Date: July 2016

SysML Extension for Physical Interaction and Signal Flow Simulation (SysPISF)

FTF – Beta 1

OMG Document Number: dtc/2016-07-01

Standard document URL: http://www.omg.org/spec/SysPISF/1.0/PDF

Machine consumable files:
Normative:
http://www.omg.org/spec/SysPISF/20160201/SysPISFProfile.xmi
http://www.omg.org/spec/SysPISF/20160201/SysPISFLibrary.xmi

This OMG document replaces the submission document (mantis/16-03-10). It is an OMG Adopted Beta specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be entered by October 1, 2016 using the Issue Reporting Form on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).

The FTF Recommendation and Report for this specification will be published on December 16, 2016. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

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.

LICENSES

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 this document and to modify this document 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, non-transferable, perpetual, worldwide license (without the right to sublicense), 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; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. 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 the specifications in your possession or control.

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

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: http://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that 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 these materials.

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 on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).

Table of Contents

Preface

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 the OMG website at: http://www.omg.org/spec. 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

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).

1 Scope

Systems engineers coordinate the work of multiple other engineering disciplines (mechanical, material, electrical, control, and so on), requiring information to flow between systems engineers and those in other disciplines. Systems engineering information intentionally does not cover all disciplines, but must integrate with them to enable systems engineers to communicate with other engineers. Using discipline-specific tools separately from system modeling tools typically leads to redundancy, inconsistency, and less efficient engineering processes.

Many engineering disciplines (mechanical, electrical, and so on) use simulation tools that present graphical interfaces for linking system components, then solve equations generated from the graphical models, and report predicted values of system properties over time. Linked components interact physically (mechanically, electrically, and so on) or send numeric signals to each other (see Subclause 6.1 for the difference between physical interaction and signal flow). The tools generate (ordinary and algebraic) differential equations to describe the evolution of numeric system properties over time, and solve them to predict system behavior. These models are sometimes known as lumped parameter or 1-D models, but this specification refers to them as physical interaction and signal flow, to emphasize their applications (or just simulation models for brevity). This kind of simulation is specified without regard to physical distances between or within components, as compared to distributed simulation models (as in finite element analysis), in which behavior specifications account for physical distances between or within components. See Subclause 6.1 for more information about this kind of simulation.

Graphical interfaces presented by physical interaction and signal flow simulators express concepts similar to the Systems Modeling Language (SysML), an extension of the Unified Modeling Language (UML). Both languages show system components, how components are connected together, and how physical substances and information flow between components. SysML and these simulators both have underlying textual languages to record models in computer-processable file formats. Simulators translate models specified through graphical interfaces into file-based formats, which are then transformed into equations for solution by numerical analysis. SysML-based tools use their filed-based formats to perform other kinds of analysis and verification, checking completeness of designs against requirements.

When SysML tools and physical interaction and signal flow simulators are used separately, simulation engineers must re-specify their systems in each tool they are using, including information that is also available in SysML models. This additional effort would not be necessary if the information to perform this kind of simulation were available in SysML and translations were defined between SysML and simulation languages.

This specification:

- Extends SysML with additional information needed to perform physical interaction and signal flow simulation independently of simulation platforms.
- Provides a human-usable textual syntax for mathematical expressions.
- Includes a platform-independent SysML library of simulation elements that can be reused in system models.
- Gives translations between SysML as extended above and two widely-used simulation languages and tools for physical interaction and signal flow simulation.

With the extension, expression language, libraries, and translations above, information in common between SysML and simulation languages only needs to be specified once in SysML and translated to simulators, rather than manually recoded for each simulation language and tool. The library enables SysML models for simulation to be built more quickly by reusing library elements rather than reconstructing them for each application. Taken together, these capabilities provide a basis for more efficient integration of SysML models and processes with those of physical interaction and signal flow simulation.

2 Conformance

A tool demonstrating conformance to this specification must satisfy at least one of these points:

- *Abstract syntax conformance*. Tools demonstrating abstract syntax conformance provide user interfaces and/or APIs that enable
    - instances of concrete stereotypes defined in this specification (which are applications of stereotypes to instances of UML metaclasses) to be created, read, updated, and deleted, including links and references from these to instances of UML elements and instances of SysML stereotypes.
    - bodies and languages of opaque expressions and opaque behaviors to be created, read, updated, and deleted conforming to the mathematical expression language defined in this specification.
    - links and references to model library elements defined in this specification to be created and deleted.

  The tools also provide a way to validate the well-formedness of the above as defined by stereotypes, grammars, and model library elements in this specification.
- *Concrete syntax conformance*. Tool demonstrating concrete syntax conformance provide user interfaces and/or APIs that enable the mathematical expression language defined in this specification and the SysML notation for the abstract syntax above to be created, read, updated, and deleted. See the SysML specification for more about SysML notation conformance.
- *Model interchange conformance*. Tools demonstrating model interchange conformance can import and export conformant XMI for all models that are valid under this specification. Model interchange conformance implies abstract syntax conformance.
- *Translation conformance*: Tools demonstrating translation conformance can translate between extended SysML and simulation models per this specification, either in one direction or both directions.

3 References

3.1 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.

[1] Object Management Group, “OMG Unified Modeling Language, version 2.5,” http://www.omg.org/spec/UML/2.5, March 2015.

[2] Object Management Group, “OMG Systems Modeling Language, version 1.4,” http://www.omg.org/spec/SysML/1.4, September 2015.

[3] Modelica Association, “Modelica® - A Unified Object-Oriented Language for Systems Modeling, Language Specification, version 3.3, revision 1,” https://www.modelica.org/documents/ ModelicaSpec33Revision1.pdf, July 2014.

[4] Modelica Association, “Modelica Standard Library,” https://github.com/modelica/Modelica, 2015.

[5] International Standards Organization, “Information technology – Syntactic metalanguage – Extended BNF,” http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_ IEC_14977_1996(E).zip, 1966.

3.2 Non-normative References

[1] Kecman, V., State-Space Models of Lumped and Distributed Systems, Springer-Verlag, 1988.

[2] Cellier, F., Elmqvist, H., Otter, M., “Modeling from Physical Principles,” in Levine, W., Control System Fundamentals, pp. 99-108, CRC Press, 1999.

[3] Raven, F., Automatic Control Engineering (Fifth Edition), McGraw-Hill, January 1995.

[4] The MathWorks, Inc., “Simulink® User’s Guide,” http://www.mathworks.com/help/pdf_doc/simulink/ sl_using.pdf, 2016.

[5] The MathWorks, Inc., “Simulink® Reference,” http://www.mathworks.com/help/pdf_doc/simulink/slref.pdf, 2016.

[6] The MathWorks, Inc., “Simscape™ Language Guide,” http://se.mathworks.com/help/pdf_doc/ physmod/simscape/simscape_lang.pdf, 2016.

[7] The MathWorks, Inc., “MATLAB® Primer,” http://www.mathworks.com/help/pdf_doc/matlab/ getstart.pdf, 2015.

[8] The MathWorks, Inc., “StateFlow® User Guide,” http://www.mathworks.com/help/pdf_doc/stateflow/ sf_ug.pdf, 2015.

4 Terms and Definitions

For the purposes of this specification, the term ‘simulation’ will refer to physical interaction and signal flow simulation, unless qualified. See Clause 1 for more information about this kind of simulation.

Stereotype names are sometimes used in place of instances of their base classes to which the stereotypes are applied. For example, the phrase ‘a property typed by a SimBlock’ refers to a property typed by an instance of Class, where the instance has the SimBlock stereotype applied.

5 Symbols

No symbols are introduced by this specification.

6 Additional Information

6.1 Signal flow and physical interaction simulation compared

The differences between physical interaction and signal flow and lie mainly in how components interact, addressing different kinds of problems:

- In signal flow modeling, system components exchange numeric and Boolean values in predetermined directions. For each component, some values will be provided by other components (inputs), and some values will be provided to other components (outputs). Component behavior is specified with assignments and algorithmic control statements. Connections between components indicate that values are passed from one output of a source component to one or more inputs of target components. Component behavior is specified by assigning values to outputs, based on values of inputs and other component variables. Signal flow is better suited for describing control systems and signal-processing systems. It is also used to define interconnected mathematical equations, although physical interaction might be more suited to represent

some of these systems.

- In physical interaction, system components exchange (possibly abstract) conserved physical substances that carry energy. Each exchange is characterized by two numerical values (flow rate and potential to flow of a physical substance), compared to one for signal flow, which does not involve physical substances. In physical interaction, the direction in which substances (and their numerical values) flow between components is not predetermined, as it is for numerical values in signal flow. Component behavior in physical interaction is specified by equations, as opposed to assignments as in signal flow, to indicate that inputs and outputs are unknown at the time of modeling. The direction in which substances flow between components is determined during simulation, and can change during simulation. Physical interaction is well suited for representing systems with components that exchange conserved substances, but is more cumbersome for using of algorithmic control statements.

In practice, physical interaction and signal flow are often combined in a same model. For example, many systems have physical components directed by control systems via sensors and actuators.

6.2 How to read this specification

Clauses 1 to 6 contain background and basics for reading this specification. Clause 1 describes the objectives of this specification and the intended readership. Clause 2 defines conformance. Clause 3 lists other specifications and documents containing provisions which, through reference in this text, constitute provisions of this specification.

Clause 4 and 5 contains definitions of terms, abbreviations, and symbols used in this document. Clause 6 provides additional information to this specification.

Clauses 7 to 11 are the technical part of this specification. Clause 7 defines a SysML extension for physical interaction and signal flow simulation. Clause 8 defines a language to be used for expressions representing equations and algorithmic statements. Clause 9 defines processing of SysML models that can be performed prior to translation to simulation platforms. Clause 10 provides translations between SysML as extended in Clause 7 and two simulation platforms, Modelica and Simulink (including extensions to Simulink, such as Simscape). Clause 11 defines a platform-independent simulation library in SysML, with components corresponding to platform-dependent library components. Clause 11.4 gives additional examples showing how to use the contents of Clauses 7, 8, and 11.

6.3 Changes to Adopted OMG Specifications

None.

6.4 Acknowledgments

The following companies submitted this specification:

- No Magic, Inc.

The following companies and organizations support this specification:

- U.S. National Institute of Standards and Technology
- Office of the Secretary of Defense
- InterCax, LLC
- ModelFoundry Pty. Ltd.
- ModelAlchemy Consulting
- XPLM Solution GmbH
- Koneksys, LLC
- oose Innovative Informatik GmbH

7 SysML extension for physical interaction and signal flow simulation

7.1 Introduction

This clause defines a SysML extension for physical interaction and signal flow. It reflects features common to various physical interaction and signal flow platforms that are not present in SysML. This clause summarizes the extension. More information is given in Subclauses 10.5 and 10.6.

7.2 Simulation profile

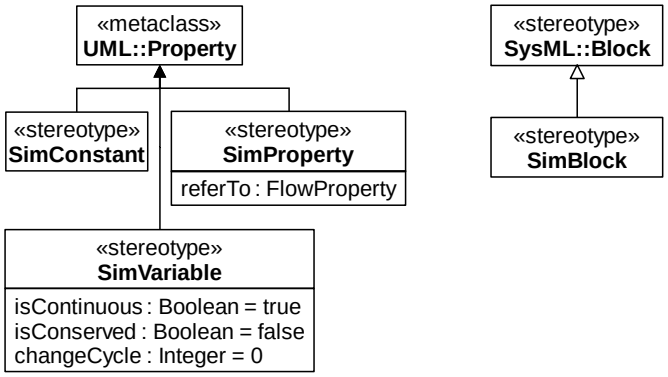

Figure 1: Simulation stereotypes

7.2.1 SimBlock

Package: SimulationProfile
isAbstract: No
Generalization: Block

Description

A SimBlock contains the characteristics of either a type of conserved physical substance, or a type of signal. A SimBlock contains only SimVariables, which can be conserved (for flow rate) or non-conserved (for potential to flow).

Constraints

[1] All owned properties must be stereotyped by SimVariable.

7.2.2 SimConstant

Package: SimulationProfile
isAbstract: No
Extended Metaclass: Property

Description

A SimConstant has values that do not change during simulation. Values can change between simulation.

Constraints

[1] A property stereotyped by SimConstant must not be stereotyped by SimVariable.

Notation

A compartment with the label ‘sim constants’ may appear as part of a block definition to list the properties stereotyped by SimConstant. The properties omit the ‘«simConstant»’ prefix.

7.2.3 SimProperty

Package: SimulationProfile
isAbstract: No
Extended Metaclass: Property

Description

A SimProperty defines a relationship between a flow property and the characteristics of a conserved physical substance or signal flowing through that property. The flow property is identified by the referTo attribute, and characteristics of the signal or conserved physical substance are given by the SimProperty’s type, which must be a SimBlock (for physical interaction) or Real, Integer, or Boolean (for signal flow).

Attributes

referTo: FlowProperty	Identifies a flow property for flows characterized by the type of the SimProperty.

Constraints

[1] The type must be Real, Integer, Boolean or a Class stereotyped by SimBlock.
[2] The value of referTo must be a FlowProperty.
[3] When the type is Real, Integer, Boolean the property referred to must have flow direction in or out, not inout.

Notation

A compartment with the label ‘sim properties’ may appear as part of a block definition to list the properties stereotyped by SimProperty. The properties omit the ‘«simProperty»‘ prefix.

7.2.4 SimVariable

Package: SimulationProfile
isAbstract: No
Extended Metaclass: Property

Description

A SimVariable has values that can vary over time in a continuous or discrete fashion. When flow properties are defined on ports linked to other ports by connectors, the values of conserved SimVariables owned by SimBlocks that refer to the flow properties add up to zero (the values have opposite signs). The change cycle for continuous SimVariables is zero. Conserved SimVariables are for physical interactions, while non-conserved SimVariables are for signal flow.

Attributes

isContinuous: Boolean = true	Determines whether the property value varies continuously or discretely.
isConserved: Boolean = false	Determines whether values of the property value are conserved or not.
changeCycle: Real = 0	Specifies the time interval at which a discrete property value changes.

Constraints

[1] The type must be Real, Integer, or Boolean.
[2] isConserved can be true only when the stereotyped property is owned by SimBlock
[3] changeCycle can be set to a value other than 0 only when isContinuous = false.
[4] The value of changeCycle must be positive or equal to 0.
[5] A property stereotyped by SimVariable must not be stereotyped by SimConstant.

Notation

A compartment with the label 'sim variables' may appear as part of a block definition to list the properties stereotyped by SimVariable. The properties omit the '«simVariable»' prefix.

8 Language for mathematical expressions

This clause describes a platform-independent textual language for mathematical expressions. The language is for use in the bodies of:

- OpaqueExpressions of constraints, corresponding to equations.
- OpaqueBehaviors, corresponding to algorithmic statements.

OpaqueExpressions and OpaqueBehaviors that use this language in their body should have an associated 'SysPISF' string as their language.

The grammar of the SysPISF grammar includes a subset of Modelica's grammar, as follows:

- All terminal symbols (*IDENT*, *Q-IDENT*, *Q-CHAR*, *S-ESCAPE*, *S-CHAR*, *DIGIT*, *UNSIGNED_INTEGER*, *UNSIGNED_NUMBER*)
- The following non-terminal symbols: *equation*, *statement*, *if_equation*, *if_statement*, *for_statement*, *for_indices*, *for_index*, *while_statement*, *expression*, *simple_expression*, *logical_expression*, *logical_term*, *logical_factor*, *relation*, *rel_op*, *arithmetic_expression*, *add_op*, *term*, *mul_op*, *factor*, *primary*, *name*, *component_reference*, *function_call_args*, *function_arguments*, *named_arguments*, *named_argument*, *function_argument*, *output_expression_list*, *expression_list*, *array_subscripts*, *subscript*

Symbols in the Modelica grammar not listed above are not included in the SysPISF grammar. The semantics of the above symbols is given in Modelica (which is the same in MATLAB, the expression language in Simulink, Simscape, and StateFlow, assuming the translations in Subclause 10.12).

The following non-terminal symbol is included in the SysPISF grammar to specify execution of a series of statements (expressed in extended BNF):

```
statements : { statement ";" }
```

When used in OpaqueExpressions, the root non-terminal symbol must be `equation`. When used in OpaqueBehaviors, the root non-terminal symbol must be `statements`.

