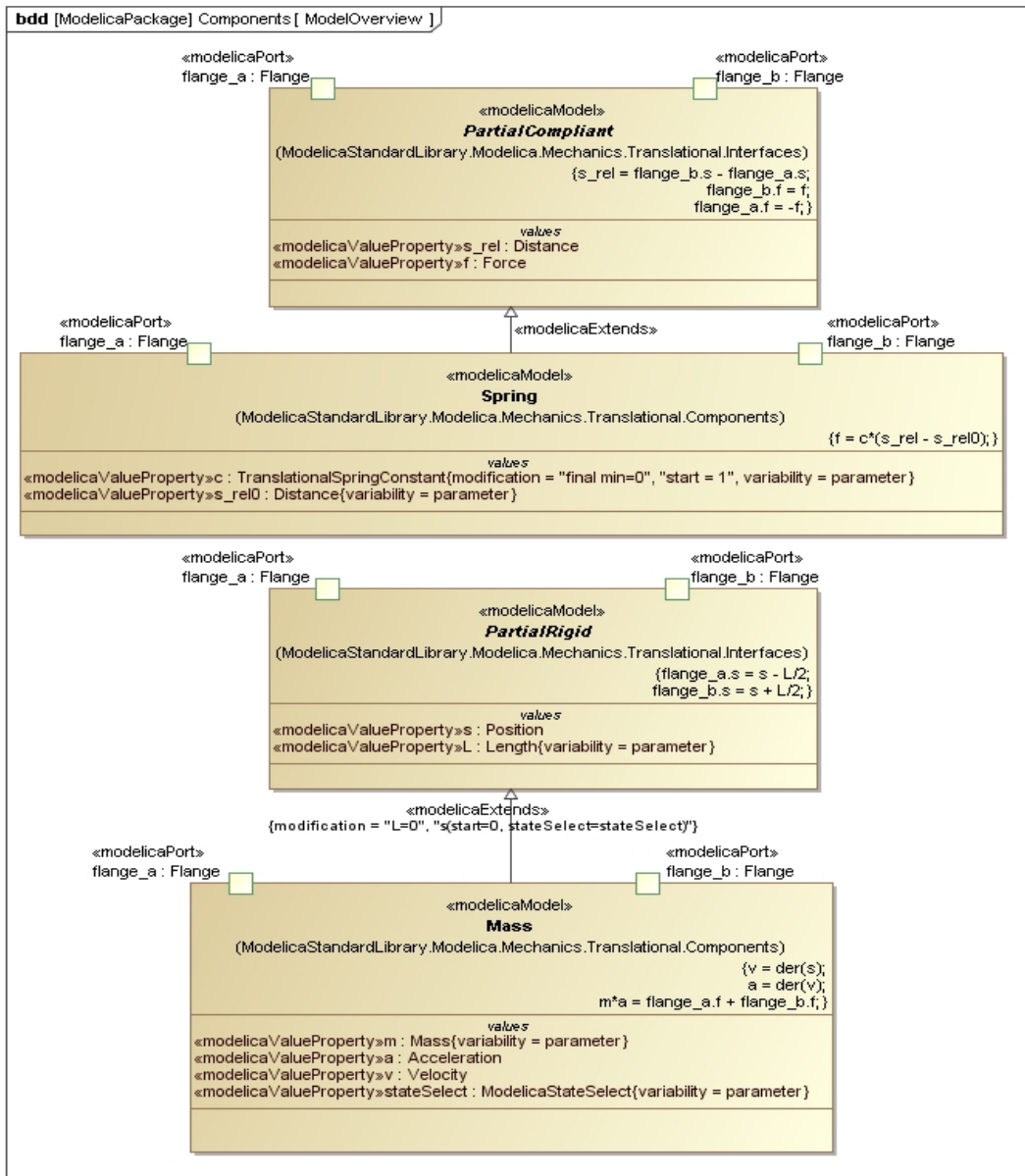
Figure A.1 - SysML descriptive model of a car suspension visualized as a BDD and IBD

Assume now that one needs to evaluate the dynamic response of the suspension by simulating the car body's position as a function of time. A possible continuous dynamics model for such a simulation models the suspension as a linear spring and the car body as a point mass. This model is illustrated in Figure A.2 in both Modelica and in the SysML4Modelica profile which represents the corresponding Modelica constructs. By stereotyping SysML ports and connectors, the semantics of Kirchhoff's laws have been introduced into SysML.



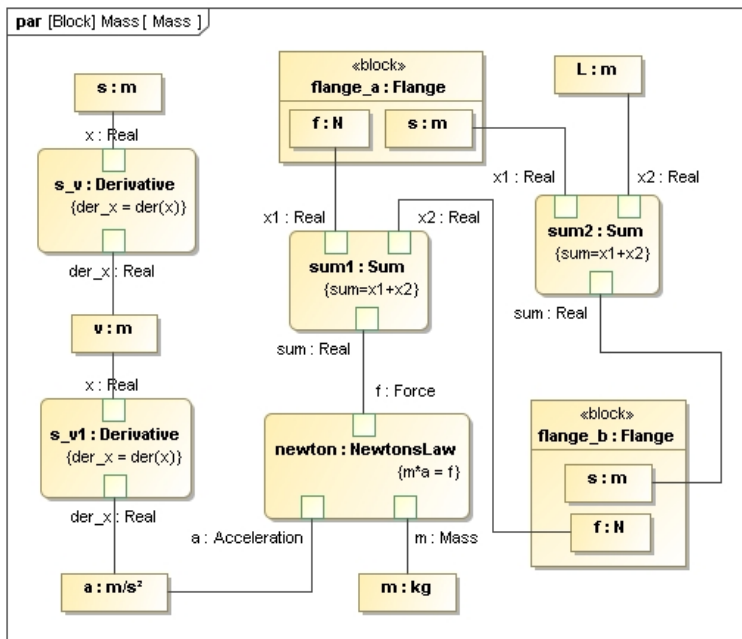
**Figure A.2 - Mass-Spring model for a car suspension, in Modelica (left) and SysML4Modelica (right)**

The SysML parts are stereotyped as «modelicaPart». (i.e., mass 1model, spring1model, fixed1model), that correspond to usages of models from the Modelica Standard Library. For instance, as illustrated in Figure A.3, the library Modelica.Mechanics.Translational.Components includes definitions of continuous dynamics models for a Spring and a Mass. Note that one could apply stereotypes in SysML that include icons equivalent to the elements from the Modelica library so that the SysML4Modelica representation in Figure A.4 could be almost identical to the Modelica representation on the left.



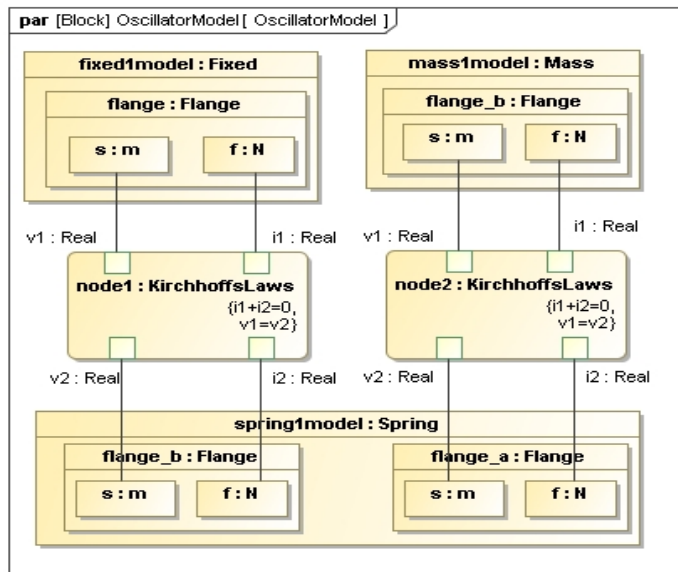
**Figure A.3 - Continuous dynamics models for Mass and Spring defined in the Modelica Standard Library**

In Figure A.2, the usages of these models, stereotyped as `<<modelicaPart>>` are connected to each other at their `<<modelicaPort>>` by `<<modelicaConnection>>`. These connections carry the semantics of Kirchhoff's Laws (in this example—or, more generally, the same semantics as an equivalent Modelica connection). These semantics can be made more explicit by using a Parametric Constraint (Figure A.4).



**Figure A.4 - Mass model as it could be represented in a Parametric Diagram**

But, as one can see by comparing Figure A.4 and Figure A.2, this comes at a cost of a much larger and less readable diagram. Similarly, one could have represented the internal equations of the Mass model in a Parametric Diagram, as is illustrated in Figure A.5, but again, the more explicit semantics come at a cost of increased complexity. For this reason, only Blocks and Internal Block Diagrams are further used in the SysML4Modelica profile, but the parametrics still provides the underlying metamodel for capturing the detailed equations. This complexity can often be abstracted and made not visible to the modeler.



