OBJECT MANAGEMENT GROUP®

SysML Extension for Physical Interaction and Signal Flow Simulation

Version 1.~~1~~1 – with change bars

OMG Document Number: formal/21-05-04~~3~~ ~~[smsc/21-05-02]~~

Release Date: June 2021

Standard Document URL: http://www.omg.org/spec/SysPhS/1.1/PDF

Machine Consumable Files:

Normative:

https://www.omg.org/spec/SysPhS/20200925/SysPhSProfile.xmi

https://www.omg.org/spec/SysPhS/20200925/SysPhSLibrary.xmi

Informative:

https://www.omg.org/spec/SysPhS/~~20200925~~20200925/SysPhSAnnexA-ElectricCircuit.xmi
https://www.omg.org/spec/SysPhS/2020092~~2~~5/SysPhSAnnexA-SignalProcessor.xmi
https://www.omg.org/spec/SysPhS/2020092~~2~~5/SysPhSAnnexA-Hydraulics.xmi
https://www.omg.org/spec/SysPhS/2020092~~2~~5/SysPhSAnnexA-Humidifier.xmi
https://www.omg.org/spec/SysPhS/2020092~~2~~5/SysPhSAnnexA-CruiseController.xmi

OMG's Issue Reporting Procedure

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

Table of Contents

Preface

About the Object Management Group

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 Meta-model); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *https://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Formal Specifications are available from this URL: *https://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
9C Medway Road, PMB 274
Milford, MA 01757
USA

Tel: +1-781-444-0404
Fax: +1-781-444-0320

Email: *pubs@omg.org*

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

Issues

The reader is encouraged to report and technical or editing issues/problems with this specification to:
https://www.omg.org/report_issue.htm

This page intentionally left blank.

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 model 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 2.5.1~~1~~," http://www.omg.org/spec/UML/2.5.1, December 2017~~7~~.

[2] Object Management Group, "OMG Systems Modeling Language, version 1.6," http://www.omg.org/spec/SysML/1.6, November 2019~~9~~.

[3] Modelica Association, "Modelica® - A Unified Object-Oriented Language for Systems Modeling, Language Specification, version 3.4," http://www.modelica.org/documents/ModelicaSpec34.pdf, April 2017.

[4] Modelica Association, "Modelica Standard Library," https://github.com/modelica/Modelica, April 2016.

[5] International Organization for Standardization, "ISO/IEC 14977:1996 Information technology – Syntactic metalanguage – Extended BNF," http://www.iso.org/standard/26153.html, 1966.

[6] International Organization for Standardization, "ISO 80000-1:2009 Quantities and units -- Part 1: General," http://www.iso.org/standard/30669.html, 2009.

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® Documentation," https://www.mathworks.com/help/releases/R2016a/simulink/, 2016.

[5] The MathWorks, Inc., "Simscape^{TM} Documentation," https://www.mathworks.com/help/releases/R2016a/physmod/simscape/, 2016.

[6] The MathWorks, Inc., “MATLAB® Documentation,” https://www.mathworks.com/help/releases/R2016a/matlab, 2016.

[7] The MathWorks, Inc., “StateFlow® Documentation,” https://www.mathworks.com/help/releases/R2016a/stateflow, 2016.

[8] Bock, C., Barbau, R., Matei, I., Dadfarnia, M., "An Extension of the Systems Modeling Language for Physical Interaction and Signal Flow Simulation", Systems Engineering, vol. 20, no. 5, pp. 395-431, 2017.

[9] Pop, A., Sjölund, M., Asghar, A., Fritzson, P., Casella, F., "Integrated Debugging of Modelica Models,” in *Modeling, Identification and Control,* vol. 35, no. 2, pp. 93-107, 2014.

[10] Dadfarnia, M., Barbau, R., “Platform-Independent Debugging of Physical Interaction and Signal Flow Models,” Proceedings of the 13th Annual IEEE International Systems Conference, 2019.

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 the base classes to which the stereotypes are applied. For example, the phrase “PhSVariable typed by Real” refers to a property that has the PhSVariable stereotype applied and that is typed by Real.

5. Symbols

There are no symbols 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 two kinds of problems:~~.~~

- In signal flow modeling, system components exchange numeric and boolean values in predetermined directions (unidirectionally). For each component, some values will be provided by other components (inputs), and some values will be provided to other components (outputs). Connections between components indicate that values~~s~~ are passed from one output of a source component to one or more inputs of target components. Component behavior is specified by equations that relate input, output, and component variables. Signal flow is well suited for describing control systems and signal-processing systems.
- In physical interaction, system components exchange physical substances that carry energy in directions determined during simulation (possibly bidirectionally). Each exchange is modeled with two numeric values (flow rate and potential to flow of a physical substance, in terms of one of its conserved characteristics), compared to one (possibly boolean) value for signal flow, which does not involve physical substances. In physical interaction, the direction in which substances flow between components is not predetermined, as it is for values in signal flow. Component behavior in physical interaction is specified by equations that relate flow rate, potential, and component variables. 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 physical substances.

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 must be performed prior to translation to simulation platforms. Clause 10 provides translations between extended, preprocessed SysML models 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.

Annex A gives additional examples showing how to use the contents of Clauses 7, 8, and 11. Annex 0 gives an overview of platform-independent debugging procedures for physical interaction and signal flow in SysML models extended with SysPhS. These are illustrated by applying them to an example from Annex A.

6.3 Changes to Adopted OMG Specifications

None.

6.4 Acknowledgements

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 10.6 and 10.7.