The following are functions available in SysPISF expressions language: *abs*, *sign*, *sqrt*, *div*, *mod*, *rem*, *ceil*, *floor*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *atan2*, *sinh*, *cosh*, *tanh*, *log*, *log10*, *exp*, *der*. The semantics of these functions is given in Modelica (which is the same in MATLAB, assuming the translations in Subclause 10.12).

9 Pre-processing SysML models

9.1 Introduction

This clause defines processing of SysML models performed prior to translation to simulation models per Clause 10, to:

- Enable translations of SysML modeling patterns not covered in Clause 10 (association blocks, Subclause 9.2).
- Accommodate longer lifecycle development in SysML while simplifying simulation models in the short term (physical modeling of signal flow, Subclause 9.3).

Pre-processing should be done on copies of SysML models, because processing changes these models, while SysML modelers continue using the original ones.

9.2 Association blocks

9.2.1 Purpose

Many physical phenomena occur due to the relationship between two system components. For example, friction occurs when two pieces in contact move relative to each other and produce heat. SysML supports a modeling technique for complex relationships (association blocks) that is not available in simulation models. SysML models using this technique are processed into simpler models suitable for the translations to simulation platforms in Clause 10.

9.2.2 SysML model before processing

SysML association blocks are both associations and blocks. They represent relationships between two blocks, like associations, and can have structural features, like blocks. Figure 2 shows an example of association block in SysML.

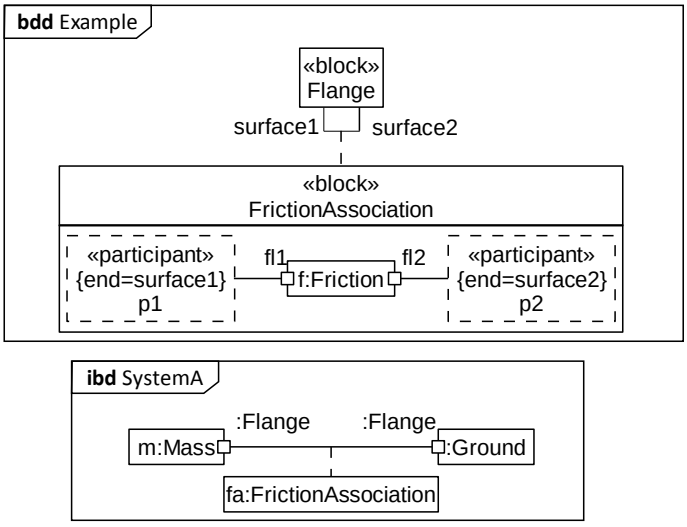

Figure 2: Association blocks and connector properties in SysML

The upper part of Figure 2 shows an association block *FrictionAssociation* relating *Flanges* (defined in Subclause 10.6.7). The internal structure of *FrictionAssociation* shows a part typed by *Friction* with two ports, each being connected to a participant of the association. The lower part of Figure 2 shows a connector between the flange of a mass and the flange of a ground. The connector has an associated connector property typed by *FrictionAssociation*.

9.2.3 SysML model after processing

SysML connector properties are replaced by the internal structure of their types (association blocks).

Figure 3 shows the content of Figure 2 after processing. The connector property *fa* in Figure 2 has been replaced by the content of the association block *FrictionAssociation* (the connector property and association block are removed). The flange of the mass and the flange of the ground replace the participant properties of the association block and are connected to the property *f* of type *Friction* in the same way as in the association block.

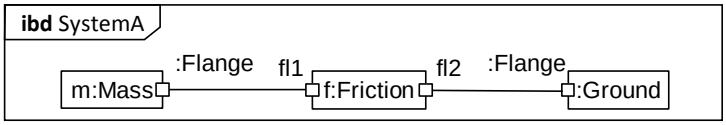

Figure 3: Replacement of connector properties

9.3 Signal flow using SimBlocks

9.3.1 Purpose

The type of signals in signal flow modeling is given by the type of SimProperties (see Subclause 7.2.3). As an alternative, SimProperties can be typed by SimBlocks that have a single SimVariable giving the signal type. This is useful when conserved physical substances carrying signals will be specified later in model development, by adding flow property types and flow rate SimVariables. SysML models using this technique can be processed into the simpler models of Subclause 10.6.3 for translation to simulation platforms.

9.3.2 SysML model before processing

Figure 4 shows an example of the alternative for signal flow modeling. It has a block *Spring* with two ports *u* and *y*, of type *SigPin* and *~SigPin*, respectively. *SigPin* has an in flow property, and a SimProperty referring to it typed by *SignalFlow*. *SignalFlow* has a continuous SimVariable *s*.

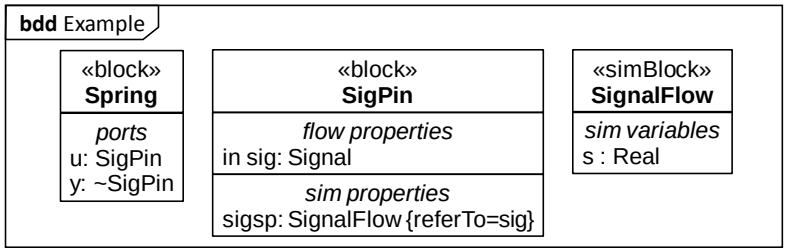

Figure 4: Signal definition using a SimBlock

9.3.3 SysML model after processing

The type of the SimProperty is replaced by the type of the SimVariable in the SimBlock. The SimBlock can be deleted if it is not used elsewhere. Figure 5 shows the content of Figure 4 after processing.

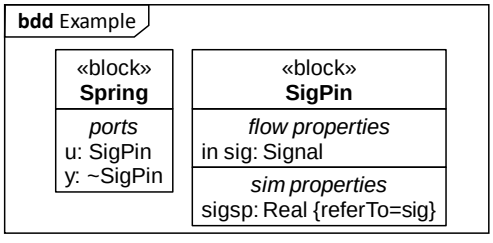

Figure 5: Replacement of SimProperty type

10 Translating between SysML and simulation platforms

10.1 Introduction

This clause shows how to translate between SysML models extended as in Clause 7 (hereafter referred to as SysML) and models in multiple simulation platforms. Translations are given as correspondences between patterns of using SysML and simulation platforms, enabling translation in either direction. However, simulation platforms have more specific purposes than SysML, resulting in loss of information when translating from SysML to simulation platforms.

The selected platforms are Modelica and Simulink, including extensions of Simulink, such as Simscape. The modeling concepts covered by these translations are available in both simulation languages.

- Modelica is a textual simulation language for physical interaction and signal flow modeling supported by various simulation tools, such as OpenModelica, Dymola®, and MapleSim® that add graphical interfaces and numerical solvers. Modelica is defined by a grammar, but does not have a metamodel. As a result, the terms used to describe Modelica models correspond to keywords defined in its grammar.

- Simulink is a graphical simulation tool for signal flow modeling (unless extended, see below). Its modeling concepts can be inferred from the simulation files generated from graphical models (no metamodel or textual language has been released for Simulink). Two file formats are currently used: the older punctuated textual format, or the newer XML format. The concepts used in these two formats are the same, but the structure and the way values are represented differ. Simulink supports S-Functions to represent system behaviors as MATLAB files (generally behavior in state-space form). S-Functions always follow the same structure and use the same concepts.

Simulink includes extensions for other aspects of systems modeling:

- Simscape is the extension of Simulink for physical interaction modeling. Physical components specifications are persisted in a file that must conform to the Simscape grammar. Simscape concepts are named in the grammar.

- Stateflow® is the Simulink extension for state machines. It uses additional concepts represented along with Simulink elements.

Subclauses 10.2 through 10.11 are divided into these parts:

- *Purpose*: Explains the particular kinds of information in system or simulation modeling covered by the subclause.

- *SysML modeling*: Describes how the above information is modeled in SysML, extended as in Clause 7 when necessary, along with a small example.

- *Simulation platform modeling*: Describes the correspondence between the portions of SysML used as above and modeling patterns in simulation platforms, along with simulation models corresponding to the SysML example above.

- *Summary*: Summarizes the correspondences between SysML and simulation platforms in a table.

Subclause 10.12 covers translations for the expression language in Clause 8.

10.2 Blocks and properties

10.2.1 Purpose

Systems and simulation models contain classes describing systems and components that share the same features. Systems and components function (play roles) within others, which are described in models as the usage of one class by another. For example, a class for cars might have a power source reusing a class for engines.

10.2.2 SysML modeling

Modeling in SysML is based on blocks, which are classes of systems or components, describing objects that share the same features. These features can be structural or behavioral.

Structural features of blocks are called properties, some of which are for values, such as numbers or strings of characters, and some of which are usages of other blocks. This difference is indicated by typing a property by a data type or by a block. Some system properties typed by blocks are parts, corresponding to usages of those block within a system or component.

Figure 6 shows a SysML block *A* that contain one part *b1* of type *B*. *B* is also a SysML block.

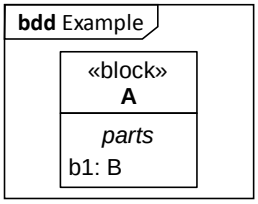

Figure 6: Block and part in SysML

10.2.3 Modelica modeling

Modelica is a textual language for physical interaction and signal flow modeling. It defines syntax and semantics that can be used to specify system simulations. Modelica is a class-oriented language, defining structural and behavioral features, like SysML, but with different terminology. Structural features in Modelica are called components. The simulation process starts by instantiating classes, then performing numerical analysis to compute values of all data components. Modelica includes various kinds of classes (model, record, block, connector, type, package, operator), some of which have restrictions and/or enhancements.

SysML blocks correspond to Modelica models, the most general kind of Modelica class and the most commonly used. SysML properties correspond to Modelica components, including the name and type.

The following Modelica example corresponds to the SysML block *A* in Figure 6.

```
model A
  B b1;
end A;
model B
end B
```

The Modelica model *A* is defined, and it has a component named *b1* of type *B*, which is also a Modelica model.

10.2.4 Simulink modeling

The structure of Simulink models differs from the structure of SysML and Modelica. Simulink has a concept similar to SysML blocks (and of the same name), but it can be used either as a container of structural features (subsystems),

as a reference to user-defined blocks, or as a reference to blocks from a library. When used as a container, structural features are actually contained in a System. Systems can be owned by models or libraries.

A SysML block and its parts correspond a Simulink block with a system containing blocks referencing other blocks.

SysML blocks that do not have constraint properties correspond to Simulink subsystem blocks. SysML blocks with constraint properties correspond to either Simulink subsystem blocks (when Simscape is not used, signal flow), or to Simscape components (when Simscape is used, physical interaction).

The following example shows Simulink code corresponding to Figure 6. It has a Simulink subsystem block *A* corresponding to the SysML block *A*, with a system that contains a reference to the Simulink block *B*.

```
<Block BlockType="SubSystem" Name="A" SID="1">
  <System>
    <Block BlockType="Reference" Name="b1" SID="2">
      <P Name="Ports">[0,0]</P>
      <P Name="SourceBlock">LibraryFile/B</P>
    </Block>
  </System>
</Block>


<Block BlockType="SubSystem" Name="B" SID="3">
  <System>
  </System>
</Block>
```

10.2.5 Simulink/Simscape modeling

Simscape is an extension of Simulink for physical interaction modeling. SysML blocks with constraint properties or binding connectors correspond to Simscape components.

The following example shows Simscape code corresponding to block *B* Figure 6, assuming the block has constraint parameters.

```
component B
end
```

The following Simulink code corresponds to block *A* in Figure 6. It has a subsystem block *A*, with a system that contains a reference *b* to the Simscape component *B* from the package *Library*.

```
<Block BlockType="SubSystem" Name="A" SID="1">
  <System>
    <Block BlockType="Reference" Name="b" SID="2">
      <P Name="SourceBlock">Library/B</P>
      <P Name="SourceType">B</P>
      <P Name="SourceFile">Library.B</P>
      <P Name="ComponentPath">Library.B</P>
      <P Name="ClassName">B</P>
    </Block>
  </System>
</Block>
```

10.2.6 Simscape modeling

SysML parts correspond to Simscape member components.

The following example shows Simscape code corresponding to block *A* in Figure 6. It has a component *A* containing a member component *b1* of type *B* (from the package *Library*).

```
component A
  components
    b1=Library.B;
  end
end
```

10.2.7 Summary

SysML	Modelica	Simulink	Simscape
Block with no constraint properties and no binding connector	Model	SubSystem block with system	N/A
Block with constraint properties or binding connectors	Model	SubSystem block with system	Component
Block name	Model name	SubSystem name	Component name
Property typed by a block, owned by block	Component owned by model	Reference block, owned by system	Member component
Property name	Component name	Reference block name	Member component name
Property type	Component type	Reference block source	Member component type

Restrictions

- SysML properties must be typed

10.3 Root element

10.3.1 Purpose

Systems and simulation models are organized in a structured way starting with root elements.

10.3.2 SysML modeling

SysML root elements are packages, which are containers for model elements.

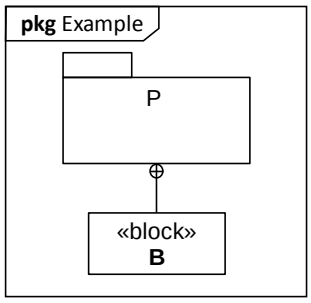

Figure 7: Package and model in SysML

Figure 7 shows a package *P* owning a block *B*.

10.3.3 Modelica modeling

A Modelica file can contain any type of class as root, including a model. SysML packages correspond to Modelica models. The following Modelica code corresponds to Figure 7. It has a model *P* owning a model *B*.

```
model P
  model B
  end B;
end P;
```

10.3.4 Simulink modeling

A Simulink file can contain a model or a library as root. Simulink blocks defined in a model cannot be reused, whereas blocks defined in a library can. A model is then used to contain references to reusable blocks. Both a library and a model are needed to be equivalent to a SysML package.

The following Simulink code corresponds to Figure 7. It has a library *P* and a model *M*, each owning a system with a block *B*.

```
<Library>
  <System>
    <Block name="B">
    …
    </Block>
  </System>
</Library>

<Model>
  <System>
    <Block name="B">
    </Block>
  </System>
</Model>
```

10.3.5 Summary

SysML	Modelica	Simulink
Package	Model	Library+System, Model+system
Object owned by package	Object owned by package	Object owned by system

10.4 Generalization

10.4.1 Purpose

Generalization simplifies systems and simulation modeling by enabling features of one class to be reused by (inherited to) another class.

10.4.2 SysML modeling

SysML provides a generalization relationship to indicate that one block reuses the features of another. A block generalized by another block will inherit all the properties of that other block. SysML supports multiple generalizations of the same block, but not all the simulation languages and tools do.

Figure 8 shows a block *A* with a property *c1* of type *C*, and a block *B* that is a specialization of that block *A*.

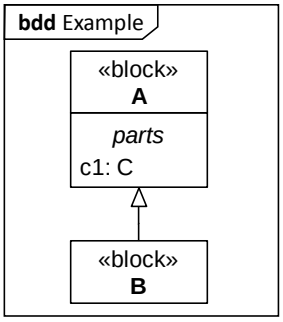

Figure 8: Generalization in SysML

10.4.3 Modelica modeling

A Modelica class can be generalized by another Modelica class. This is done through an "extend" clause. There can be several "extend" clauses, for multiple generalizations of the same class. Modelica components inherited from more general classes can be redefined.

The following Modelica code corresponds to Figure 8. It has a model *A* with a component *c1* of type *C*, and a model *B* that extends *A*. As a result, *B* inherits the component *c1* from *A*.

```
model A
  C c1;
end A;

model B
  extends A;
end B;
```

10.4.4 Simulink modeling

Simulink does not support generalization. Simulink blocks cannot inherit features from another blocks. It is possible for a system to use another block and add to its behavior, but this does not necessarily correspond to generalization. As a result, features that are inherited in SysML must be explicitly added to all Simulink blocks that are generalized in SysML.

The following Simulink code corresponds to Figure 8. The blocks *A* and *B* are defined, and the system contain a block *c1* that is a reference to the block *C* from the library *LibraryFile*. There is no generalization between *A* and *B*.

```
<Block BlockType="SubSystem" Name="A" SID="1">
  <System>
    <Block BlockType="Reference" Name="c1" SID="2">
      <P Name="Ports">[0,0]</P>
      <P Name="SourceBlock">LibraryFile/C</P>
    </Block>
  </System>
</Block>
<Block BlockType="SubSystem" Name="B" SID="3">
  <System>
    <Block BlockType="Reference" Name="c1" SID="4">
      <P Name="Ports">[0,0]</P>
      <P Name="SourceBlock">LibraryFile/C</P>
    </Block>
  </System>
</Block>
```

10.4.5 Simscape modeling

Simscape supports generalization of components, but not multiple generalizations of the same component. Component members and equations are available to other components by generalization.