**Figure A.5 - Mass-Spring model as represented in a Parametric Diagram**

Finally, it is worth illustrating how the SysML4Modelica continuous dynamics model in Figure A.2 relates to the SysML descriptive model in Figure A.1. Since both the descriptive and the continuous dynamics models are views of the same system, they cannot be independent of each other. Changes to the descriptive model are likely to require corresponding changes to the continuous dynamics model and vice versa. Such dependencies can be modeled in an analysis context - the context in which the analysis model (i.e., the continuous dynamic analysis in this case) is defined.

The analysis context is illustrated in Figure A.6. It establishes the dependencies between the descriptive model components and their corresponding analysis models. In addition, the detailed bindings between the descriptive and analysis properties are defined in the Parametric Diagram illustrated in Figure A.7.

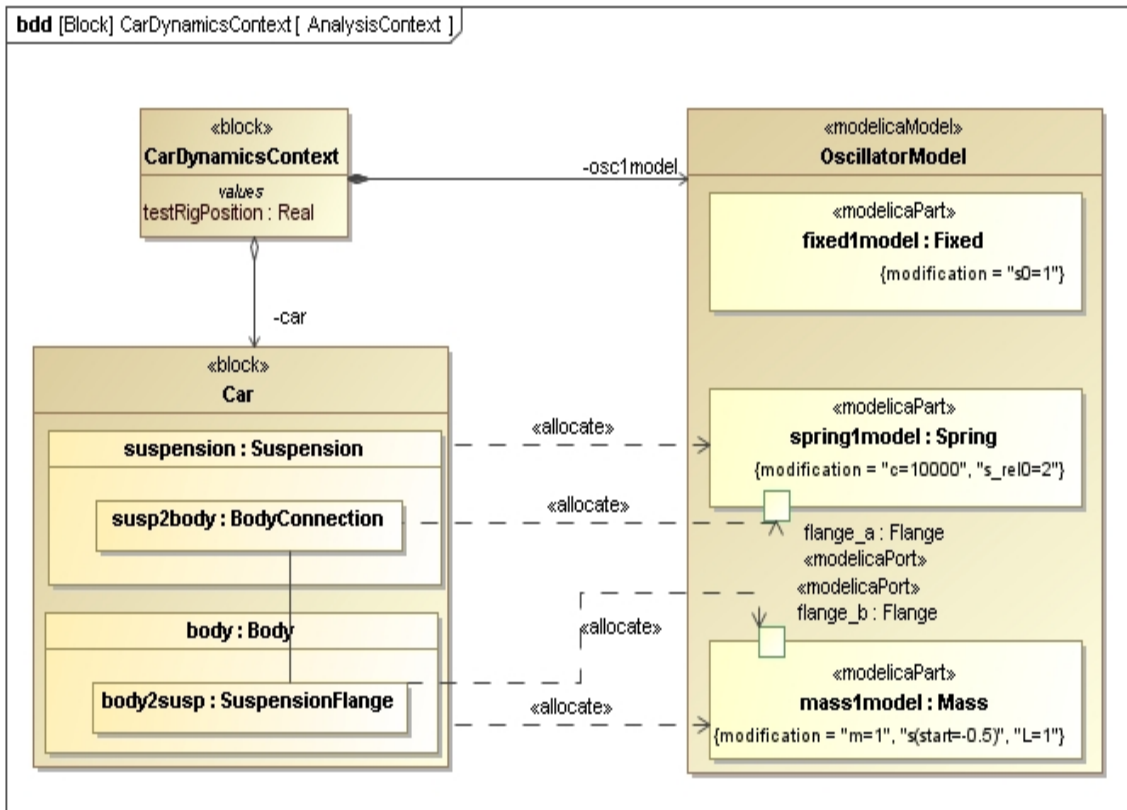
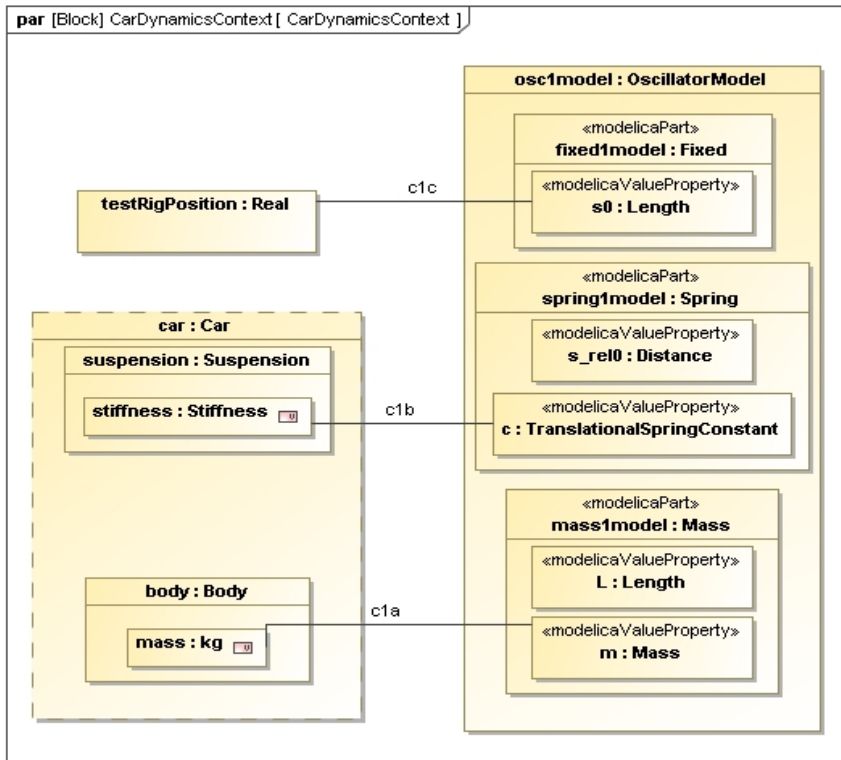


Figure A.6 - The Block Definition Diagram for the Analysis Context of the continuous dynamics analysis





**Figure A.7 - The Parametric Diagram for the Analysis Context of the continuous dynamics analysis; the properties of the descriptive model are bound to the corresponding properties in the analysis model.**

For very simple problems, one could consider combining the descriptive and analysis views into one model; e.g., suspension and spring1model would be combined into one component that includes both the descriptive properties and the analysis constraints/ equations. However, for larger problems in which more than one analysis perspective needs to be considered (e.g., mechanical, electrical, controls, manufacturing, different levels of abstraction, etc.), combining all such analyses into one model would be difficult to manage. One would likely encounter problems with naming conflicts or duplication of properties. In addition, combining all the models severely limits the opportunity for model reuse because models from libraries (such as the Modelica Standard Library) would have to be combined with descriptive models rather than just included in an analysis context.

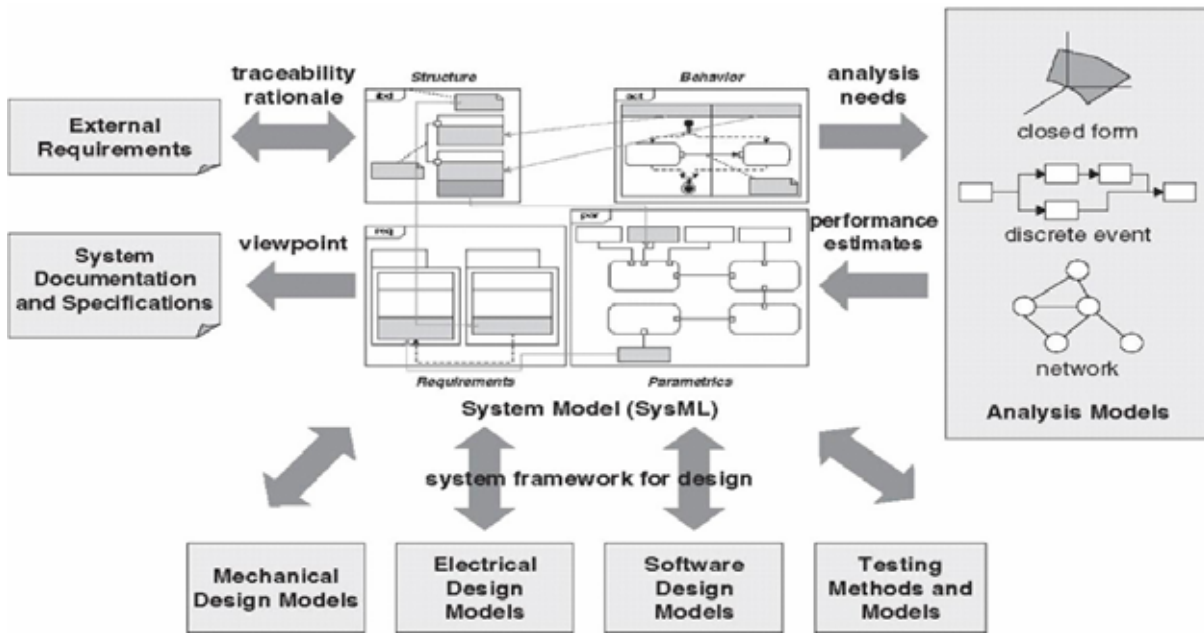
## A.1.1 A Robot Model

### Introduction

The example in this section is intended to illustrate how a SysML model can be transformed to a Modelica model in accordance with the transformation approach specified in this document. In particular, the transformation is accomplished by first applying the SysML4Modelica profile as described in Part II of this document, and then mapping the SysML4- Modelica model to the Modelica model as described in Part IV of this document. The robot example is based on the robot model that is contained in the standard Modelica library which can be found at [www.modelica.org](http://www.modelica.org). Refer to Part I of this document for a brief introduction to SysML and Modelica.