The following Simscape code corresponds to Figure 8. The component *A* has a node *c1* typed by *C* (from the package *CurrentLibrary*), and the component *B* is generalized by *A* (from the package *CurrentLibrary*).

```
component A
  nodes
    c1 = CurrentLibrary.C;
  end
end

component B < CurrentLibrary.A
end
```

10.4.6 Summary

SysML	Modelica	Simulink	Simscape
Generalization	Extend clause	N/A	Subclassing
Inherited features	Not translated	Translated	Not translated

Restrictions

- SysML blocks must not have multiple generalizations

10.5 SimVariables and SimConstants

10.5.1 Purpose

Simulation modeling specifies how variable values can change during simulation, particularly for numeric values, whereas system models do not. Simulation modeling distinguishes numeric variables with values that can change continuously (possible infinitesimally) over time from those that always change discretely (finitely), possibly only at regular intervals. It also identifies variables with values that can only change between simulations (constants), rather than during simulation.

10.5.2 SysML modeling

The simulation extension in Subclause 7.2 distinguishes properties as described above. Continuous SysML properties are stereotyped by SimVariable, with isContinuous=true. Discrete properties are stereotyped by SimVariable, with isContinuous=false. Constant properties are stereotyped by SimConstant.

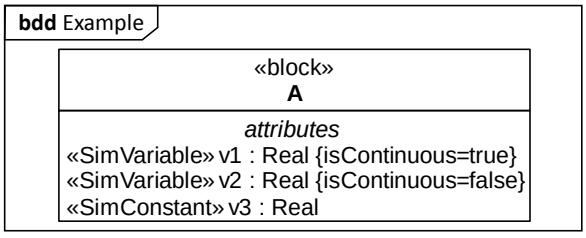

Figure 9: SimVariables and SimConstant in SysML

Figure 9 shows a block *A* with three properties: one continuous SimVariable *v1*, one discrete SimVariable *v2*, and one SimConstant *v3*.

Note: SysML notation for stereotype properties can omit a property if the default value is used. For example, isContinuous is true by default, and can be omitted from the notation for variables that are continuous.

10.5.3 Modelica modeling

The variability of Modelica properties are of four kinds: continuous, discrete, parameter, and constant. By default, Modelica properties are continuous. SimVariables with isContinuous=true correspond to continuous components, SimVariables with isContinuous=false correspond to discrete components, and SimConstants correspond to parameter variables.

The following Modelica code corresponds to Figure 9. It has a model A, with three properties *v1*, *v2* and *v3* of type Real, that are continuous, discrete, and parameter, respectively.

```
model A
  Real v1;
  discrete Real v2;
  parameter Real v3 = “...”;
end A
```

10.5.4 Simulink modeling

See Subclause 10.8 for Simulink corresponding to SysML value properties in the context of SysML constraint blocks and binding connectors.

10.5.5 Simscape modeling

Data properties in Simscape can either be (continuous) variables or (constant) parameters. Discrete variables are not supported. SimVariables with isContinuous=true correspond to Simscape variables, and SimConstants correspond to parameters.

The following Simscape code corresponds to Figure 9. It has a component *A* with one variable *v1*, and one parameter *v3*. The variable *v1* is continuous.

```
component A
  variables
    v1 = 1;
  end
  parameters
    v3 = 10;
  end
end
```

10.5.6 Summary

SysML	Modelica	Simulink	Simscape
Property stereotyped by SimVariable, with isContinuous=true	Continuous component	N/A	Variable
Property stereotyped by SimVariable, with isContinuous=false	Discrete component	N/A	N/A
Property stereotyped by SimConstant	Parameter component	N/A	Parameter
Property type (DataType)	Component type (Type)	N/A	Member type (DataType)

Restrictions

- SysML properties must be typed

10.6 Ports, FlowProperties, SimProperties, and SimBlocks

10.6.1 Purpose

Systems and simulation modeling describe interactions between system components. These interactions include exchanges of conserved physical substances, signals, or both. System and simulation components include structural features used as connection points to other components. System and simulation models include connections between these points when the components are used. System models specify the kind of things exchanged between connection points, while simulation models give characteristics of these exchanges, in particular the rate of flow and potential to flow.

10.6.2 SysML modeling

In SysML, interactions between parts are modeled using connectors. Connections are often between ports of these parts. Ports are properties used as connection points to other blocks.

The type of a port describes individual flows through the port using flow properties, which have flow directions (in/out/inout).

The extension for simulation in Subclause 7.2 adds information needed for simulation that is not available in SysML, in particular, flow rates and potentials to flow. Physical interaction uses both of these, while signal flow only uses potential to flow. These are given by attributes of SimVariables (non-conserved and coniserved, respectively). SimVariables are linked to flow properties by adding SimVariables to SimBlocks, then typing SimProperties by those SimBlocks, and linking those SimProperties to flow properties via the referTo attribute of SimProperty. To simplify signal flow modeling, which only uses potentials to flow, SimVariables and SimBlocks can be omitted, and signal types (Real, Integer, or Boolean) given by the type of SimProperties.

Subclauses 10.6.3 through 10.6.6 cover signal flow modeling in SysML and simulation platforms, while subclauses 10.6.7 through 10.6.10 cover physical interaction modeling.

10.6.3 SysML modeling, signal flow

When modeling signal flow, flow properties of the port type must be either in or out. A flow property in the opposite direction can be obtained by conjugating the ports. The SimProperty referring to this flow property must be typed by Real, Integer, or Boolean.

Figure 10 shows an example signal flow application. The block *Spring* has two ports *u* and *y*, of type *SigPin* (*v* is conjugated). *SigPin* has an *in* flow property *sig* and a SimProperty *sig* referring to that flow property. The SimProperty is typed by Real.

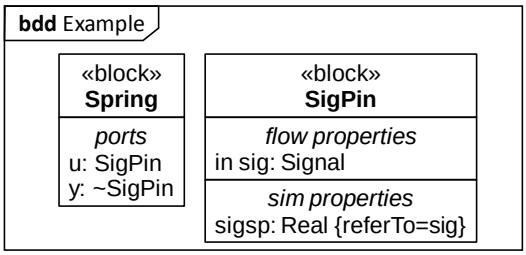

Figure 10: Ports for signal flow in SysML

See Subclause 9.3 for an alternative way for modeling signal flow in SysML that is useful when conserved physical substances carrying signals will be specified later in model development.

10.6.4 Modelica modeling, signal flow

SysML ports with a type containing a flow property referred to by a SimProperty typed by Real, Integer, or Boolean correspond to Modelica components typed by that data type. The Modelica component has a direction given by the flow property, accounting for conjugation of the port, if any.

The following Modelica code corresponds to Figure 10. It has a model *Spring*, with two components *u* and *y* of type *Real* and of direction respectively in and out.

```
model Spring
  in Real u;
  out Real y;
end Spring;
```

10.6.5 Simulink modeling, signal flow

SysML ports with a type containing a flow property referred to by a SimProperty typed by a Real, Integer, or Boolean correspond to Simulink inports or outports. The choice is made depending on the direction of the flow property, accounting for conjugation of the port, if any.

The following Simulink code corresponds to Figure 10. It has a block *Spring*, with one inport *u* and one outport *y*. Note the Ports property of the block, which counts the number of inports (1st position) and outports (2nd position).

```
<Block BlockType="SubSystem" Name="Spring" SID="1">
  <P Name="Ports">[1,1]</P>
  <System>
    <Block BlockType="Inport" Name="u" SID="2">
      <P Name="Port">1</P>
      </Block>
    <Block BlockType="Outport" Name="y" SID="3">
      <P Name="Port">1</P>
    </Block>
  </System>
</Block>
```

10.6.6 Simscape modeling, signal flow

The following Simscape code corresponds to Figure 10. It has a component *Spring*, with one input *u* (displayed on the left side of the block) and one output *y* (displayed on the right side of the block).

```
component Spring
inputs
  u = {0, 'unit'}; % :left
end

outputs
  y = {0, 'unit'}; % :right
end
end
```

10.6.7 SysML modeling, physical interaction

When modeling for physical interaction, flow properties of the port type must be *inout*. The SimProperty referring to this flow property must be typed by a SimBlock, which contains conserved and non-conserved SimVariables (the same number of each).

Figure 7 shows an example physical interaction application. The block *Spring* has two ports *p1* and *p2*, of type *Flange*. *Flange* has an inout flow property *mo* and a SimProperty *me*, referring to that flow property. The SimProperty is typed by the SimBlock *MomentumFlow*, which has one conserved SimVariable *f* and one non-conserved SimVariable *v*.

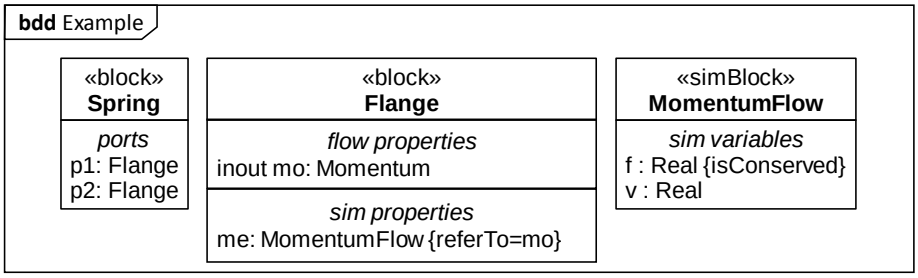

Figure 11: Ports for physical interaction in SysML

10.6.8 Modelica modeling, physical interaction

In physical interaction modeling, SysML ports correspond to Modelica components, and SysML port types correspond to Modelica connectors. SysML flow properties have no corresponding constructs in simulation platforms. SimProperties and SimBlocks do not either, but are used to identify SimVariables, which correspond to variables in simulation platforms. SimVariables from SimBlocks typing SimProperties correspond to Modelica connectors. Conserved SimVariables correspond to Modelica flow components. Modelica components corresponding to SysML ports have no direction, indicating that information can go in either direction. The direction is determined during simulation, rather than when components are defined.

The following Modelica code corresponds to Figure 11. It has a model *Spring*, with two components *p1* and *p2* of type *Flange*. *Flange* is a connector that has one flow component *f*, and one regular component *v*.

```
model Spring
  Flange p1;
  Flange p2;
end Spring;
connector Flange
  flow Real f;
  Real v;
end Flange;
```

10.6.9 Simulink modeling, physical interaction

Simulink supports connection ports for representing bidirectional flows, but exchanges between them are defined in Simscape (see Subclause 10.6.10). Simulink connection ports must be connected to Simscape nodes.

10.6.10 Simscape modeling, physical interaction

Simscape adds support for physical ports to Simulink. These ports can receive an input signal, an output signal, or an inout signal. They can either be defined directly in Simulink (called connection ports) or they can be defined in Simscape (called nodes). Nodes are typed by a domain, which corresponds to a SysML port type with an inout flowProperty and SimProperties. Conserved SimVariables in SimBlocks correspond to Modelica balancing variables in the domain.

The following Simscape code corresponds to Figure 11. It has a component *Spring*, with two nodes *p1* and *p2* of type *Flange*. These two nodes will be displayed on the left side and right side of the block in Simulink, respectively. Flange is a domain from the package *CurrentLibrary*, with two variables: one non-balancing variable *v*, and one balancing variable *f*.

```
component Spring
  nodes
    p1 = CurrentLibrary.Flange; % :left
    p2 = CurrentLibrary.Flange; % :right
  end
end

domain Flange
  variables
    v = {0, 'm/s'};
  end
  variables(Balancing=true)
    f = {0, 'N'};
  end
end
```

10.6.11 Summary

SysML	Modelica	Simulink	Simscape
Port typed by block with an in flow property and a SimProperty typed by a DataType	component typed by a data type	inport	input variable
Port typed by block with an out flow property and a SimProperty typed by a DataType	component typed by a data type	outport	output variable
Port typed by block with an inout flow property and associated SimProperties	component typed by connector	Connector	Node typed by domain
Block with a flow property and an associated SimProperty typed by SimBlock	connector	N/A	domain
SimVariables owned by SimBlock	components owned by connector	N/A	variables owned by domain

Restrictions:

- Ports must be typed.

Note: Conjugation of a port reverses the direction of the flow properties in that port.

10.7 Connectors

10.7.1 Purpose

Once connection points for parts are defined, it is possible to specify connections between these points. A connection between two connection points belonging to different parts enables exchange of conserved physical substances or signals between these parts through their connection points.

10.7.2 SysML modeling

In SysML, connectors are used to link two ports belonging to two parts. These connections exist only in the context of the block that owns the connector, and other blocks generalized by it (connectors inherit).

Figure 12 shows an example of SysML connectors. It has two parts *s1* and *s2*, of type *Spring*. The block *Spring* has two ports, *p1* and *p2* of type *Flange*, as defined in Figure 11. The figure shows a connector between the port *p2* of *s1*, and the port *p1* of *s2*.

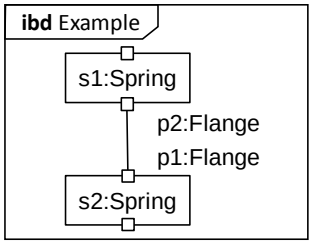

Figure 12: Connectors in SysML

10.7.3 Modelica modeling

SysML connectors correspond to Modelica connect equations, which link two components typed by Modelica connectors. This depends on the correspondence between SysML port types and Modelica connectors (see Subclause 10.6.8).

The following Modelica code corresponds to Figure 12. It has a model *Model* with two components *s1* and *s2*, of type *Spring*. *Spring* is a model with two components *p1* and *p2* of type *Flange* (see Subclause 10.6.8 for a definition). *Model* contains a connect equation linking the component *p2* of *s1* to the component *p1* of *s2*.

```
model Model
  Spring s1;
  Spring s2;
equation
  connect(s1.p2, s2.p1);
end Model;

model Spring
  Flange p1;
  Flange p2;
end Spring;
```

10.7.4 Simulink modeling, signal flow

The correspondence between SysML connectors and Simulink constructs depends on the kind of Simulink ports being connected. Connections between Simulink inports and Simulink outports are represented by regular Simulink lines. Simulink lines have a direction, which must be consistent with the kind of Simulink ports being connected (inports or outports). SysML connectors have no direction.

The following Simulink code corresponds to Figure 12. It has two blocks *s1* and *s2*, each referencing to the *Spring* block and having one inport and one outport (see Subclause 10.6.5). A line is defined between the outport port of *s1* (*p2*) and the inport of *s2* (*p1*).

```
<Block BlockType="Reference" Name="s1" SID="1">
  <P Name="Ports">[1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
</Block>
<Block BlockType="Reference" Name="s2" SID="2">
  <P Name="Ports">[1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
</Block>
<Line>
  <P Name="Src">1#out:1</P>
  <P Name="Dst">2#in:1</P>
</Line>
```

10.7.5 Simulink modeling, physical interaction

If two SysML blocks correspond to Simscape components, and the blocks are connected through bidirectional ports (have types with inout flow properties), SysML connectors correspond to a type of Simulink line called connections.

The following Simulink code correspond to Figure 12. It has two blocks *s1* and *s2* referring to Simscape component *Spring*. *Spring* has one left port (*p1*), and one right port (*p2*). A line of type "connection" connects the right port of *s1* to the left port of *s2*.

```
<Block BlockType="Reference" Name="s1" SID="1">
  <P Name="Ports">[0,0,0,0,0,1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
  <P Name="SourceType">Spring</P>
  <P Name="SourceFile">Library.Spring </P>
  <P Name="ComponentPath">Library.Spring </P>
  <P Name="ClassName">Spring</P>
</Block>
<Block BlockType="Reference" Name="s2" SID="2">
  <P Name="Ports">[0,0,0,0,0,1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
  <P Name="SourceType">Spring</P>
  <P Name="SourceFile">Library.Spring </P>
  <P Name="ComponentPath">Library.Spring </P>
  <P Name="ClassName">Spring</P>
</Block>
<Line LineType="Connection">
  <P Name="Src">1#rconn:1</P>
  <P Name="Dst">2#lconn:1</P>
</Line>
```

10.7.6 Simulink modeling, physical interaction and signal flow

When physical interaction and signal flow are combined in Simulink, a signal flow might be defined between a port of a regular Simulink block and a port of a Simscape block. Since the connected ports are of different kind, it is necessary to use an additional block that will convert a regular Simulink signal into a Simscape signal, or vice versa.

The following Simulink code connects a signal flow component and a physical interaction component, corresponding to Figure 12. It has a block *s1* referring to a Simulink block *Spring*, a block *tr1* converting regular signals to physical signals, a block *s2* referring to a Simscape component *Spring*, a block *tr2* converting physical signals to regular signals, and a block *s3* also referring to a Simulink block *Spring*. Lines of type *Connection* link *s1*, *tr1*, *s2*, *tr2*, and *s3*.

```
<Block BlockType="Reference" Name="s1" SID="1">
  <P Name="Ports">[1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
</Block>

<Block BlockType="Reference" Name="tr1" SID="2">
  <P Name="Ports">[1, 0, 0, 0, 0, 0, 1]</P>
  <P Name="SourceBlock">nesl_utility/Simulink-PS
Converter</P>
  <P Name="SourceType">Simulink-PS
Converter</P>
</Block>

<Block BlockType="Reference" Name="s2" SID="3">
  <P Name="Ports">[0,0,0,0,0,1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
  <P Name="SourceType">Spring</P>
  <P Name="SourceFile">Library.Spring </P>
  <P Name="ComponentPath">Library.Spring </P>
  <P Name="ClassName">Spring</P>
</Block>
<Block BlockType="Reference" Name="tr2" SID="4">
  <P Name="Ports">[0, 1, 0, 0, 0, 1]</P>
  <P Name="SourceBlock">nesl_utility/PS-Simulink
Converter</P>
  <P Name="SourceType">PS-Simulink
Converter</P>
</Block>
<Block BlockType="Reference" Name="s3" SID="5">
  <P Name="Ports">[1,1]</P>
  <P Name="SourceBlock">Library/Spring</P>
</Block>

<Line>
  <P Name="Src">1#out:1</P>
  <P Name="Dst">2#in:1</P>
</Line>
<Line LineType="Connection">
  <P Name="Src">2#rconn:1</P>
  <P Name="Dst">3#lconn:1</P>
</Line>
<Line LineType="Connection">
  <P Name="Src">3#rconn:1</P>
  <P Name="Dst">4#lconn:1</P>
</Line>
<Line>
  <P Name="Src">4#out:1</P>
  <P Name="Dst">5#in:1</P>
</Line>
```

10.7.7 Simscape modeling

SysML connectors correspond to Simscape connections.

The following Simscape code corresponds to Figure 12. It has two components *s1* and *s2* of type *Spring*, with a connection between *s1.p2* and *s2.p1*.

```
component Model
  components
    s1=Library.Spring;
    s2=Library.Spring;
  end
  connections
    connect(s1.p2, s2.p1);
  end
end
```

10.7.8 Summary

SysML	Modelica	Simulink	Simscape
Connector between ports with in or out flow properties	Connect equation between components	Line between inport/outports	Connect statement
Connector between ports with inout flow properties	Connect equation between ports	Physical line between left/right connectors	Connect statement
Connector owner	Class containing equations	System of the Subsystem	Component

10.8 Blocks with constraint properties and binding connectors

10.8.1 Purpose

System behavior is represented in simulation models by assignments or equations relating values of system properties. Simulating assignments and equations involves computing an unknown variable from known variables. In assignments, unknown and known variables are specified at the time of modeling, while in equations, unknown and known variables are determined during simulation.

10.8.2 SysML modeling

Simulation equations and assignments correspond to constraint blocks in SysML. Constraint blocks are blocks that have parameters and constraint properties (properties typed by constraint blocks). Parameters are properties present in the equations, while constraints are equations. Modeling assignments in SysML is not used in this translation.

Regular SysML blocks use constraint blocks by typing properties with them (constraint properties), and owning binding connectors that link parameters of the constraint blocks to other properties of the block.

Subclauses 10.8.3 through 10.8.6 cover signal flow modeling, while subclauses 10.8.7 through 10.8.10 cover physical interaction modeling.

10.8.3 SysML modeling, signal flow

Figure 13 shows an example constraint block for a signal flow application, using the port types defined in Figure 10, Subclause 10.6.3. It has a SysML constraint property *sc* typed by *SpringConstraint*. The constraint block has six parameters, each bound to a property reachable from the containing block *Spring*: *f* is bound to the signal of a port *u*, which has a type with an in flow property, *pos* is bound to the signal of a port *y*, which has a type with an out flow property, *x* is bound to a SimVariable *position*, *k* is bound to a SimConstant *springcst*, *v* is bound to the SimVariable *velocity*, and *m* is connected to the SimConstant *mass*. The constraint block defines three constraints representing equations, written in the expression language specified in Clause 8.

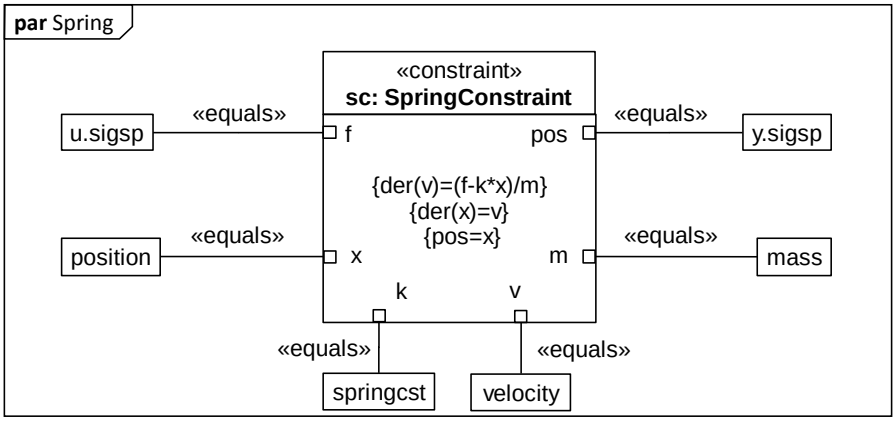

Figure 13: Constraint block for signal flow in SysML

10.8.4 Modelica modeling, signal flow

In a SysML block that contains a constraint property, the equations from the constraint block are the same as in Modelica (assuming the expression language of Clause 8 is used in the constraint block) and the SysML parameters in those equations are replaced in Modelica by the properties they are bound to in SysML.

The following Modelica code corresponds to Figure 13. It has three equations from the constraint block. Parameter names in the Modelica equations are replaced by the property names they are bound to in SysML: *f* is replaced by *u*, *x* is replaced by *position*, *k* is replaced by *springcst*, *v* is replaced by *velocity*, *m* is replaced by *mass*, and *pos* is replaced by *y*. Note that the bound property for SysML ports is modified in the corresponding Modelica, to reflect that SimProperties have no corresponding construct in Modelica.

```
model Spring
  input Real u;
  output Real y;
  Real position;
  parameter Real springcst = 1;
  Real velocity;
  parameter Real mass = 10;
equations
  der(velocity)=(u-springcst*position)/m;
  der(position)=velocity;
  y=position;
end Spring;
```

10.8.5 Simulink modeling, signal flow

SysML constraint blocks for signal flow correspond to Simulink S-Functions. S-Functions are a kind of MATLAB function that define input variables, output variables, continuous state variables, and discrete state variables. S-Function variables are identified by numbers, rather than names. State variable are accessible only inside an S-Function (this is different from states in state machines, see Subclause 10.11). SysML constraint block parameters correspond to S-Functions based on how they are bound in SysML, which can be different for each constraint property typed by the same constraint block. This means that a separate S-Function corresponds to each SysML constraint property. Each S-Function is used only in a specific context (corresponding to the constraint property), and the name of the S-Function must reflect that context.

S-Functions contain assignments of continuous state variable derivatives, discrete state variables, and output variables. These assignments correspond to constraints of SysML constraint blocks that have exactly one variable on the left-hand side, which determines the variable being assigned, and the kind of assignment it is:

- a continuous state variable on the left-hand side corresponds to a derivative assignment
- a discrete state variable on the left-hand side corresponds to an update assignment
- an output variable on the left-hand side is corresponds to an output assignment

SysML parameter names are used as variable names in the S-Functions. SysML parameters bound to SimConstants are replaced in S-Functions by the value given for the SimConstant.

Binding connectors involving ports with in or out flow properties correspond to Simulink lines (see Subclause 10.7.4) linking inports and outports to inputs and outputs of the S-Function, respectively.

The following Simulink code corresponds to Figure 13.

```
<Block BlockType="SubSystem" Name="Spring" SID="1">
  <P Name="Ports">[1,1]</P>
  <System>
    <Block BlockType="Inport" Name="u" SID="2">
      <P Name="Port">1</P>
      </Block>
    <Block BlockType="Outport" Name="y" SID="3">
      <P Name="Port">1</P>
    </Block>
    <Block BlockType="M-S-Function" Name="sc" SID="4">
      <P Name="FunctionName">Spring_sc_SpringConstraint</P>
      <P Name="Ports">[1,1]</P>
    </Block>
    <Line>
      <P Name="Src">2#out:1</P>
      <P Name="Dst">4#in:1</P>
    </Line>
    <Line>
      <P Name="Src">4#out:1</P>
      <P Name="Dst">3#in:1</P>
    </Line>
  </System>
</Block>
function Spring_sc_SpringConstraint(block)
  setup(block);
end
function setup(block)
  block.NumInputPorts =1;
  block.NumOutputPorts =1;
  block.NumContStates =2;
  block.RegBlockMethod('Derivatives',@Derivative);
  block.RegBlockMethod('Outputs',@Output);
  block.SampleTime=[0 0];
end
function Derivative(block)
  block.Derivatives.Data(1)=(block.InputPort(1).Data-1*block.ContStates.Data(2))/10;
  block.Derivatives.Data(2)=block.ContStates.Data(2);
end
function Output(block)
  block.OutputPort(1).Data=block.ContStates.Data(2);
end
```

The top part of the code shows a Simulink block *Spring* with one inport and one outport. *Spring* also contains a S-Function block that points at the S-Function *Spring_sc_SpringConstraint*, which has one inport and one outport. The inports and outports of *Spring* are linked to the inport and outport of the S-Function block, respectively.

The bottom part of the code shows the S-Function *Spring_sc_SpringConstraint*. The setup function indicates that the S-Function has one input port, one output port, and two continuous states. The function also registers two functions that will be called for derivative calculations and output calculations. These functions contain the assignments from the SysML constraints, with all the necessary substitutions performed.

10.8.6 Simscape modeling, signal flow

Simscape supports signal flow by providing a way to specify input and output signals for components. The content of SysML constraint blocks and binding connectors is be merged in Simulink components, with the necessary substitutions made in the equations (similar to Modelica, see Subclause 10.8.4). Simscape does not support discrete variables (compare to S-Functions, see Subclause 10.8.5).

The following Simscape code corresponds to Figure 13. It has a component *Spring* with an input *u*, an output *y*, two parameters *springcst* and *mass*, as well as two variables *position* and *velocity*. The component has equations connecting these variables: two equations that compute the derivative of the continuous variables, and one that determines the output.

```
component Spring
  inputs
    u = 0;
  end
  outputs
    y = 0;
  end
  parameters
    springcst = 1;
    mass = 10;
  end
  variables
    position = 0;
    velocity = 0;
  end
  equations
    der(velocity)=(u-springcst*position)/m;
    der(position)=velocity;
    y=position;
  end
end
```

10.8.7 SysML modeling, physical interaction

Figure 14 shows an example constraint block for a signal flow application, using the port typed defined in Figure 11, Subclause 10.6.7. It has a constraint block *SpringConstraint* with 8 parameters, and 5 constraints. The parameters include the force and the velocity from the two ends of the spring (*f1*, *v1*, *f2*, *v2*), the position of the spring (*x*), the spring constant (*k*), as well as the force and velocity related to the spring (*v*, *f*). The constraints relate values of the parameters. These constraints are written using the expression language specified in Clause 8.

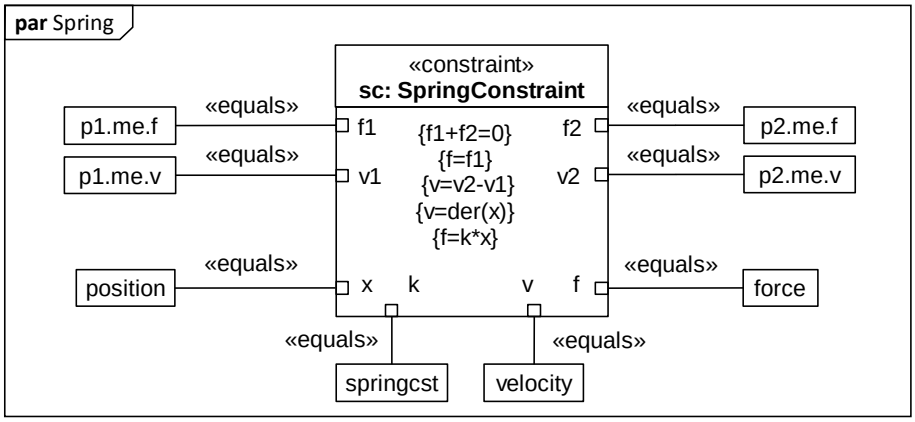

Figure 14: Constraint block for physical interaction in SysML

10.8.8 Modelica modeling, physical interaction

The correspondence between physical interaction in SysML and Modelica is the same as for signal flow (see Subclause 10.8.4). In a SysML block with constraint properties, the equations of the constraint blocks are the same as in Modelica (assuming the expression language of Clause 8 is used in the SysML constraint block), except the SysML parameters in those equations correspond in Modelica to the properties they are bound to in SysML.

The following Modelica code corresponds to Figure 14. It has a model Spring with two components *p1* and *p2* of type *Flange*, three continuous components *position*, *velocity*, and *force* of type Real, and one parameter *springcst* of type Real. *Spring* also contains the equations linking the variables together, with parameter names in the Modelica equations replaced by the property names they are bound to in SysML: *f1* is replaced by *p1.f*, *v1* is replaced by *p1.v*, *x* is replaced by *position*, *k* is replaced by *springcst*, *v* is replaced by *velocity*, *f* is replaced by *force*, *v2* is replaced by *p2.v*, and *f2* is replaced by *p2.f*.

```
model Spring
  Flange p1;
  Flange p2;
  Real position;
  parameter Real springcst = “10”;
  Real velocity
  Real force
equation
  p1.f+p2.f=0
  force=p1.f;
  velocity=p1.v-p2.v;
  velocity=der(position);
  force=springcst*position;
end Spring;
```

10.8.9 Simulink modeling, physical interaction

Physical interaction is modeled with the Simscape extension to Simulink, see Subclause 10.8.10.

10.8.10 Simscape modeling, physical interaction

For SysML blocks with constraint properties, the constraints of the constraint blocks are the same as the equations in the corresponding Simscape components (see Subclause 10.2.6), with substitutions in Simscape similar to those for Modelica (see Subclause 10.8.8), plus additional subsitutions for conserved variables on Simscape domains (see Subclause 10.6.10), which cannot be used in Simscape equations. A new Simscape variable is defined for each conserved variable used by a node (even if multiple nodes have the same domain), along with a branch statement indicating that the new variable value is the same as the conserved variable. The new variable is used in Simscape equations instead of the conserved variable.