## Integrating SysML Descriptive Models with Analytical Models

This transformation specification will typically support the system requirements analysis and design activity as part of a systems engineering process. A SysML model will be developed to specify the system requirements, architect the system, and allocate the system requirements to the hardware and software components of the system. The SysML model serves as a descriptive model to capture multiple aspects of the system of interest, including its functionality, inputs/output and control flow, structural composition and interconnection, and traceability to its text based requirements as indicated in Figure A.8. As part of the requirements analysis and design effort, many different engineering analysis are often performed to evaluate the extent that the system can satisfy its system performance, physical, reliability, maintainability, and cost requirements.



**Figure A.8 - A SysML model in which models for multiple analysis tools are defined**

The SysML descriptive model can capture relevant aspects of the system that can be used by many different types of analytical models and tools to support the above analysis. One mechanism is to use SysML parametrics to capture the analysis as a network of equations, and then pass this analysis to an analytical tool. The analytical tool then performs the computation and provides the quantifiable results back to the SysML model. A simple example may be for a SysML parametric model to capture the system overall reliability in terms of the mean time between failures of each of its components.

The reliability of each component may in turn be estimated based on some equation. This set of equations are passed to a reliability analysis tool to perform the computation, and return the reliability values back to the SysML model.

Sometimes, SysML parametrics is used in a more abstract way. In this case, the SysML model does not capture the equations, but only the input and output parameters of the analysis. When this is done, the equations that relate the input and output parameters of the analysis are included in the analytical tool or solver.

An alternative approach for providing relevant aspects of the descriptive model to an analytical model is to use the transformation approach specified in this document. In this particular case, the SysML model is transformed to a Modelica model in two steps. First, the SysML4Modelica profile is applied to create an analytical representation from the structural portion of the SysML model. In the second step, this SysML4Modelica analytical model is mapped to the

Modelica model where it can be executed. The additional step to apply the SysML4Modelica profile to create the analytical model facilitates a more straightforward mapping from SysML to Modelica, as compared to mapping the SysML model to Modelica directly without applying the profile.

This transformation approach provides advantages over creating a parametric model and providing the parametric model to the Modelica model directly as describe above. In particular, the approach enables the SysML4Modelica analytical model to more effectively map to reusable components in the Modelica standard library. The Modelica model encapsulates the equations in its components, and then defines standard equations for connecting them. The detailed equations are generally assumed to be captured in the Modelica model using Modelica's textual notation. It is generally assumed that the SysML4Modelica analytical model captures the structure, interconnection, and properties, but not the detailed equations. This transformation approach allows the modeler to provide an abstract description of the system in SysML and the SysML4Modelica analytical model, and then establish direct correspondence to the Modelica model.

## A.1.2 Robot Example

This robot example only highlights the aspects of the SysML model that are used in the SysML4Modelica transformation. The primary aspects of the SysML model that are used in the transformation are the block definition diagrams (bdd) and the internal block diagrams (ibd). In a more typical case, the SysML model would include other aspects of the model as described in Figure A.8, and integrate with other analytical models and tools as well as the Modelica model.

For the robot example, the block definition diagrams and internal block diagrams are used to describe the system composition and interconnection at increasing levels of detail. This is typical of how SysML models are developed to support system specification and design. The corresponding Modelica analytical model may be created at different levels of abstraction. The following paragraphs illustrate a sequence diagrams one may create in a modeling and design process. All figures are included at the end of the section.

The SysML model organization for the Robot model is shown in the package diagram in Figure A.9. The model structure includes the SysML4Modelica Profile, the Modelica Standard Library, and the Robot Model itself. The Robot model includes packages for defining interfaces, types, structure, and analysis.

As described above, a typical SysML model may include integration with a diverse set of analytical models. The analysis package captures the various types of analysis that are being performed. In particular, Figure A.10 shows a parametric model of the top level objective function for the robot. In particular, several key performance parameters have been identified that characterize the overall value to the end user, including the weight, power, reliability, cost and trajectory performance in terms of the position error. Each of these performance parameters are analyzed by different analytical models and tools. Note that the Modelica model will be used primarily to analyze the trajectory performance. This is indicated by the refine relationship between the Modelica robot model and the trajectory performance model.

The top level SysML block definition diagram is shown in Figure A.11. The robot domain block serves as a context for the robot, which is the system of interest. The robot domain block is composed of the robot and the other actors that are external to the robot, and interact with it. The actors include the load the robot manipulates, the platform the robot is attached to, the power source that provides power to the robot, and the driver that provides the desired trajectory input to the robot. The trajectory input may be provided in real time, such as might be done by joystick control, or prior to the robot actually executing the trajectory.

In Figure A.12, the corresponding internal block diagram is shown. In this diagram, the interconnection between the robot and the actors is shown. The ports on the robot represent the connection points to each external actor.

The top level bdd and ibd are sometimes referred to as a black box view which specify the robot from an external perspective without any internal details. The corresponding Modelica model may be created to provide an abstract analytical representation of the black box robot, with limited or no internal detail. This analytical model may be used to

assess required trajectory characteristics, such as precision and response time to manipulate a load of specified mass, and perhaps the minimum power requirements needed of the robot, based on some assumptions on a robot power efficiency factor. Again, this analysis may be performed without any consideration for the internal details of the robot.

The standard Modelica library does not include this black box model explicitly. However, it could be added by creating the SysML4Modelica analytical model and developing the corresponding Modelica model. Although the robot model may be abstract, the models of the actors such as the Load, Power Source, and Driver could be specified in detail and reused for the detailed robot analytical models.

The block definition diagram in Figure A.13 decomposes the robot into its next level of components including the Path-Planner, Control Bus, Actuators, and Arm. Only one of the six actuators is shown in the bdd. The Actuators are all assumed to be of the same type, but each actuator could have been modeled as a subclass of a more generic actuator to represent a unique component type.

The internal block diagram in Figure A.14 shows the interconnection among the robot parts. Note that the black box interfaces to the external actors are preserved. Each actuator is shown as a unique part. Once again, a robot designer may choose to perform an analysis of the robot at this level of abstraction, where all of the components in the ibd are treated as black boxes without internal detail. This would further refine the black box analysis, and provide a basis for allocating specific performance requirements to the components. For example, the actuator efficiency could be estimated, and the trajectory could be analyzed as a function of different assumptions of actuator black box characteristics. Again, the Modelica library does not explicitly contain a model of these components at this level, but the Modelica model could be expanded to include them. If so, the SysML4Modelica analytical model would be created, and then mapped to the corresponding Modelica model.

The next diagrams include the block definition diagram and internal block diagram for the actuator and arm. The path planner and control bus were not further decomposed in the SysML model, although they could have been. The actuator block definition diagram and internal block diagram are shown in Figure A.15 and Figure A.16, respectively. The actuator includes the Controller, Motor Assembly, Gear and Sensor. The Motor Assembly is further decomposed into a Motor and Drive Electronics on the bdd, but no further interconnection detail is shown. The level of detail of the SysML model typically corresponds to the level of detail that the system is being specified by the system designer. Below this level, other domain specific hardware and software models are used to model the system design.

The Arm block definition diagram and internal block diagram are shown in Figure A.17 and Figure A.18, respectively. Note that the black box interfaces for the actuator and arm are preserved on their internal block diagrams, providing consistency from the robot black box level to the component 1 level.

The transformation to the Modelica model is performed at this level of detail of the SysML model of the robot. The first step in the transformation is to create the SysML4Modelica analytical model. In Figure A.19 and Figure A.20, the SysML structural model is allocated to corresponding elements of the SysML4Modelica analytical model. Based on these allocations, the SysML4Modelica analytical model for the robot is shown in Figure A.21.

Once the SysML4Modelica analytical model has been defined, the mapping to the corresponding Modelica model can be performed. Figure A.22 shows the corresponding graphical representation of the resulting Modelica model. The detailed equations are embedded in the Modelica model elements that are represented by the graphical elements.