The following Simscape code corresponds to Figure 14. It has a component Spring, with two nodes (*p1*, *p2*) of type *Flange*, five variables (*force*, *velocity*, *position*, *p1f*, *p2f*) and one parameter (*springcst*). Note the last two variables and the corresponding branch statement, which replace *p1.f* by *p1f* and *p2.f* by *p2f*.

```
component Spring
  variables
    force={0,'N'};
    velocity={0,'m/s'};
    position={0, 'm'};
    p1f={0,'N'};
    p2f={0,'N'};
  end
  nodes
    p1=Library.Flange;% :left
    p2=Library.Flange;% :right
  end
  parameters
    springcst={10,'1'};
  end
  function setup
  end
  branches
    p1f: p1.f->*;
    p2f: p2.f->*;
  end
  equations
    p1f+p2f=0;
    force=p1f;
    velocity=p1.v-p2.v;
    velocity=der(position);
    force=springcst*position;
  end
end
```

10.8.11 Summary

SysML	Modelica	Simulink	Simscape
Constraint block, used in constraint properties	N/A	S-Function	N/A
Constraint parameter bound to a SimProperty referring to an in flow property	N/A (parameter substituted in equations)	Input variable	N/A (parameter substituted in equations)
Constraint parameter bound to a SimProperty referring to an out flow property	N/A (parameter substituted in equations)	Output variable	N/A (parameter substituted in equations)

Constraint parameter bound to continuous SimVariable	N/A (parameter substituted in equations)	Continuous state variable	N/A (parameter substituted in equations)
Constraint parameter bound to discrete SimVariable	N/A (parameter substituted in equations)	Discrete state variable	N/A (parameter substituted in equations)
Constraint parameter bound to discrete SimParameter	N/A (parameter substituted in equations)	Numerical value (substituted in equations)	N/A (parameter substituted in equations)
Constraint	Equation in the model containing the constraint property (with substitution of parameters)	Output, discrete, or derivative assignment depending on type of the left-hand side variable in the equations	Equation in the component containing the constraint property (with substitution of parameters)

Restrictions:

- constraint parameters must be named, but not necessarily be public or typed

10.9 Default values and initial values

10.9.1 Purpose

Systems and simulation models can specify values for data type properties to be used when values are not otherwise given.

10.9.2 SysML Modeling

SysML has two ways to specify values for properties that are used when values are not otherwise given:

- *Default values* are defined on the properties that will be given the values. A default value is given to every instance of the block owning the property (or any block it generalizes) when each instance is created.
- *Initial values* are defined on other properties that are typed by the block owning the property (or any block it generalizes) that will be given the values. The values are given to instances of the block when (and if) they become values of the other properties.

Initial values override default values, because initial values are set when an instance that is already created becomes the value of another property that specifies initial values, but default values are only set when instances are created. Default and initial values can be changed after they are given to the instances.

Figure 15 shows how default and initial values are used in SysML. The left side of the figure shows a block *B* with an attribute *val* with a default value on 10. The right side shows a block *A* with an attribute *b* of type *B*. An initial value of 20 is given to the *val* of *b*.

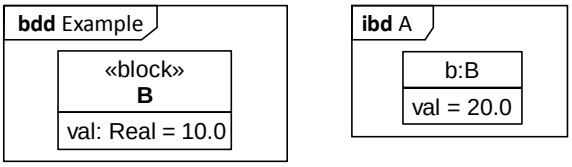

Figure 15: Default values and initial value in SysML

10.9.3 Modelica modeling

SysML default values correspond to start values of Modelica components. Start values are usually marked as fixed, requiring at the beginning time of the simulation (otherwise, simulators only take the values as suggestions,

calculating their own start values to solve the equations). Initial values in SysML correspond to start values on components, marked as fixed.

The following Modelica code corresponds to Figure 11. It has a model *B* with a *val* component. The *val* component has a start value of 10. A class *A* is defined with a component *b* of type *B*. A component modification indicates that the start value of *b.val* is 20.0.

```
model B
  Real val(start = 10.0, fixed = true);
end B;
model A
  B b(val.start = 20.0, val.fixed = true);
end A;
```

10.9.4 Simulink modeling

Default values (or overriding initial values) of SimVariables correspond to initial values of the corresponding S-Functions variables.

The following Simulink code corresponds to Figure 11, assuming the SimVariable *var* is bound to a constraint parameter (which translates into an S-Function variable, see Subclause 10.8.5). The code shows an S-Function setting initial values for discrete and continuous variables. It also shows a *setup* function that defines one continuous variable and one discrete variable, which are identified by number (1 for both in this example, see Subclause 10.8.5). The properties *NumDworks*, *Dwork*, *NumContStates*, and *ContStates* are predefined in Simulink, the first two for discrete variables, the second two for continuous variables. A value of 20 is given to both variables.

```
function setup(block)
  block.NumDworks = 1;
  block.Dwork(1).Data = 20.0;

  block.NumContStates = 1;
  block.ContStates.Data(1) = 20.0;
end
```

10.9.5 Simscape modeling

SysML default values correspond to initial values of Simscape variables and parameters. SysML initial values correspond to Simscape components used in Simulink. The priority of the initial value in Simscape must be set to high (otherwise simulators calculate initial values that solve the equations at the beginning time of the simulation)

The following Simscape code corresponds to the BDD in Figure 11. It code shows a Simscape component *B* defining a variable *val* with an initial value of 10.

```
component B
  variables
    val={value=10,priority=priority.high};
  end
end
```

The following Simulink code corresponds to the IBD in Figure 11. It has a usage of the Simscape component in Simulink that overrides the initial value of the variable *val* with a value of 20.

```
<Block BlockType="Reference" Name="b" SID="2">
  <P Name="SourceBlock">Library/B</P>
  <P Name="SourceType">B</P>
  <P Name="SourceFile">Library.B</P>
  <P Name="ComponentPath">Library.B</P>
  <P Name="ClassName">B</P>
  <P Name="val">20.0</P>
</Block>
```

10.10 Data types and units

10.10.1 Purpose

Systems and simulation models include units of physical quantities to enable checking that variables in equations and assignments have consistent units.

10.10.2 SysML modeling

SysML numeric data types can be linked to units, where units are modeled with the SysML Unit block. These units are linked to datatypes that are generalized by one SysML's numeric data types.

Figure 16 shows how a data type with units is defined. It has a value type *Force*, which specializes the *Real* datatype, and has *newton* as unit. The *newton* unit has a symbol *N*.

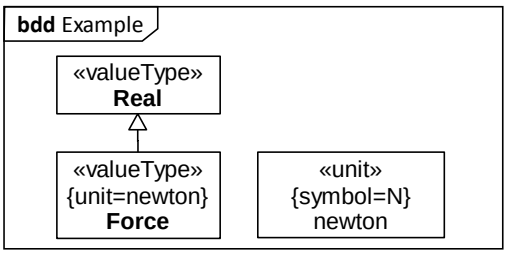

Figure 16: Units in SysML

10.10.3 Modelica modeling

Modelica data types can be subtyped to add a unit symbol. The interpretation of this symbol is not defined in Modelica.

The following Modelica code corresponds to Figure 16. It has a type *Force*, which extends from *Real*, and the symbol *N* assigned to it.

```
type Force=Real(unit="N");
```

10.10.4 Simulink modeling

Simulink does not support modeler-defined datatypes and units.

10.10.5 Simscape modeling

Unit symbols can be associated to variables and parameters in Simscape. Simscape will check that variables in equations have consistent units. Simscape has predefined unit symbols, and modelers can define their own.

The following Simscape code corresponds to Figure 16. It has a variable *force* with an initial value of *0*, and a predefined unit symbol *N*. Simscape interprets *N* as the unit for Newton, and checks consistency of units in equations involving *force*.

```
variables
  force={0,'N'};
end
```

10.10.6 Summary

SysML	Modelica	Simulink	Simscape
ValueType with unit	Type with unit symbol	N/A	N/A
Property typed by ValueType	Component typed by type	N/A	Variable with associated unit
Real	Real	double	double
String	String	N/A	N/A
Boolean	Boolean	boolean	N/A
Integer	Integer	int32	N/A

10.11 State machines

10.11.1 Purpose

State machines in system and simulation modeling specify how systems and components react to changes, usually caused by their environment (this is different than simulation state variables, see Subclause 10.8.5). State machines contain states and transitions between them. Objects are said to be "in" particular states, with transitions specifying when objects change the state they are in. States define behaviors for objects that are in those states. Transitions have conditions specifying when their objects change state. When conditions change for an object, usually as an effect of its environment, transitions can react by changing the state of the object, and consequently the behavior of the object. State machines can contain other state machines and can be in multiple states at the same time, but this specification does not provide translations for these capabilities.

10.11.2 SysML modeling

SysML state machines can be behaviors for blocks. The SysML capabilities of concern to simulation are:

- Triggering transitions based on evaluation of expressions, involving time and property values, including values arriving in flow properties on port types. These can be modeled using ChangeEvents.
- Sending values out of an object through a port with an out flow property when a specific state is on.

Figure 17 shows a block *Computer* with a simple state machine.

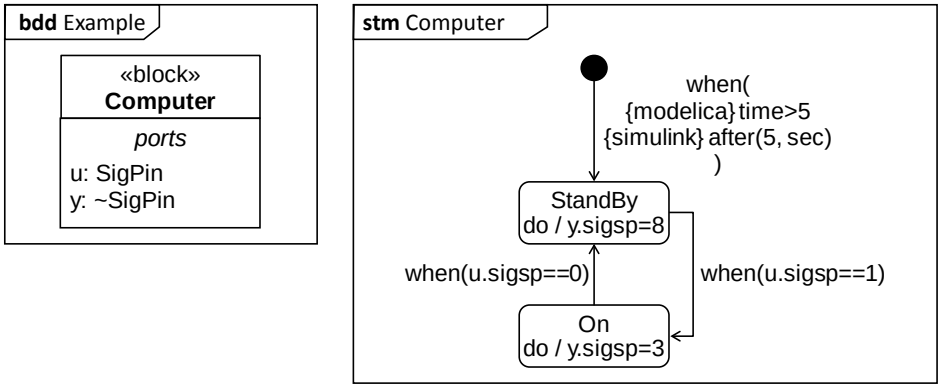

Figure 17: State machine in SysML

Computer has one port *u* of type SigPin (see Subclause 10.6.3 for the definition of SigPin), and one conjugated port *y* of type SigPin. The state machine has one initial pseudostate, and two states *StandBy* and *On*. The transition from

the initial pseudostate to *StandBy* has a ChangeEvent with an expression indicating that the transition is triggered 5 seconds after the beginning of the simulation. The expression has a body written in Modelica, and a body in Simulink (indicated by the language name between curly brackets). The transition from *StandBy* to *On* has a ChangeEvent with an expression indicating that the transition is triggered when *u.sigsp* is equal to 1 (this is a signal as in signal flow simulation, not as in SysML). The transition from *On* to *StandBy* has a ChangeEvent with an expression indicating that the transition is triggered when *u.sigsp* is equal to 0. When the computer is in *StandBy*, *y.sigsp* is set to 8, and when the computer is *On*, *y.sigsp* is set to 3.

10.11.3 Modelica modeling

Modelica 3.3 introduced support for state machines, but they are not widely implemented in simulation tools as of the date of this specification. Instead, this translation uses the Modelica standard library, which supports some aspects of state machines. SysML state machines correspond to Modelica models, and all the SimVariables and constants of a SysML block owning a state machine are the same as in the Modelica state machine. SysML state machine elements correspond to Modelica state machines as follows:

- Initial pseudostates correspond to InitialSteps.
- States correspond to Steps.
- Transitions correspond to Transitions.
- Change events correspond to transition conditions.
- State behaviors (specified with doActivity) that are OpaqueBehaviors correspond to Modelica code executed when objects are in particular states.

The following Modelica code corresponds to Figure 17.

```
model Computer
  input Real u;
  output Real y;
  ComputerSM _ComputerSM;
  model ComputerSM
    Modelica.StateGraph.InitialStep state0(nIn = 0, nOut = 1);
    Modelica.StateGraph.Step StandBy(nIn = 2, nOut = 1);
    Modelica.StateGraph.Step On(nIn = 1, nOut = 1);
    Modelica.StateGraph.Transition tr0(condition = time>5);
    Modelica.StateGraph.Transition tr1(condition = u==1);
    Modelica.StateGraph.Transition tr2(condition = u==0);
    Real u;
    Real y;
  equation
    connect(state0.outPort[1], tr0.inPort);
    connect(tr0.outPort, StandBy.inPort[1]);
    connect(StandBy.outPort[1], tr1.inPort);
    connect(tr1.outPort, On.inPort[1]);
    connect(On.outPort[1], tr2.inPort);
    connect(tr2.outPort, StandBy.inPort[2]);
  algorithm
    if StandBy.active then
      y := 8;
    end if;
    if On.active then
      y := 3;
    end if;
  end ComputerSM;
equation
  u = _ComputerSM.u;
  y = _ComputerSM.y;
end Computer;
```

The code shows the model *Computer* with an input variable *u*, and an output variable *y*, and a component *_ComputerSM* for a state machine *ComputerSM*, defined next. *ComputerSM* duplicates the components of *Computer*, except for the state machine component. It has an initial step *state0*, two steps *StandBy* and *On*, and three transitions *tr0*, *tr1* and *tr2*. Each transition has a condition for traversing it, and each step indicates how many inputs and outputs it has. *ComputerSM* contains equations linking ports of steps and transitions, and an algorithm section for assigning numeric component values when the machine starts or stops each step. Returning to *Computer*, equations bind its components to the components of the state machine.

10.11.4 Simulink/StateFlow modeling

Simulink has an extension for state machines called Stateflow, providing some features of SysML state machines. StateFlow supports transitions with conditions determining whether to traverse them, and actions performed when objects are in particular states. It uses default transitions, rather than transitions from initial pseudostates as in SysML. StateFlow state machines are blocks, rather than separate behaviors, as in SysML.

The following Simulink and StatFlow code corresponds to Figure 17.

```
<Block BlockType="SubSystem" Name="Computer" SID="2">
  <P Name="Ports">[1,1]</P>
  <P Name="SFBlockType">Chart</P>
  <System>
    <P Name="Open">off</P>
    <Block BlockType="Inport" Name="u" SID="2::1">
      <P Name="Port">1</P>
    </Block>
    <Block BlockType="Outport" Name="y" SID="2::2">
      <P Name="Port">1</P>
    </Block>
    <Block BlockType="S-Function" Name=" SFunction " SID="2::5">
      <P Name="FunctionName">sf_sfun</P><P Name="Ports">[1,2]</P>
    </Block>
    <Block BlockType="Demux" Name="Demux" SID="2::6">
      <P Name="Outputs">1</P>
    </Block>
    <Block BlockType="Terminator" Name="Terminator" SID="2::7"/>


    <Line>
      <P Name="Src">2::1#out:1</P><P Name="Dst">2::5#in:1</P>
    </Line>
    <Line>
      <P Name="Src">2::5#out:2</P><P Name="Dst">2::2#in:1</P>
    </Line>
    <Line>
      <P Name="Src">2::5#out:1</P><P Name="Dst">2::6#in:1</P>
    </Line>
    <Line>
      <P Name="Src">2::6#out:1</P><P Name="Dst">2::7#in:1</P>
    </Line>
  </System>
</Block>

<Stateflow>
  <machine id="1">
    <P Name="isLibrary">0</P>
    <Children>
      <target id="2" name="sfun"/>
      <chart id="3">
        <P Name="name">Computer</P>
```

```
        <P Name="chartFileNumber">1</P>
        <P Name="saturateOnIntegerOverflow">1</P>
        <P Name="userSpecifiedStateTransitionExecutionOrder">1</P>
        <P Name="disableImplicitCasting">1</P><P Name="actionLanguage">2</P>
        <Children>
          <state SSID="5">
            <P Name="labelString">StandBy
during:y=8;</P>
          </state>
          <state SSID="6">
            <P Name="labelString">On
during:y=3; </P>
          </state>
          <data SSID="7" name ="u">
            <P Name="scope">INPUT_DATA</P>
          </data>
          <data SSID="8" name ="y">
            <P Name="scope">OUTPUT_DATA</P>
          </data>
          <transition SSID="11">
            <P Name="labelString">[after(5, sec)]</P>
            <src/>
            <dst>
              <P Name="SSID">5</P>
            </dst>
            <P Name="executionOrder">1</P>
          </transition>
          <transition SSID="12">
            <P Name="labelString">[u==1]</P>
            <src>
              <P Name="SSID">5</P>
            </src>
            <dst>
              <P Name="SSID">6</P>
            </dst>
            <P Name="executionOrder">1</P>
          </transition>
          <transition SSID="13">
            <P Name="labelString">[u==0]</P>
            <src>
              <P Name="SSID">6</P>
            </src>
            <dst>
              <P Name="SSID">5</P>
            </dst>
            <P Name="executionOrder">1</P>
          </transition>
        </Children>
      </chart>
    </Children>
  </machine>
  <instance id="4">
    <P Name="name">Computer</P>
    <P Name="machine">1</P>
    <P Name="chart">3</P>
  </instance>
</Stateflow>
```

The *Block* section of the code at the top is the part of state machine represented in Simulink. It shows a block *Computer* of type Chart, containing one inport (*u*), one outport (*y*), and one S-Function corresponding to the state machine. The two other blocks, *Demux* and *Terminal*, are needed by Simulink to execute state machines. Lines connect the inport of the block to the input of the S-function, and the second output of the S-Function to the outport of the block.

The *Stateflow* section of the code at the bottom is the part of the state machine represented in Stateflow. It shows a machine containing one input *u*, one output *y*, two states *StandBy* and *On*, a default transition (which has no source), and two transitions. The *during* string in *StandBy* indicates that the output y is set to 8 while the computer is in *StandBy*. The label in the default transition indicates that the transition is fired after 5 seconds. The condition of the two transitions indicate that the first transition fires when the input *u* is equal to 1, and the second transition fires when the input *u* is equal to 0.

10.11.5 Summary

SysML	Modelica	Simulink	Stateflow
Block with StateMachine as classifierBehevior	Model (regular)	Block of type SFBlockType	N/A
StateMachine	Block	S-Function	Chart in machine
Initial pseudostate	InitialStep component	N/A	N/A
State	Step component	N/A	State
Transition	Transition component	N/A	Transition
Transition from initial PseudoState	Transition component	N/A	Default transition
doActivity with OpaqueExpression	Statements in a state conditionalized by object being in that state	N/A	During statements in a state
ChangeEvent Trigger	Transition condition	N/A	Transition condition

10.12 Mathematical expressions

The following table shows replacements to be made in the syntax of the SysPISF expression language (see Clause 8) when translating to MATLAB, the expression language in Simulink, Simscape, and StateFlow. Translation to Modelica requires no replacements.

SysPISF expression	MATLAB equivalent
‘if’ … ‘then’ … ‘elseif’ … ‘then’ … ‘else’ … ‘end’ ‘if’	‘if’ … … ‘elseif’ … … ‘else’ … ‘end’
‘for’ … ‘in’ … ‘loop’ … ‘end’ ‘for’	‘for’ … ‘=’ … … ‘end’
‘=’	‘==’
‘<>’	‘~=’
‘not’	‘~’
‘and’	‘&&’

‘or’	‘||’
‘:=’	‘=’
‘div’	‘idivide’

11 Platform-independent component library

11.1 Introduction

This clause defines a platform-independent library of port types and component blocks for physical interaction and signal flow modeling in Subclauses 11.2 and 11.3, respectively. These elements can be reused in system models. Subclause 11.4 defines simulation platform stereotypes used in Subclause 11.3.

11.2 Port types

11.2.1 Signal flow

This subclause defines signal flow port types.

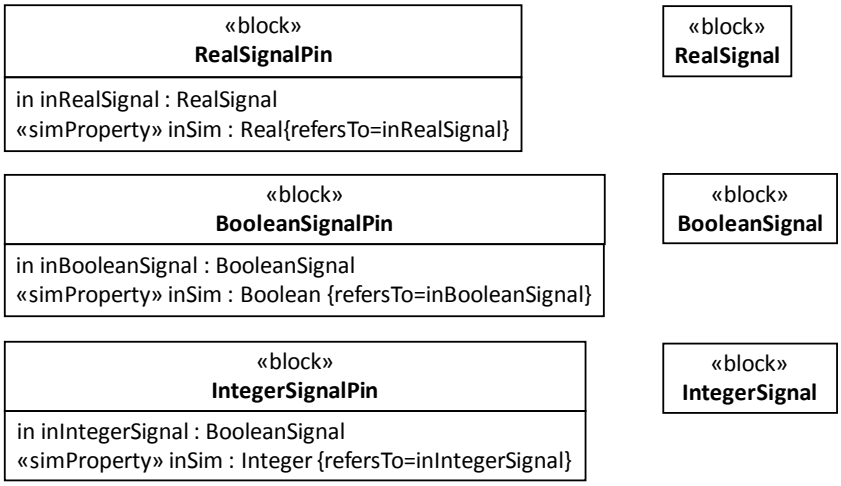

Figure 18: Basic signal flow port types

11.2.2 Physical interaction

This subclause defines basic physical interaction port types (see Figure 19) and associated value types and units (see Figure 20).

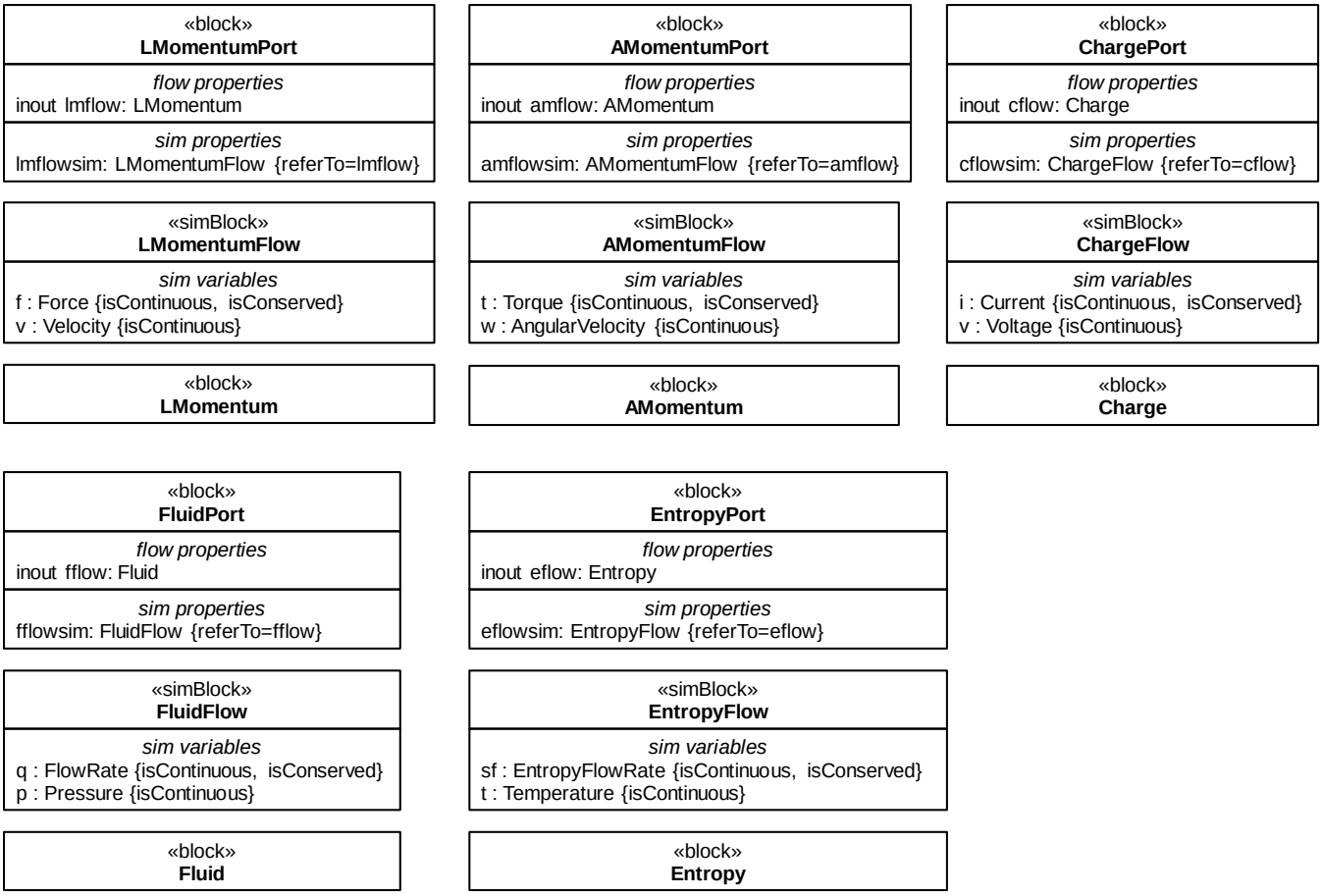

Figure 19: Basic physical interaction port types

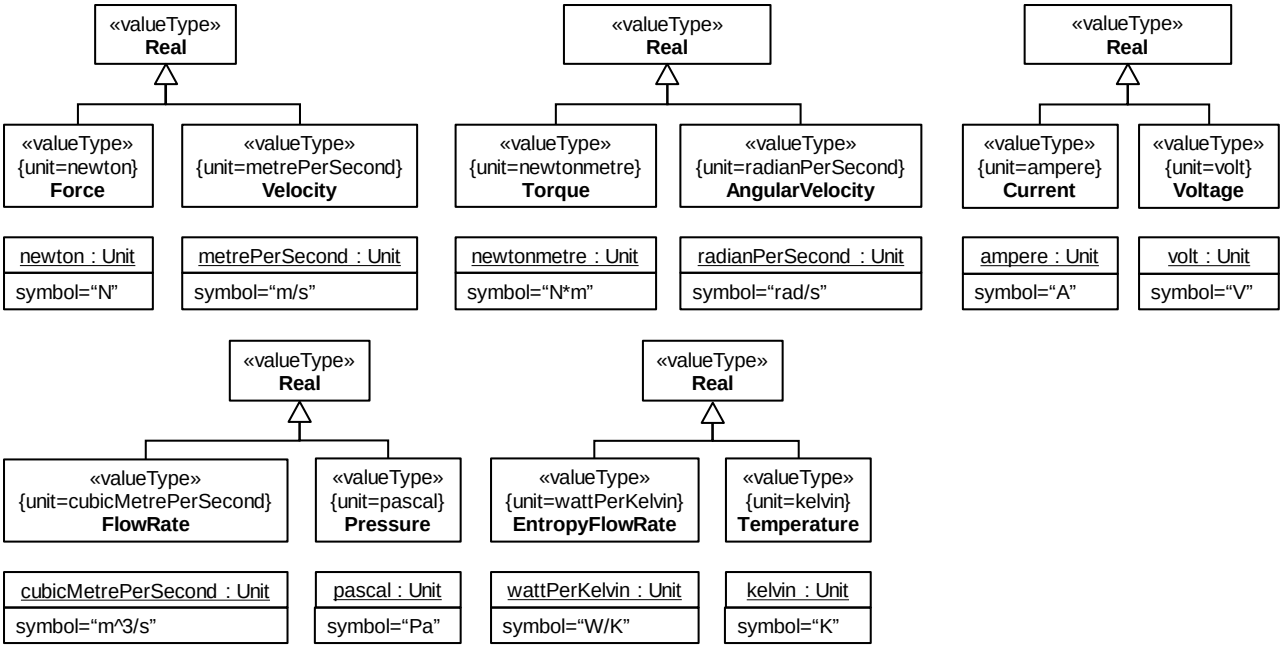

Figure 20: Basic physical interaction value types and units

11.3 Component blocks

This subclause defines SysML blocks corresponding to reusable component types supported by the libraries of both Modelica and Simulink or its extensions. The semantics of these blocks are given in the Modelica libraries (which is the same in the libraries of Simulink or its extensions). The base classes and properties (including ports) of component blocks have stereotypes of the simulation platform profile applied (see Subclause 11.4) to specify which simulation library elements correspond to them. For brevity, the blocks are described in tables, with each row defining one block.

The blocks in Subclauses 11.3.1 and 11.3.2 are for signal flow modeling. The columns of the tables are:

- *Block*: Name of the component block defined by the row.
  - *Simulink name*: Value of the name property of the SimulinkBlock stereotype applied to the base class of the block defined in the row.
  - *Modelica name*: Value of the name property of the ModelicaBlock stereotype applied to the base class of the block defined in the row.
- *InputPorts* and *OutputPort*: Each line in the cells of these columns is the name of a port stereotyped by ModelicaPort and SimulinkPort on the block defined in the row.
- *Parameters*: Each line in the cells of this column is the name of a property of the block defined by the row, with a corresponding line in the two columns below.
  - *Simulink Parameters*, *Modelica Parameters*: Value of the name properties of SimulinkParameter and ModelicaParameter stereotypes, respectively, applied to the corresponding property in the Parameter column. Additional lines, if any, indicate other parameters needed to obtain the same behavior in Simulink and Modelica, with the value of the parameter indicated by an equals sign.
- *Behavior*: Tells whether the behaviors of the Simulink and Modelica library elements is supposed to yield the same value or not, when this can be determined from the library specifications.

The values of simulation platform variables specified in the Input, Output, and Parameter columns are scalar, unless followed by a V (vector) or an M (matrix). This is represented in the block defined by each row using the MultidimensionalElement stereotype (see Subclause 11.4).

The blocks in Subclause 11.3.3 are for electrical modelling. The columns of the table are explained in that subclause.

11.3.1 Real-valued components

The input, outputs and parameters of the following blocks are of type Real, unless otherwise indicated.

11.3.1.1 Continuous components

Block	Simulink name	Modelica name	Input Ports	Output Port	Parameters	Simulink Parameters	Modelica Parameters	Behavior
Integrator	Integrator	Modelica.Blocks.Continuous.Integrator	u	y	init	InitialCondition	y_start	Same
Derivative	Derivative	Modelica.Blocks.Continuous.Derivative	u	y				Different
StateSpace	StateSpace	Modelica.Blocks.Continuous.StateSpace	u (V)	y (V)	A (M) B (M) C (M) D (M) init (V)	A (M) B (M) C (M) D (M) X0 (V)	A (M) B (M) C (M) D (M) x_start (V)	Same
TransferFunction	TransferFcn	Modelica.Blocks.Continuous.TransferFunction	u	y	num (V) denom (V)	Numerator (V) Denominator (V)	b (V) a (V)	
FixedDelay	TransportDelay	Modelica.Blocks.Nonlinear.FixedDelay	u	y	delay	DelayTime InitialOutput=0	delayTime	Different
VariableDelay	VariableTimeDelay	Modelica.Blocks.Nonlinear.VariableDelay	u delayTime	y	delayMax	MaximumDelay InitialOutput=0	delayMax	Different

11.3.1.2 Discrete components

Block	Simulink name	Modelica name	Input Ports	Output Port	Parameters	Simulink Parameters	Modelica Parameters	Behavior
State space	DiscreteStateSpace	Modelica.Blocks.Discrete.StateSpace	u (V)	y (V)	A (M) B (M) C (M) D (M)	A (M) B (M) C (M) D (M)	A (M) B (M) C (M) D (M)	Same
Transfer Function	DiscreteTransferFcn	Unit	u	y	numerator (V) denominator (V)	Numerator (V) Denominator (V)	b (V) a (V)	Same
Unit delay	UnitDelay	Modelica.Blocks.Discrete.UnitDelay	u	y	initialCondition	InitialCondition	y_start	Same

11.3.1.3 Non-linear components

Block	Simulink name	Modelica name	Input Ports	Output Port	Parameters	Simulink Parameters	Modelica Parameters	Behavior
Saturation	Saturate	Modelica.Blocks.Nonlinear.Limiter	u	y	upper lower	UpperLimit LowerLimit	uMax uMin	Same (min AND max mandatory)
Dynamic Saturation	Reference	Modelica.Blocks.Nonlinear.VariableLimiter	limit1 u limit2	y		SourceBlock= simulink/Discontinuities /Saturation Dynamic SourceType=Saturation Dynamic		Same
Dead zone	DeadZone	Modelica.Blocks.Nonlinear.DeadZone	u	y	lower upper	LowerValue UpperValue	uMin uMax	Same
Rate limiter	RateLimiter	Modelica.Blocks.Nonlinear.SlewRateLimiter	u	y	rising falling	RisingSlewLimit FallingSlewLimit	Rising Falling	Different

11.3.1.4 Mathematical components

Block	Simulink name	Modelica name	Input Ports	Output Port	Parameters	Simulink Parameters	Modelica Parameters	Behavior
Gain	Gain	Modelica.Blocks.Math.Gain	u	y	gain	Gain	k	Same
Product	Product	Modelica.Blocks.Math.Product	u1 u2	y		Inputs=**		Same
Division	Product	Modelica.Blocks.Math.Division	u1 u2	y		Inputs=*/		Same
Addition	Add	Modelica.Blocks.Math.Add	u1 u2	y		Inputs=++		Same
Subtraction	Add	Modelica.Blocks.Math.Add	u1 u2	y		Inputs=+-		Same
Abs	Abs	Modelica.Blocks.Math.Abs	u	y				Same
Exp	Math	Modelica.Blocks.Math.Exp	u	y		Operator=exp		Same
Log	Math	Modelica.Blocks.Math.Log	u	y		Operator=log		Same
Log10	Math	Modelica.Blocks.Math.Log10	u	y		Operator=log10		Same
Sign	Signum	Modelica.Blocks.Math.Sign	u	y				Same
Sqrt	Sqrt	Modelica.Blocks.Math.Sqrt	u	y				Same
Sin	Trigonometry	Modelica.Blocks.Math.Sin	u	y		Operator=sin		Same
Cos	Trigonometry	Modelica.Blocks.Math.Cos	u	y		Operator=cos		Same
Tan	Trigonometry	Modelica.Blocks.Math.Tan	u	y		Operator=tan		Same
Asin	Trigonometry	Modelica.Blocks.Math.Asin	u	y		Operator=asin		Same