In Figure A.23, the results of the analysis are shown for a specific simulation execution.

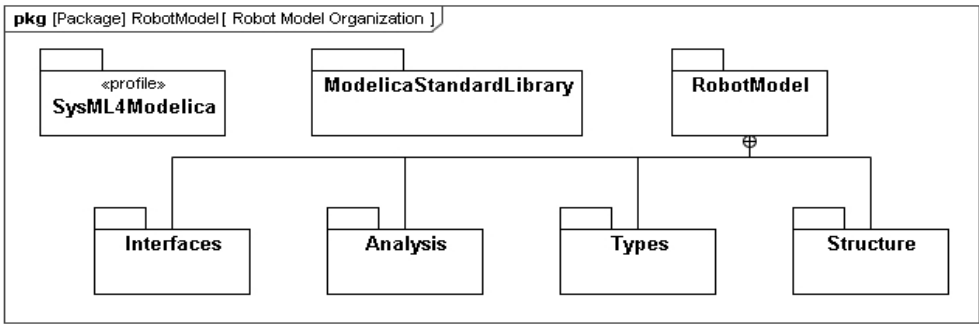


Figure A.9 - The package organization of the robot model

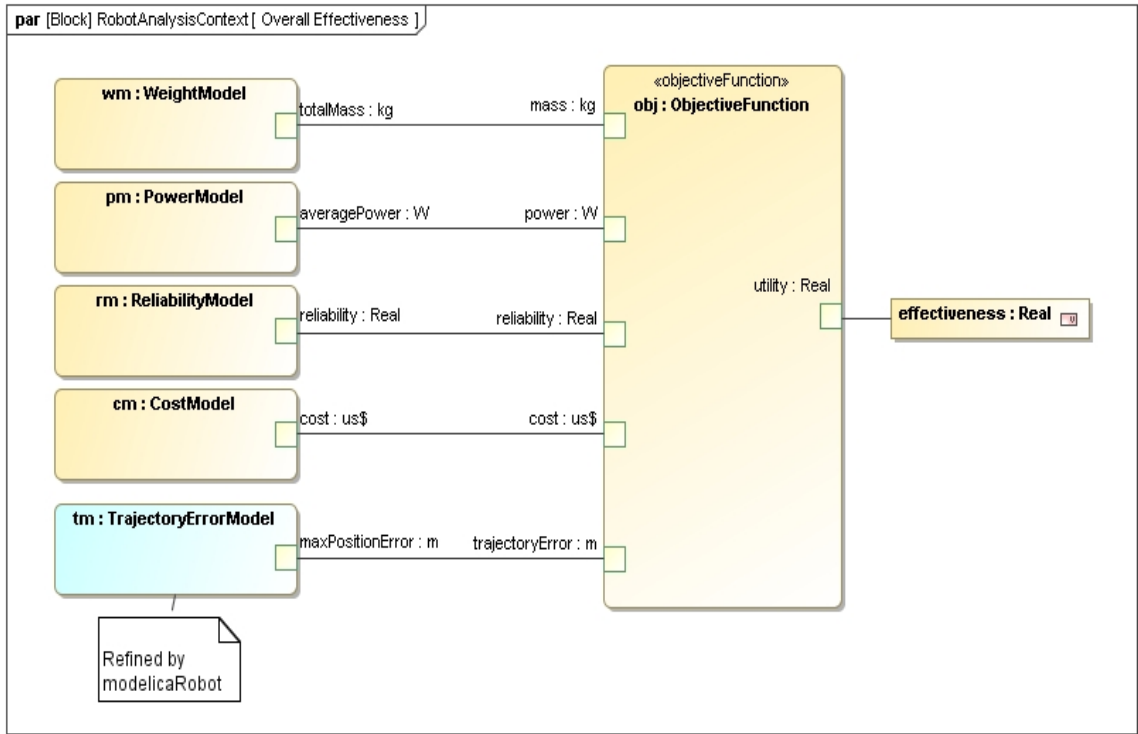


Figure A.10 - A parametric model of the top level objective function for the robot

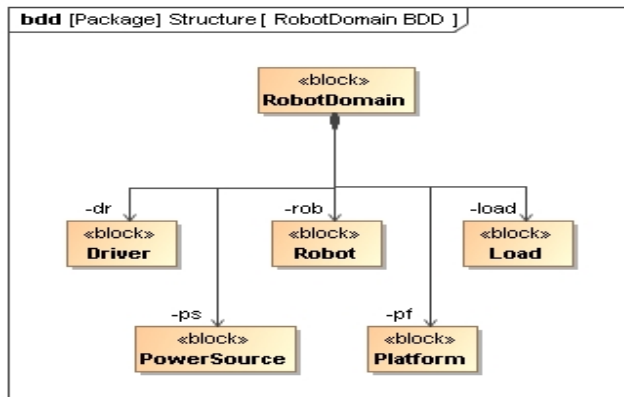


Figure A.11 - The top level block definition diagram for the robot domain

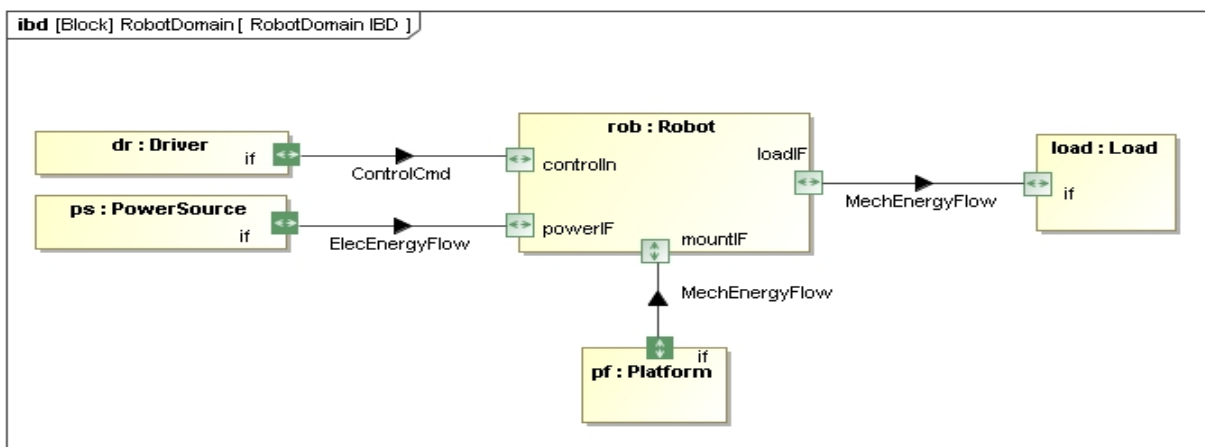


Figure A.12 - The internal block diagram of the robot domain

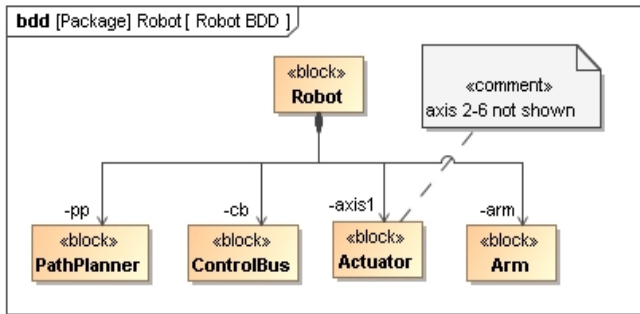


Figure A.13 - The block definition diagram for the decomposition of the robot into its main subsystems