Acos	Trigonometry	Modelica.Blocks.Math.Acos	u	y		Operator=acos		Same
Atan	Trigonometry	Modelica.Blocks.Math.Atan	u	y		Operator=atan		Same
Atan2	Trigonometry	Modelica.Blocks.Math.Atan2	u1 u2	y		Operator=atan2		Same
Sinh	Trigonometry	Modelica.Blocks.Math.Sinh	u	y		Operator=sinh		Same
Cosh	Trigonometry	Modelica.Blocks.Math.Cosh	u	y		Operator=cosh		Same
Tanh	Trigonometry	Modelica.Blocks.Math.Tanh	u	y		Operator=tanh		Same

11.3.1.5 Sources and sinks

Block	Simulink name	Modelica name	Input Ports	Output Port	Parameters	Simulink Parameters	Modelica Parameters	Behavior
Constant	Constant	Modelica.Blocks.Sources.Constant		y	k	Value	k	Same
Sine wave	Sin	Modelica.Blocks.Sources.Sine		y	amplitude offset frequency phase	Amplitude Bias Frequency Phase	amplitude offset freqHz phase	Same
Clock	Clock	Modelica.Blocks.Sources.Clock		y				Same
Pulse	DiscretePulseGenerator	Modelica.Blocks.Sources.Pulse		y	amplitude period width delay	Amplitude Period PulseWidth PhaseDelay	amplitude period width startTime	Same
Step	Step	Modelica.Blocks.Sources.Step		y	startTime after	Time After Before=0	startTime height	Same
RealScope	Scope	Modelica.Blocks.Interaction.Show.RealValue	numberPort					
BooleanScope	Scope	Modelica.Blocks.Interaction.Show.BooleanValue	activePort					

11.3.1.6 Routing components

Block	Simulink name	Modelica name	Input Ports	Output Ports	Parameters	Simulink Parameters	Modelica Parameters	Behavior
Mux2	Mux	Modelica.Blocks.Routing.Multiplex2	u1 u2	y		Inputs=2		Same
Mux3	Mux	Modelica.Blocks.Routing.Multiplex3	u1 u2 u3	y		Inputs=3		Same

Mux4	Mux	Modelica.Blocks.Routing.Multiplex4	u1 u2 u3 u4	y		Inputs=4		Same
Mux5	Mux	Modelica.Blocks.Routing.Multiplex5	u1 u2 u3 u4 u5	y		Inputs=5		Same
Mux6	Mux	Modelica.Blocks.Routing.Multiplex6	u1 u2 u3 u4 u5 u6	y		Inputs=6		Same
Demux2	Demux	Modelica.Blocks.Routing.DeMultiplex2	u	y1 y2		Outputs=2		Same
Demux3	Demux	Modelica.Blocks.Routing.DeMultiplex3	u	y1 y2 y3		Outputs=3		Same
Demux4	Demux	Modelica.Blocks.Routing.DeMultiplex4	u	y1 y2 y3 y4		Outputs=4		Same
Demux5	Demux	Modelica.Blocks.Routing.DeMultiplex5	u	y1 y2 y3 y4 y5		Outputs=5		Same
Demux6	Demux	Modelica.Blocks.Routing.DeMultiplex6	u	y1 y2 y3 y4 y5 y6		Outputs=6		Same
Switch			u1 u2 u3	y		Criteria = u2~=0 Threshold=0		Same

11.3.2 Logical components

The input, outputs and parameters of the following blocks are of type Boolean, unless otherwise indicated.

Block	Simulink name	Modelica name	Input Ports	Output Port	Parameters	Simulink Parameters	Modelica Parameters	Behavior
AND	Logic	Modelica.Blocks.Logical.And	u1 u2	y		Operator=AND Inputs=2		Same
OR	Logic	Modelica.Blocks.Logical.Or	u1 u2	y		Operator=OR Inputs=2		Same
NAND	Logic	Modelica.Blocks.Logical.Nand	u1 u2	y		Operator=NAND Inputs=2		Same
NOR	Logic	Modelica.Blocks.Logical.Nor	u1 u2	y		Operator=NOR Inputs=2		Same
XOR	Logic	Modelica.Blocks.Logical.Xor	u1 u2	y		Operator=XOR Inputs=2		Same
NOT	Logic	Modelica.Blocks.Logical.Not	u	y		Operator=NOT Inputs=1		Same
Less	RelationalOperator	Modelica.Blocks.Logical.Less	u1 (R) u2 (R)	y		Operator = <		Same
LessEqual	RelationalOperator	Modelica.Blocks.Logical.LessEqual	u1 (R) u2 (R)	y		Operator = <=		Same
Greater	RelationalOperator	Modelica.Blocks.Logical.Greater	u1 (R) u2 (R)	y		Operator = >		Same
GreaterEqual	RelationalOperator	Modelica.Blocks.Logical.GreaterEqual	u1 (R) u2 (R)	y		Operator = >=		Same
LessThreshold	Compare To Constant	Modelica.Blocks.Logical.LessThreshold	u (R)	y	threshold (R)	Const Relop = <	threshold	Same
LessEqualThreshold	Compare To Constant	Modelica.Blocks.Logical.LessEqualThreshold	u (R)	y	threshold (R)	Const relop = <=	threshold	Same
GreaterThreshold	Compare To Constant	Modelica.Blocks.Logical.GreaterThreshold	u (R)	y	threshold (R)	const relop = >	threshold	Same
GreaterEqualThreshold	Compare To Constant	Modelica.Blocks.Logical.GreaterEqualThreshold	u (R)	y	threshold (R)	const relop = >=	threshold	Same

11.3.3 Electrical components

The columns are the same as in Subclauses 11.3.1 and 11.3.2, except there is only one column for ports, because they are bidirectional, unless otherwise noted. Each line in the Ports column corresponds to a port stereotyped by SimulinkPort and ModelicaPort (the SimulinkPorts is for Simscape ports in this table). Each line in the Simulink Ports and Modelica Ports columns give the value of the name property of the SimulinkPort and ModelicaPort stereotypes, respectively, for the corresponding port in the Ports column.

Block	Simulink name	Modelica name	Ports	Simulink Ports	Modelica Ports	Parameters	Simulink Parameters	Modelica Parameters	Behavior
Ground	Reference	Ground	p	V	p				
Capacitor	capacitor	Capacitor	p n	p n	p n	c	c r=0 g=0	C	Same
Diode	pwl_diode	IdealDiode	p n	p n	p n	ron goff vforward	Ron Goff Vf	Ron Goff Vknee	
IdealTransformer	ideal_transformer	IdealTransformer	p1 n1 p2 n2	p1 n1 p2 n2	p1 n1 p2 n2	n	n	n	Same
Inductor	inductor	Inductor	p n	p n	p n	r	l r=0 g=0	L	Same
InfiniteResistance	infinite_resistance	Idle	p n	p n	p n				Same
OpAmp	op_amp	IdealOpAmp3Pin	p n out	p n out	in_p in_n out				Same
Resistor	resistor	Resistor	p n	p n	p n	r	R	R	Same
ControlledSwitch	controlled_switch	ControlledIdealOpeningSwitch	p n control (input)	p n vT	p n control	level	Threshold	level	Same
VariableResistor	variable_resistor	VariableResistor	p n r (input)	p n R	p n R				Same
CurrentSensor	current	CurrentSensor	p n i (output)	p n I	p n i				Same

VoltageSensor	voltage	VoltageSensor	p n v (output)	p n V	p n v				Same
SignalCurrent	controlled_current	SignalCurrent	p n i (input)	p n iT	p n i				Same
SignalVoltage	controlled_voltage	SignalVoltage	p n v (input)	p n vT	p n v				Same
DCCurrent	dc_current	ConstantCurrent	p n	p n	p n	i	i0	I	Same
DCVoltage	dc_voltage	ConstantVoltage	p n	p n	p n	v	v0	V	Same
ACCurrent	ac_current	SineCurrent	p n	p n	p n	amp phase freq	amp shift frequency	I phase freqHz	Same
ACVoltage	ac_voltage	SineVoltage	p n	p n	p n		amp shift frequency	V phase freqHz	Same

11.4 Simulation platform stereotypes

This subclause defines stereotypes that the platform-independent component library in Subclause 11.3.2 applies to the base classes and properties (including ports) of its blocks, to specify which library elements of Modelica and Simulink correspond to them. In this subclause, the Simulink library is taken as including the libraries of its extensions, for brevity.

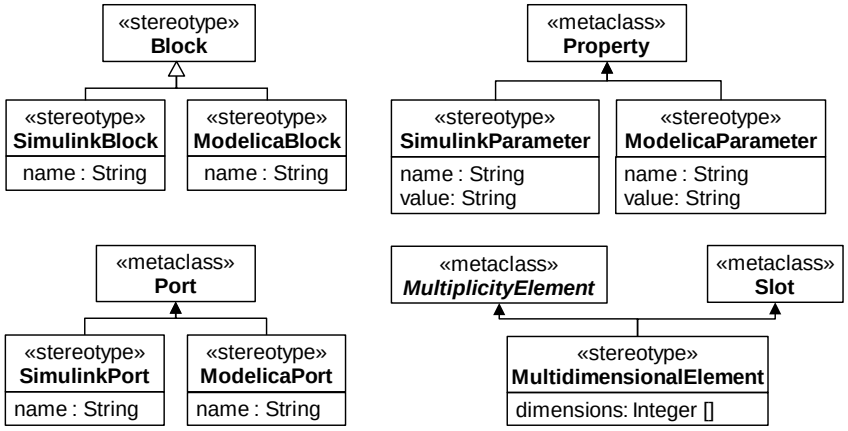

Figure 21: Simulation platform stereotypes

11.4.1 ModelicaBlock

Package: SimulationLibrary
isAbstract: No
Generalization: Block

Attributes

name: String	Fully qualified name of the component in the Modelica library corresponding to a platform-independent component block

Description

A class stereotyped by ModelicaBlock has an equivalent in the Modelica library. The value of the name attribute gives the fully qualified name of the corresponding component in the Modelica library.

11.4.2 ModelicaParameter

Package: SimulationLibrary
isAbstract: No
Extended Metaclass: Property

Attributes

name: String	Name of the parameter in the Modelica library corresponding to a parameter of a platform-independent component block
value: String[0..1]	Value of the parameter in the Modelica library

Description

A property stereotyped by ModelicaParameter has an equivalent parameter of a Modelica library component. The value of the name attribute is the name of the corresponding parameter, and the value attribute gives the value of this parameter. If the value attribute is empty, the value of the parameter must be given using initial values of the stereotyped property.

Constraints

[1] Must be owned by a class stereotyped by ModelicaBlock.

11.4.3 ModelicaPort

Package: SimulationLibrary
isAbstract: No
Extended Metaclass: Port

Attributes

name: String	Name of the port in the Modelica library corresponding to a port of a platform-independent component block

Description

A property stereotyped by ModelicaPort has an equivalent in the Modelica library. The value of the name attribute gives the name of the corresponding port in the Modelica library.

Constraints

[1] Must be owned by a class stereotyped by ModelicaBlock.

11.4.4 MultidimentionalElement

Package: SimulationLibrary
isAbstract: No
Extended Metaclass: MultiplicityElement, Slot

Attributes

dimensions: UnlimitedNatural [*] {ordered, non-unique}	Dimensions of the multiplicity element or slot

Description

The values of a slot stereotyped by MultidimensionalElement can be composed into an array with (possibly multiple) dimensions specified by the applied stereotype. The values are composed by taking each number in the dimension list of the applied stereotype from the last number to the second, and creating lists of that length from the result of the next higher dimension. The last dimension number results in lists of values of the multiplicity element or a slot, while the previous dimension number results in lists of those lists, and so on, ending at the second dimension number.

Constraints

[1] A multiplicity element stereotyped by MultidimensionalElement must be ordered and non-unique.
[2] When this stereotype is applied to a multiplicity element, the dimensions must be either all unlimited or all positive integers.
[3] When this stereotype is applied to a multiplicity element and the dimensions are all unlimited, the lower bound of the multiplicity element must be 0, and the upper bound of the multiplicity element must be unlimited.
[4] When this stereotype is applied to a multiplicity element and the dimensions are all be positive integers, the lower bound and the upper bound of the multiplicity element must be equal to the product of all the dimensions.
[5] When this stereotype is applied to a slot, the dimensions must all be positive integers and the number of values of the slot must be equal to the product of all dimensions.
[6] A slot stereotyped by MultidimensionalElement must have its defining feature stereotyped by MultidimensionalElement.
[7] The number of dimensions of a MultidimensionalElement applied to a slot must be the same as the number of dimensions of the MultidimensionalElement applied to the defining feature of the slot.
[8] A slot must be stereotyped by MultidimensionalElement if and only if its defining feature is stereotyped by MultidimensionalElement with dimensions that are all unlimited.

11.4.5 SimulinkBlock

Package: SimulationLibrary
isAbstract: No
Generalization: Block

Attributes

name: String	BlockType in Simulink library corresponding to a platform-independent component block

Description

A class stereotyped by SimulinkBlock has an equivalent in the libraries of Simulink or its extensions. The value of the name attribute gives the name of the corresponding component in the libraries of Simulink or its extensions.

11.4.6 SimulinkParameter

Package: SimulationLibrary
isAbstract: No
Extended Metaclass: Property

Attributes

name: String	Name of the parameter in the Simulink library corresponding to a parameter of a platform-independent component block
value: String[0..1]	Value of the parameter in the Simulink library

Description

A property stereotyped by SimulinkParameter has an equivalent parameter of a Simulink library component. The value of the name attribute is the name of the corresponding parameter in the Simulink library, and the ‘value’ attribute gives the value of this parameter. If the value attribute is empty, the value of the parameter must be given using initial values of the stereotyped property.

Constraints

[1] Must be owned by a class stereotyped by SimulinkBlock.

11.4.7 SimulinkPort

Package: SimulationLibrary
isAbstract: No
Extended Metaclass: Port

Attributes

name: String	Name of the port in the Simulink library corresponding to a port of a platform-independent component block

Description

A property stereotyped by SimulinkPort has an equivalent in the Simulink library. The value of the name attribute gives the name of the corresponding port in the Simulink library.

Constraints

[1] Must be owned by a class stereotyped by SimulinkBlock.

A. Tutorial (non-normative)

A.1 Introduction

This clause shows how to use the simulation profile and library to create a simple electrical circuit model.

A.2 System being modeled

The electrical circuit is made of three components: a ground, an electrical source, and a resistor, see Figure 22.

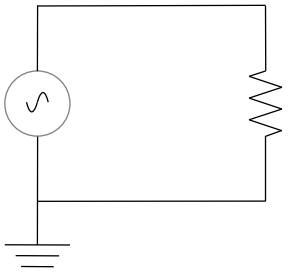

Figure 22: Electric circuit example

A.3 Blocks and ports

The electrical circuit model in SysML has a main block, the circuit. Create a *Circuit* block in a Block Definition Diagram (BDD). The circuit has three parts: a source, a ground, and a resistor. These parts are of different types, with different behaviors (behaviors are defined in Subclauses A.6 and A.7). Create a block for each of these part types. The three parts of the *Circuit* block are connected through ports, which represent electrical pins (connections are defined in Subclause A.4). The source and resistor have a positive and a negative pin. The ground has only one pin, which is positive. Electricity (electric charges) is transmitted through the pins. Create an abstract block *TwoPinComponent* with two ports (pins). The two ports are named *p* and *n*, and they are of type *ChargePort*, from the simulation library (see Subclause 11.2.2). Figure 23 shows what the BDD should look like, with the blocks *Circuit*, *Ground*, *TwoPinComponent*, *Source*, and *Resistor*.

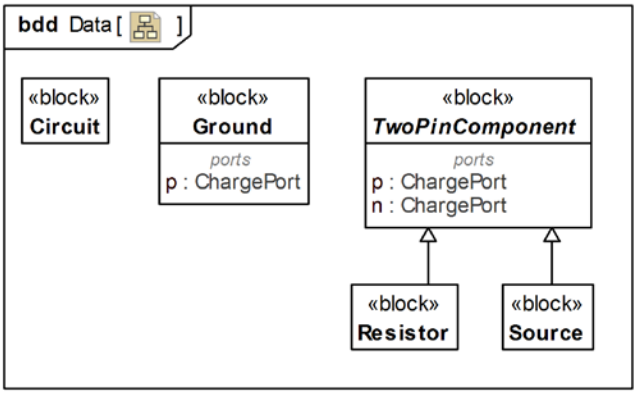

Figure 23: Electrical blocks, parts, and ports

A.4 Internal structure (parts and connectors)

Create an Internal Block Diagram (IBD) for *Circuit* from Subclause A.3. Add properties for the source, resistor, and ground, typed by the corresponding blocks from Subclause A.3 (some tools might require property definition in BDDs before showing them in IBDs). Connect the ports with connectors. The positive pin of the source is connected to the negative pin of the resistor. The positive pin of the resistor is connected to the negative pin of the source. The ground is also connected to the negative pin of the source. Figure 24 shows what the IBD should look like.

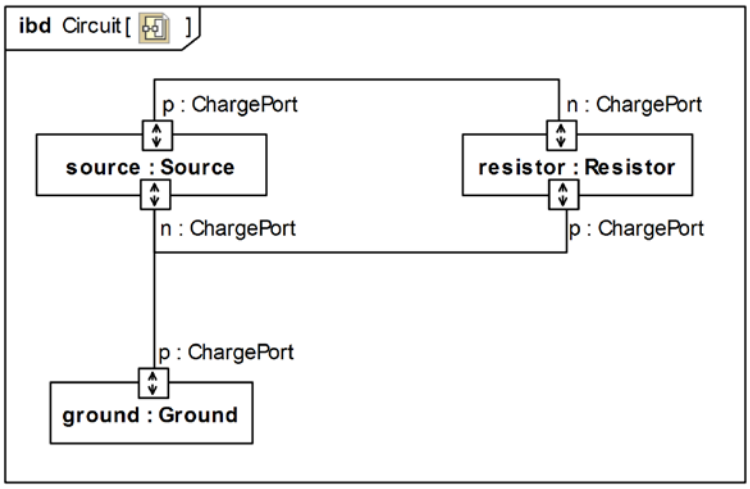

Figure 24: Internal structure of circuit

A.5 Properties (variables)

Flows of conserved physical substances are described by numeric variables for potential to flow and flow rate corresponding to the physical domain being modeled, in this example, flows of electrical charges described by voltage and current. For electrical components, voltage is the difference in potentials of the positive and negative pins (across the component), while current is the amount of charge going through the component per unit time. Create two properties *v* and *i* on *TwoPinComponent*. Their type must be a ValueType that has a unit with a symbol. Voltage and Current are provided by the simulation library. Apply the SimVariable stereotype to the properties, because the voltage and the current will vary over time during simulation. Create a property *r* in the resistor for its resistance. Create a new ValueType called *Resistance* to type for this property. The unit of *Resistance* must be an instance of SysML Unit with a symbol slot set to *ohm*. Generalize *Resistance* by *Real*. Apply the SimConstant stereotype to the *r* property, because the resistance will not change during simulation. Figure 25 shows what the BDD should look like at this point.

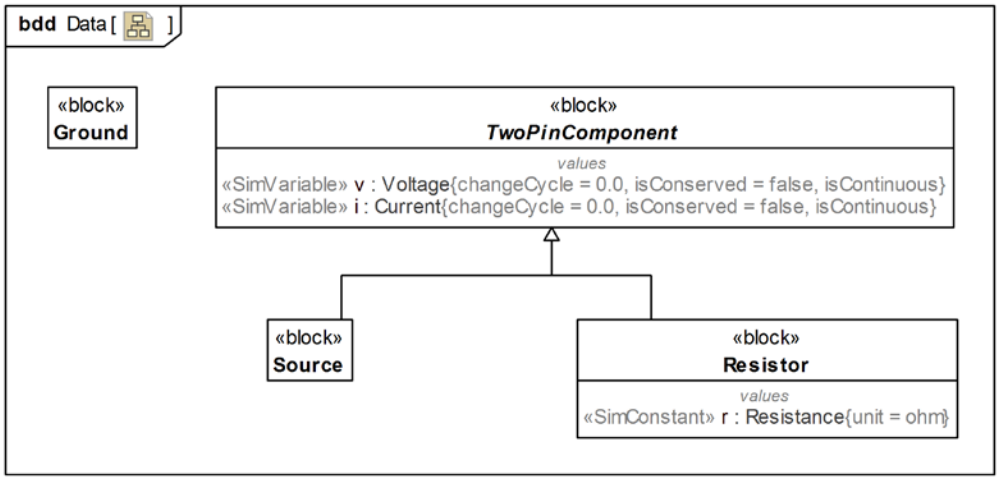

Figure 25: Component properties

A.6 Constraint blocks and constraints (equations)

Equations define mathematical relationships between numeric properties. In SysML, equations are represented as constraints in constraint blocks. Parameters of constraint blocks correspond to SimVariables and SimConstant of blocks (*i*, *v*, *r* in this example), as well as to SimVariables present in the type of the ports (*pv*, *pi*, *nv*, *ni* in this example).

Create an abstract constraint block *TwoPinComponentConstraint* to define parameters and equations common to sources and resistors. The equations should state that the voltage of the component is equal to the difference between the voltage at the positive and negative pin. The current of the component is equal to the current going through the positive pin. The sum of the current going through the two pins must add up to zero (one is the negative of the other). The ground constraint states that the voltage at the ground pin is zero. The source constraint defines the voltage as a sine wave with the current simulation time as parameter. Figure 26 shows what these constraints should look like in a BDD.

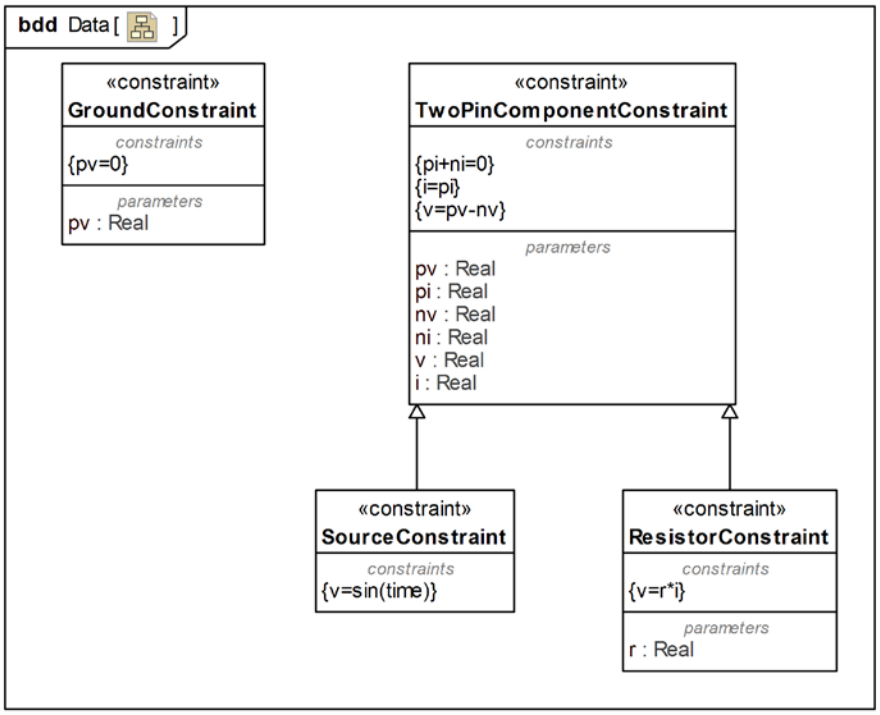

Figure 26: Constraints

A.7 Constraint properties and bindings

The values of constraint parameters are equated to variable and constant values with binding connectors. Create constraint properties on each block (properties typed by constraint blocks) and bind the block variables and constants to the constraint parameters to apply the constraint to the block. Figure 27, Figure 28, and Figure 29 show the bindings for the ground, the source, and the resistor respectively. For the ground constraint, bind gc.*pv* to *p.cflowsim.v* . For the source constraint, bind *sc.pi* to *p.cflowsim.i*, *sc.pv* to *p.cflowsim.v*, *sc.v* to *v*, *sc.i* to *i*, *sc.ni* to *n.cflowsim.i*, and *sc.nv* to *n.cflowsim.v* . For the resistor constraint, bind *rc.pi* to *p.cflowsim.i*, *rc.pv* to *p.cflowsim.v*, *rc.v* to *v*, *rc.i* to *i*, *rc.ni* to *n.cflowsim.i*, *rc.nv* to *n.cflowsim.v*, and *rc.r* to *r*.

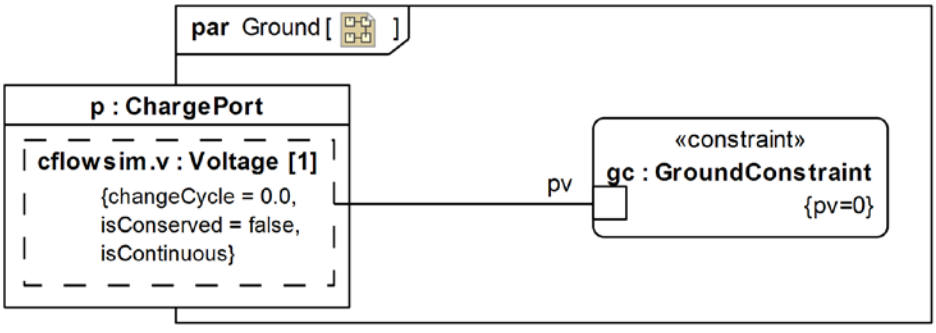

Figure 27: Parametric diagram of the ground

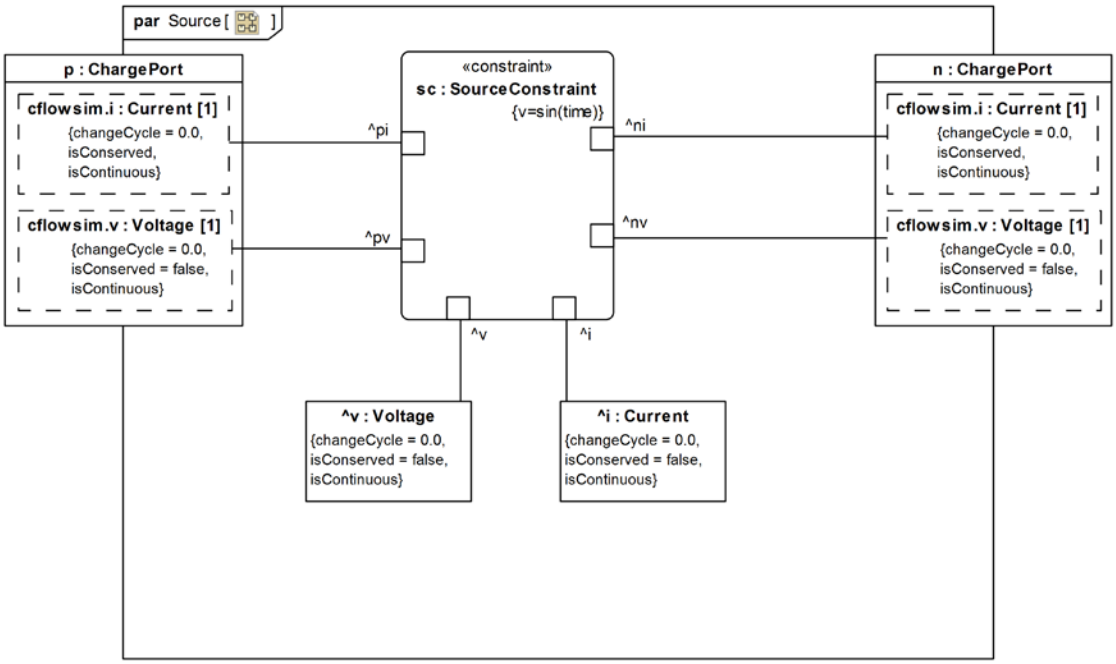

Figure 28: Parametric diagram of the source

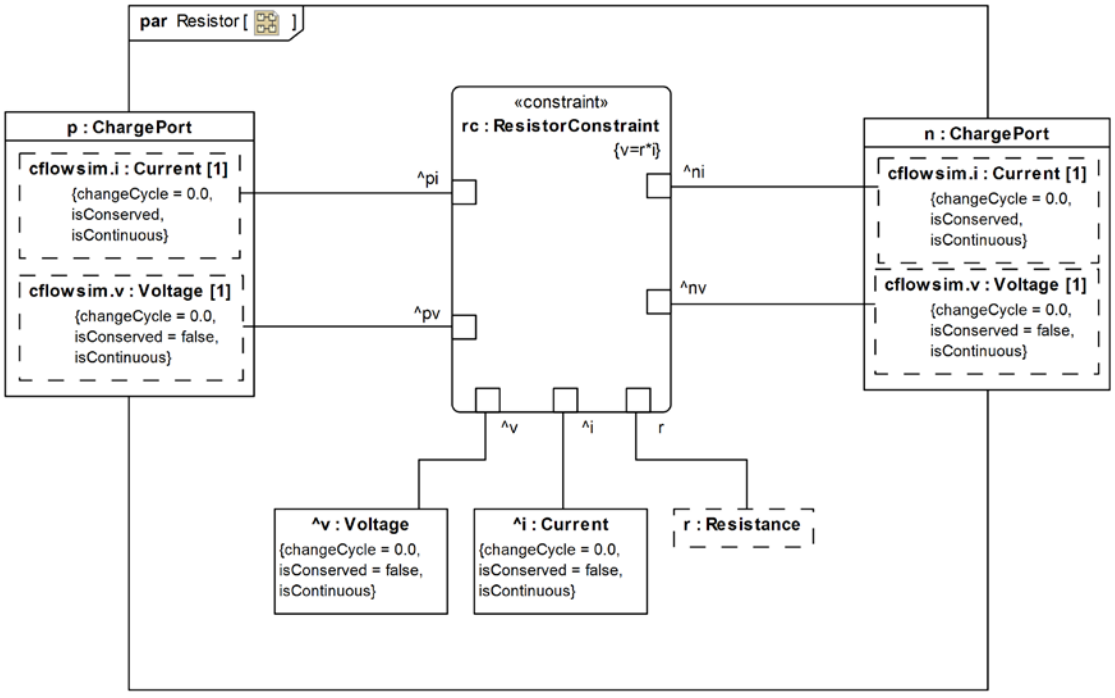

Figure 29: Parametric diagram of the resistor

A.8 Initial values

Set the value for resistance using initial values. Tools might provide a graphical way to do this. Otherwise, create an InstanceSpecification classified by Resistor, give a default value for its resistance, then use the InstanceSpecification as the default value of *r* part in circuit. The resulting diagram is shown in Figure 30

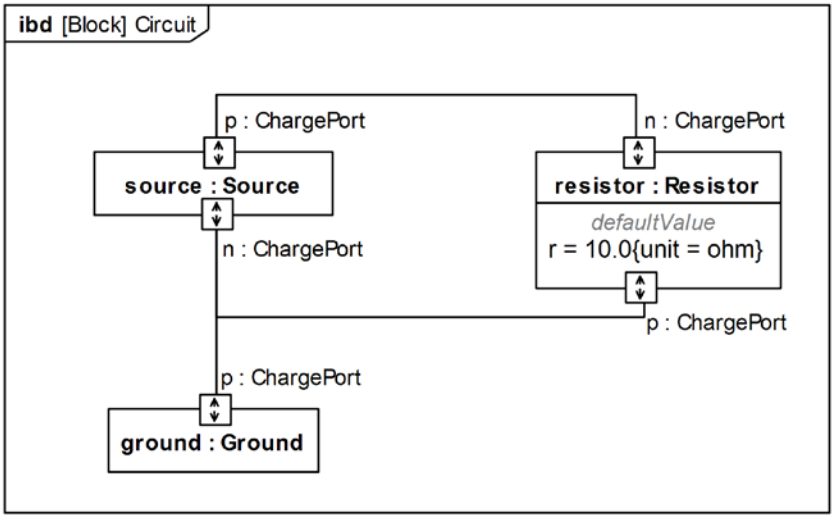

Figure 30: Internal structure of circuit with initial values