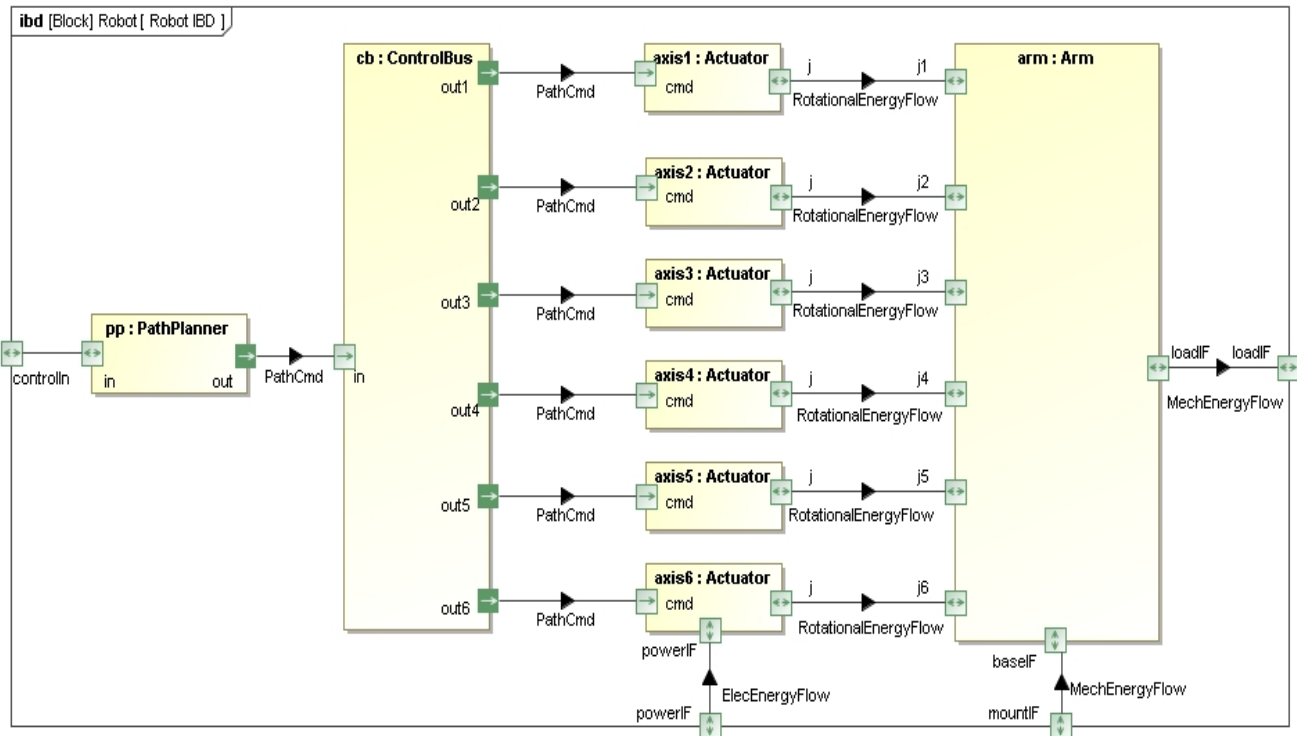


Figure A.14 - The internal block diagram for the robot, illustrating its decomposition into the path planner, the control bus, the actuators and the mechanical structure of the robot arm.

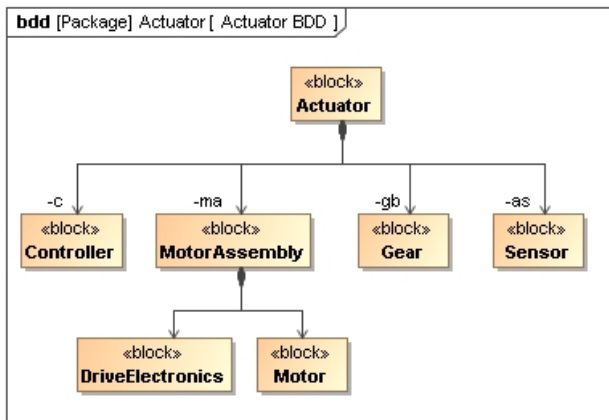


Figure A.15 - The block definition diagram for the structure of an actuator of the robot

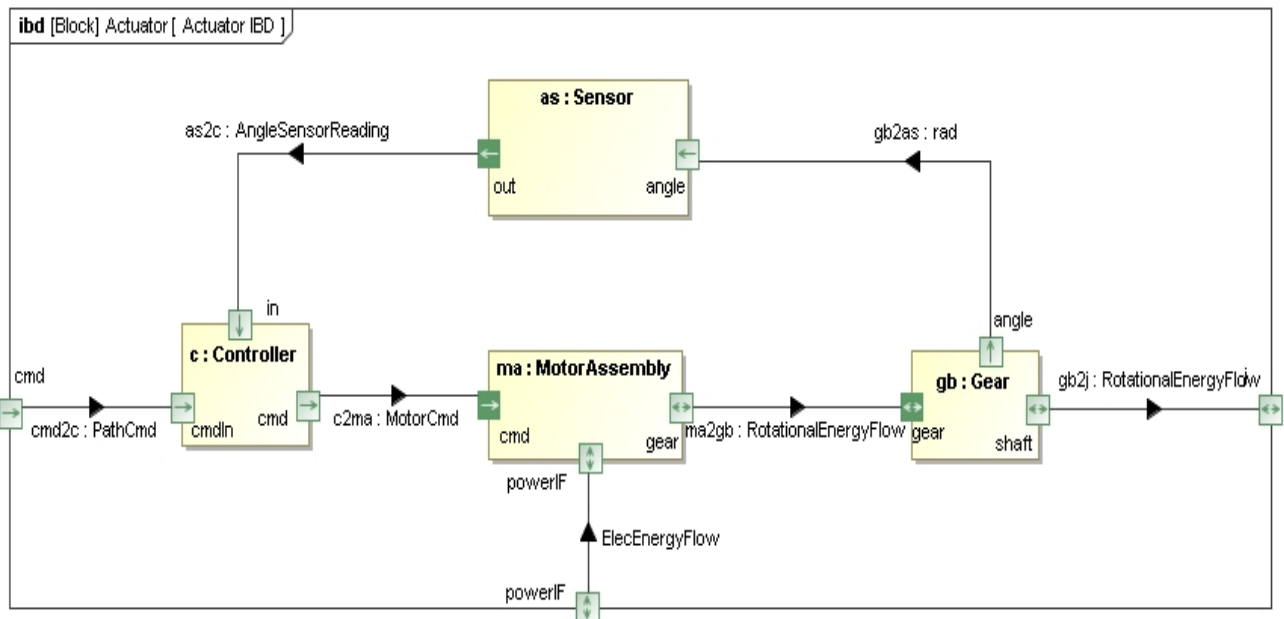


Figure A.16 - The internal block diagram for a robot actuator



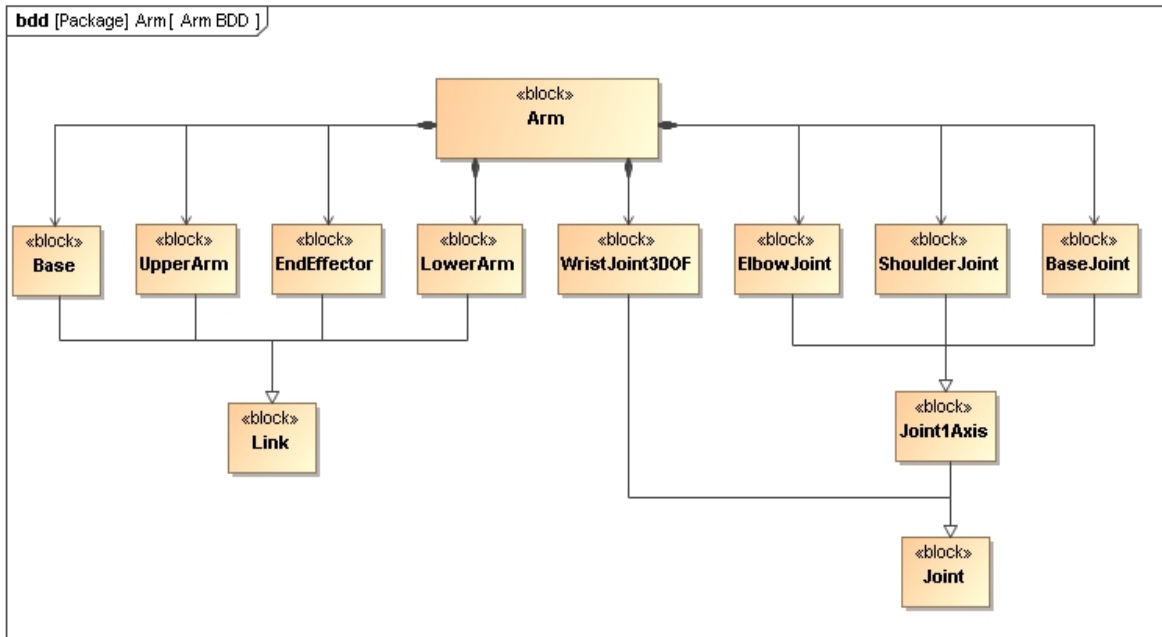


Figure A.17 - The block definition diagram for the robot arm's mechanical structure

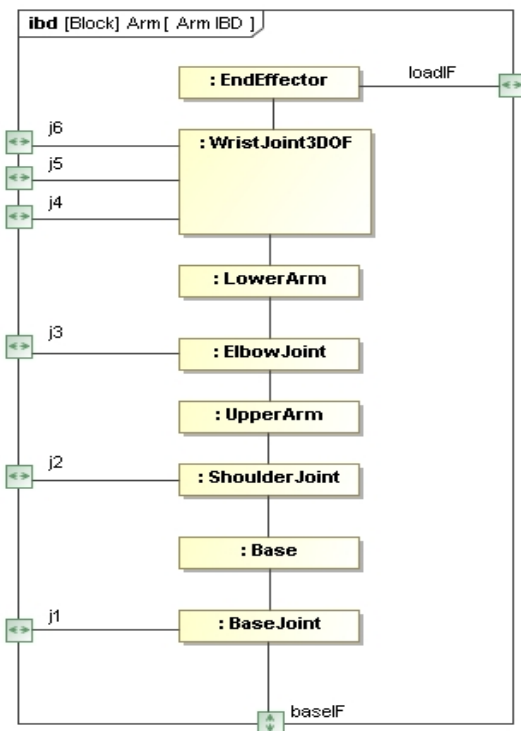


Figure A.18 - The internal block diagram for the robot arm

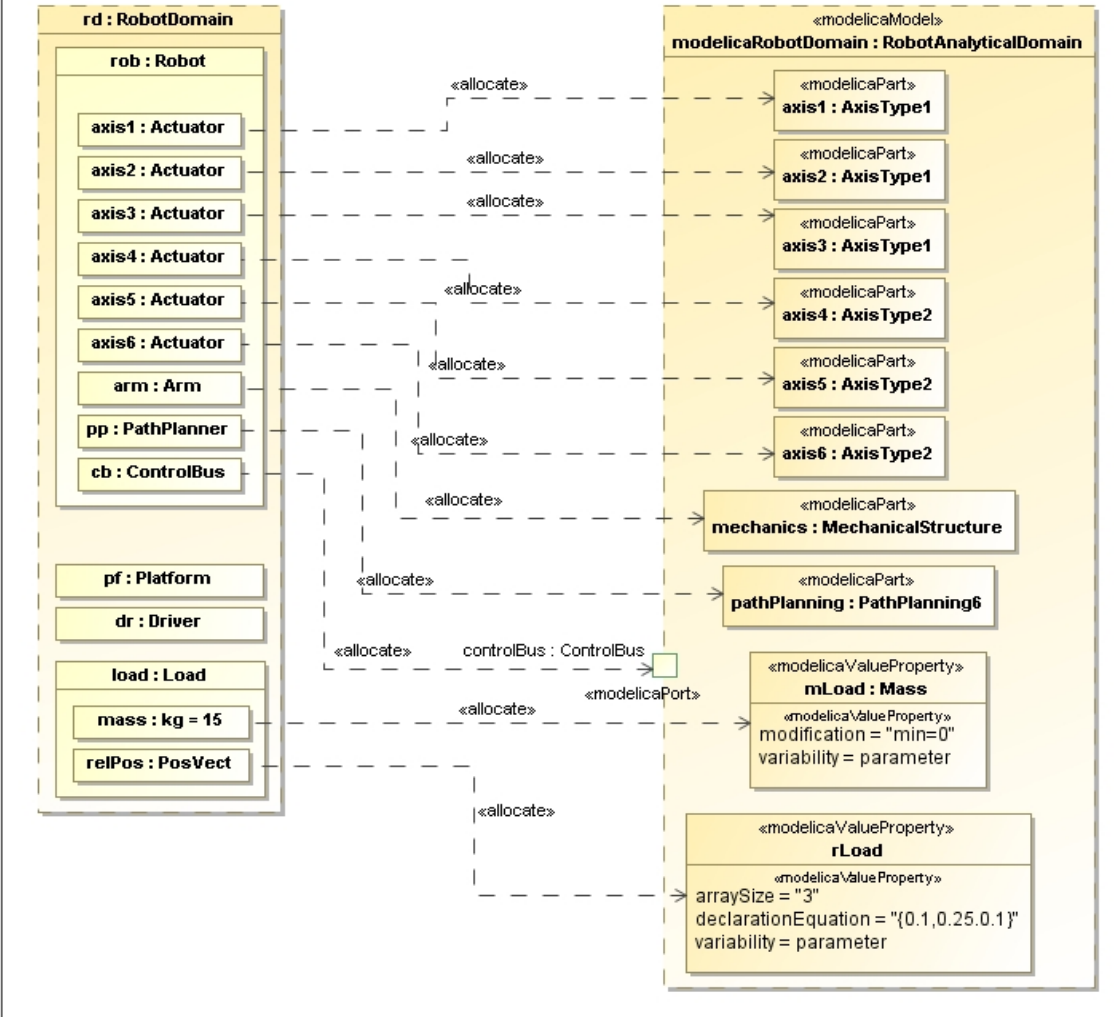
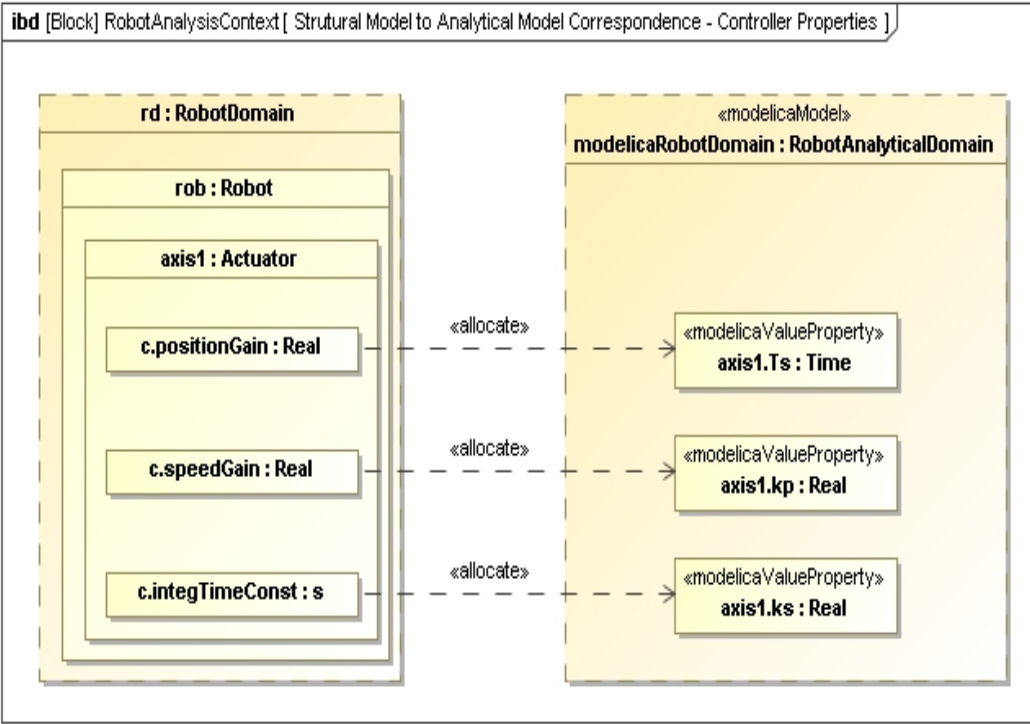


Figure A.19 - The Analysis Context in which the descriptive model of the robot domain is allocated to the corresponding analytical model as expressed in the SysML4Modelica profile.



**Figure A.20 - A detailed diagram of the allocation of the robot actuator descriptive model to the analytical SysML4-Modelica Model**

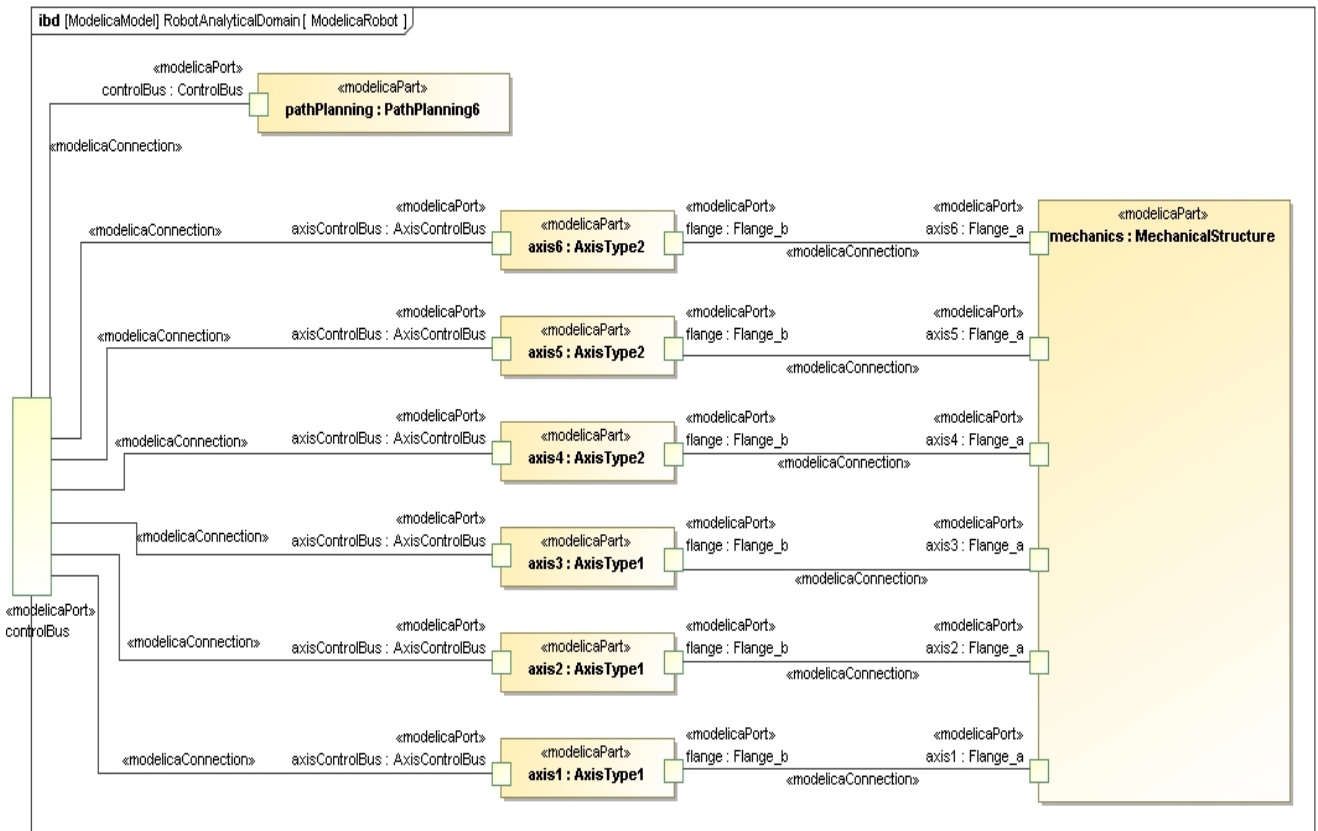


Figure A.21 - The top-level robot problem shows as an ibd in the SysML4Modelica profile

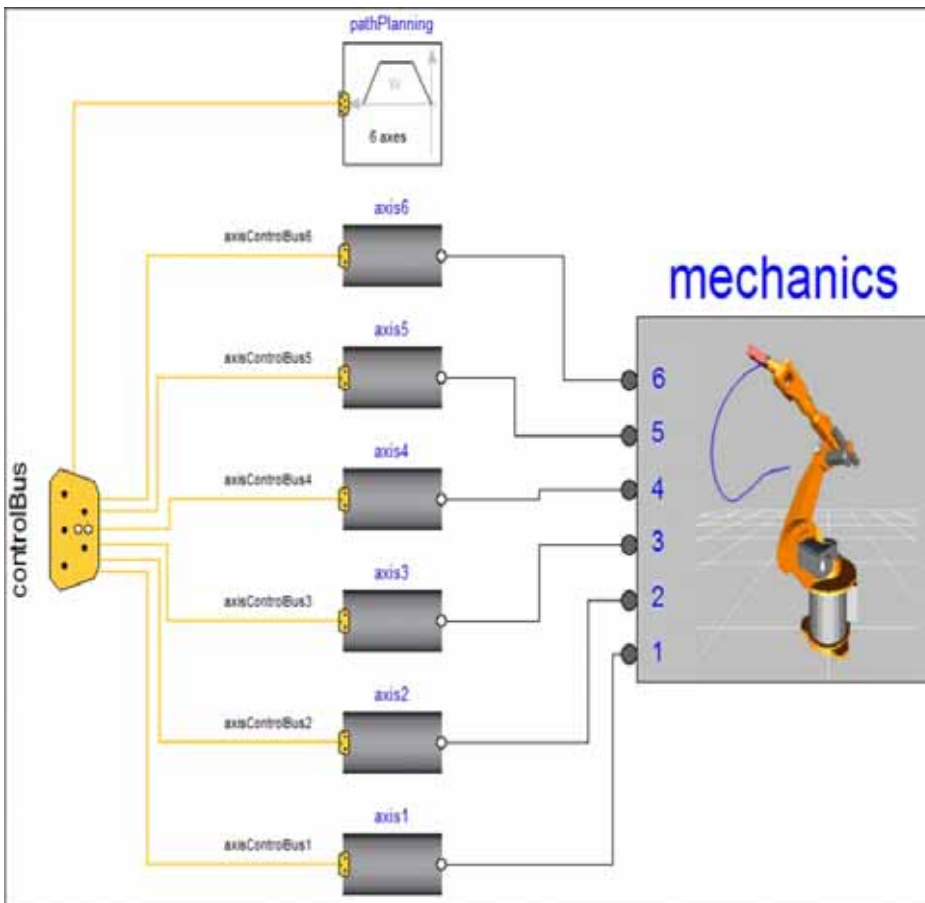


Figure A.22 - The top-level Modelica model of the robot

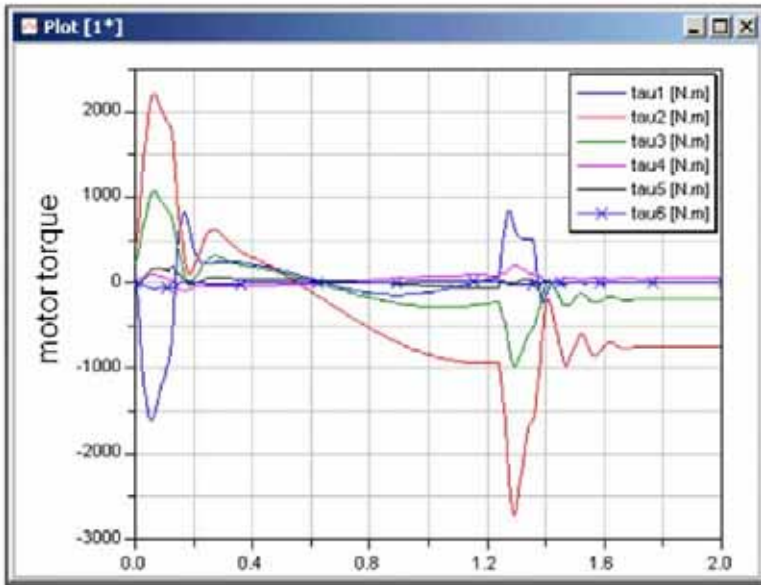


Figure A.23 - The simulation results with the motor torques as function of time

# Annex B: Justification

(non-normative)

## B.1 Semantic Comparison between SysML and Modelica

Before focusing on the detailed modeling constructs, a high-level decision needs to be made regarding the choice of SysML elements to represent Modelica models. Although Modelica is a textual language, it also supports a graphical view through its annotation mechanism. This graphical view illustrates clearly the strong similarity that exists between SysML and Modelica. Both languages support the decomposition of systems (or behavioral models of systems) into subsystems or components and the interactions between them. For instance, the Modelica model of a motor controller (shown in Figure 3) contains sub-components (such as motor, gearbox, and controller). The interactions between them are illustrated by edges connecting the interface locations (called connectors in Modelica) of the components. Such hierarchical compositions of Modelica models and the connections between them constitute the primary modeling approach in Modelica. Before considering the details of the language, it is thus important to consider carefully how these primary modeling constructs map to SysML.

As illustrated in Table B.1, in SysML there are three kind of construct built on abstractions that have similar semantics compared to the hierarchical, connector-based composition of Modelica models: the hierarchical Blocks, shown in Internal Block Diagrams), the Parametric Constraints (shown in Parametric Diagrams), and the Activity graphs. All three constructs support some sort of “ports,” some sort of connection of “port-based” objects through “port-connections,” and hierarchical encapsulation through “port-delegation.” We use these three constructs to discuss the main question: “What are the SysML elements that match the Modelica semantics best?”

**Table B.1 - A comparison between Modelica concepts and SysML abstractions and diagrams**

Concepts	Modelica constructs	SysML				
		Construct abstractions	Availability in diagrams			
			<----- Modelica “like” ----->			
BDD	IBD	Parametric	Activity			
<i>Model Definition</i>	Model	Block	Yes	Yes	Restricted	No
<i>Model Usage</i>	Component	Property (Part Property)	Yes	Yes	Restricted	No
<i>Port Definition</i>	Connector	Block	Yes	Yes	Yes	No
		ValueType	Yes	Yes	No	No
		FlowSpecification	Yes	No	No	No
<i>Properties</i>	Component (Variables)	Block	Yes	Yes	Yes	Ref.Only
		ValueType	Yes	Yes	Yes	Ref.only
		FlowProperty	Yes	No	No	Ref.only
<i>Part</i>	Component	Port	Yes	Yes	Yes	Ref.only
<i>Causal link</i>	Connection	Connector	No	Yes	No	No
		ObjectFlow	No	No	No	Yes
<i>Acausal link</i>	Connection	Connector	No	Yes	Yes	No

## B.2 Modelica

In Modelica, ports are called connectors and the edges between ports are called connections [Modelica Spec, Chapter 9]. The ports (connectors) can include four types of quantities: inputs, outputs, flows and non-flows. Inputs and output are used when the direction of the flow is known and fixed, as for instance in signals flowing in a control system. Flow and non-flow quantities are used to describe energy or material flow (they are also sometimes referred to as through and across variables, respectively). When connecting two Modelica connectors with a connection, the semantics for inputs and outputs are causal binding: the input is assigned the value of the output to which it is connected. Input and output connectors must therefore be used in conjugate pairs, and only one output can be connected to each input. For flow and non-flow variables, the connection semantics correspond to Kirchhoff's Laws, namely, the value of the flow variables add up to zero and the values of the non-flow variables are set equal (in an equation-based, acausal fashion). When more than one connection is made to a connector containing a flow variable, then an ideal, loss-less energy or material exchange is assumed by imposing that the values of flow variables of all connected connectors add up to zero. To impose the correct modeling of energy exchange, Modelica requires that the number of flow and non-flow quantities of a connector be equal.

In addition to connectors, Modelica models can contain variables and submodels (i.e., model usage in Table B.1). Although Modelica does not explicitly distinguish between these three categories of “components” (i.e., connectors, variables, sub-models), it may still be useful and desirable to distinguish explicitly among them when mapping to SysML.

## B.3 SysML Hierarchical Blocks, ports and connectors

The primary purpose of the SysML hierarchical Block constructs, is to express system structural decomposition and interconnection of its parts [SysML Spec, Chapters 8 and 9]. The SysML concepts used in those constructs have quite flexible semantics and may be used to establish logical and conceptual decompositions, for instance, as in a context view [SysML Spec, Section B.4.2.1]. The Blocks in SysML are similar to Classes in Modelica (specifically the specialized class types of Model, Block, Connector, etc.). Blocks can be decomposed in the same way Modelica Classes can be decomposed.

The “ports” on the blocks are called Ports and the connections between ports are called Connectors. There are two kinds of ports: Flow Ports and Standard Ports. The Standard Ports are particularly geared towards service-based interactions by representing the interfaces (e.g., software methods) that are provided or required by a particular block. Such service-based interactions are not appropriate for modeling the connections found in Modelica. Flow Ports on the other hand do provide semantics that reflect Modelica connectors more closely.

A Flow Port describes an interaction point through which input and/or output of items such as data, material, or energy may flow in and out of a block. For Modelica-type interactions, the exchanged "items" could be either signals (for input and output quantities) or energy/material (for flow and non-flow quantities). Modelica signal exchanges are causal and so the semantics of a SysML Flow Port typed by a Flow specification is convenient. SysML binding connectors provide acausal connections between properties. They imply equality between connected properties and then does not carry the Kirchhoff laws semantics. The equivalent of a Binding Connector does not actually exist in Modelica, but can be captured in a non-graphical fashion by introducing an equality equation between the two variables that are bound. Therefore, in order to capture the semantics of a Modelica connection, one solution would be to introduce a new SysML connector element that is equivalent to a Modelica Connector, and that reflects the semantics of Kirchhoff's laws. Another possibility would be to make the equations for Kirchhoff's laws, which are implicit in Modelica connections, explicit as another SysML Constraint Property. This option is appealing because it makes the semantics very explicit, but has the disadvantage that it makes the models more cumbersome to create and more difficult to read.

In conclusion, although blocks seem to have very similar constructs to Modelica, there are some subtle differences in so that new stereotypes will have to be introduced to adequately capture the Modelica semantics of Connectors and Connections.



## B.4 SysML Parametric Constraints

The purpose of Parametric Constraints is to express mathematical relationships between parameters. A Parametric Constraints is modeled through a special kind of Block named “Constraint Block”. “Ports” of those blocks are Constraint Parameters and the “connections” to those parameters are made using Binding Connectors. Inside a Constraint Block, mathematical relationships are defined constraining its Constraint Parameters. A Constraint Property is a usage of a Constraint Block. Its Constraint Parameters are then bound to other Constraint Parameters or to Properties of Blocks. The semantics of a Binding Connector indicate a mathematical equality between the (Block) Properties or Constraint Parameters being connected. This mathematical equality is an acausal relationship.

## B.5 SysML Activity Graphs

The purpose of an Activity graph in SysML is to specify the transformation of inputs to outputs through a controlled sequence of actions. An Activity decomposes into Actions. In activity graphs, the Object Nodes (i.e., Pins and Parameter Nodes) correspond to buffers to place input and output tokens. The connections between Object Nodes correspond to Object Flows. These flows typically represent the transfer of one or more objects at a discrete moment in time, although it is possible to specify a streaming flow that could be continuous, i.e., the time between arrival of tokens (or “objects”) is zero. It is this latter case that needs to be described in terms of differential equations.

It must be underlined that as defined in the context of SysML activities “flows”, are they continuous or not, correspond to the concept of “dataflow” which is related to an asynchronous approach. Conversely, a Modelica flow specifies the existence of relationships between the value of respectively flow and non-flow variables on both sides of a connection, as defined by Kirchhoff’s laws. Those relationships are mathematical equations and then corresponds to a synchronous approach.

In conclusion, SysML Activity Graphs can be convenient only to model Modelica input/output variables. Thus Activity graphs therefore seems to be the least appropriate for a mapping from Modelica Class, although they will be explored when mapping the Modelica Function and Algorithm to SysML4Modelica.

## B.6 Selected foundation concept: SysML Hierarchical Blocks with Embedded Constraints

It is clear from the discussion in the previous sub clauses that there is not a single concept that embeds the Modelica semantics perfectly. As a result, the use of more than one SysML concept with multiple stereotypes will need to be defined to extend the SysML semantics.

Blocks, ConstraintBlocks, FlowPorts, classical Connectors and BindingConnectors can be used to map Modelica Models, Components, Connectors, and Connections to SysML. This is illustrated in Annex A.



# Annex C: QVT Transformation

(non-normative)

## C.1 Overview

The overview of an implementation of the SysML-Modelica Transformation based on QVT is shown in Figure C.1. The QVT transformation files can be retrieved at these URLs:

- QVT Transformation: <http://www.omg.spec/SysM/20120214/Modelica2ModelicaUnparsed.qvto>
- QVT Transformation: <http://www.omg.spec/SysM/20120215/ModelicaUnparsed2SysML.qvto>
- QVT Transformation: <http://www.omg.spec/SysM/20120216/SysML2ModelicaUnparsed.qvto>
- QVT Transformation: <http://www.omg.spec/SysM/20120217/ModelicaUnparsed2Modelica.qvto>

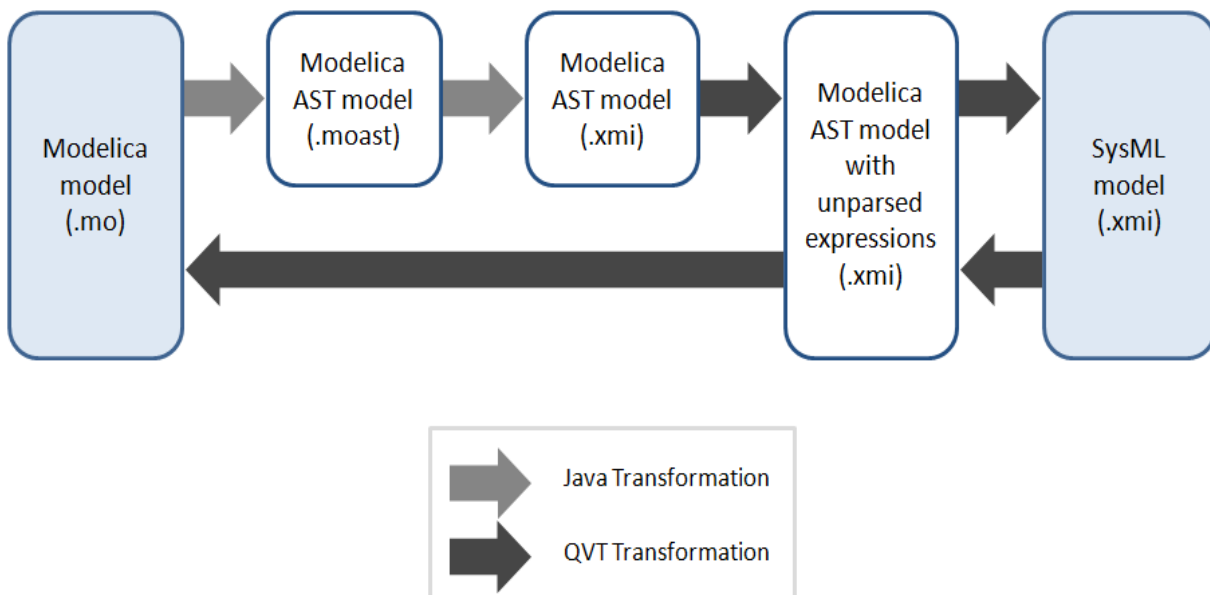


Figure C.1 - Overview of an implementation of the SysML-Modelica Transformation based on QVT