7.2 Simulation profile

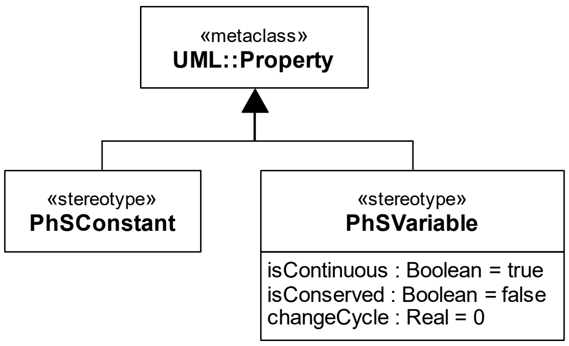


Figure 1: Simulation stereotypes

7.2.1 PhSConstant

Package: SysPhS
isAbstract: No
Extended Metaclass: Property

Description

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

Constraints

[1] Properties stereotyped by PhSConstant must be typed by Real, Integer, or Boolean, or one of their specializations.
[2] Properties stereotyped by PhSConstant must have multiplicity 1, unless they are also stereotyped by MultidimensionalElement (see Subclause 11.5).
[3] Properties stereotyped by PhS Constant must not redefine more than one other property, which must have the same name and type and must be stereotyped by PhSVariable or PhSConstant.

Notation

The stereotype label between guillemets is “phsConstant”.

A compartment with the label “phs constants” may appear as part of a block definition to list the properties stereotyped by PhSConstant. The properties omit the ‘«phsConstant»’ prefix.

7.2.2 PhSVariable

Package: SysPhS
isAbstract: No
Extended Metaclass: Property

Description

A PhSVariable has values that can vary over time in a continuous or discrete fashion. Continuous variables have values that are close to their values at nearby times in the past and future. Discrete variables have values that are the same as their values at nearby times in either the past or future, or both. The effect is that continuous variables vary smoothly over time, including the possibility of remaining constant, while discrete variables are always constant for a period of time, then change instantaneously to a possibly very different value for another period of time. Discrete variables can be restricted to change values only at regular intervals (change cycle greater than zero), though they do not need to change at every interval. Variables being continuous or discrete does not imply any restriction on the range of their values, only the way in which those values change over time.

PhSVariables are used to model exchanges between components (physical interaction and signal flow), as described below, and behavior within components (see Subclause 6.1).

Component interactions are modeled on blocks describing the things that are interacting, rather than on associations between these blocks. The interacting blocks can type parts and ports. PhSVariables and flow properties are used to model component interactions:

- Physical interactions are specified by inout flow properties typed by blocks that characterize substances crossing their boundaries in terms of a conserved characteristic of those substances. For example, electrons passing the boundary of an object are modeled as the flow of charge, rather than electrons. Blocks typing the flow properties (indirectly) specialize ConservedQuantityKind, each named for a physical characteristic (quantity kind) that is conserved in flows between components (see Subclause 11.2.2). The blocks describe flows with two PhSVariables, one conserved and one non-conserved, see below.

- Signal flows are specified by in or out flow properties that are also non-conserved PhSVariables. They are typed by the kind of signal (numeric or boolean).

Connected flow properties are on blocks typing parts or ports that have a connector linking them. *Matching flow properties* are defined in SysML. Physical interactions and signal flows can only occur between connected and matching flow properties that satisfy the constraints in the Constraints section below.

In physical interactions:

- Conserved PhSVariables give the rate at which substances are crossing the boundary of an object (*flow rate*) as a rate of the quantity kind that types the flow property. For example, fluids might cross the boundary of a tank, but the flow rate is given as volume (a quantity kind typing the flow property) per time, regardless of the kind of fluid. When physical flow properties are connected and match, the values of conserved PhSVariables on their types on all ends add up to zero (positive and negative flow rates indicate flows in opposite directions).
- Non-conserved PhSVariables give the potential for substances to cross the boundary (*potential to flow*), whether any substance is crossing or not, as a potential of the same quantity kind used for the paired conserved PhSVariable. For example, fluid might have a high potential to flow at the boundary of a tank, but the potential is in terms of pressure (force per volume surface), whether any fluid is crossing the boundary or not, and regardless of the kind of fluid. When physical flow properties are connected and match, the values of non-conserved PhSVariables on all ends are equal.

In signal flows:

- PhSVariables (that are also flow properties) give a numeric or boolean value crossing the boundary of an object. When signal flow properties are connected and match, their values on all ends are equal (they act like non-conserved PhSVariables).

Component behavior can be defined for blocks that type parts (*component blocks*), not ports. Components might pass physical substances and signals through them, possibly transforming them on the way, or creating, destroying, or storing them. These behaviors are specified with constraints blocks applied to component blocks. The constraints are mathematical equations relating values of:

- PhSVariables for flow properties (*flow variables*, for modeling component interactions above).
- PhSVariables not for flow properties (*component variables*, internal to components, not for modeling component interaction). The idea of conservation (or lack thereof) does not apply to these (because they are not related to interactions with other components), but they are specified as non-conserved.

Constraints on flow variables specify the effect components have on physical substances or signals going through flow properties and might depend on component variables. Component variables might have values giving:

- Potential differences between physical flow properties. These differences must be non-zero for physical substances to flow through a component.
- Rates at which physical substances flow through a component. This differs from flow rates through flow properties when the component creates, destroys, transforms, or stores substances.
- Internal states, such as, how much of a physical substance is currently stored, the temperature of a component, or the current value of a signal integrator.

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 may change.

Constraints

[1] The stereotyped property must be typed by Real, Integer, or Boolean, or one of their specializations.
[2] isContinuous may be true only when the stereotyped property is typed by Real or one of its specializations.
[3] isConserved may be true only when isContinuous is true and the stereotyped property is on a block specialized from ConservedQuantityKind (see Subclause 11.2.2).
[4] changeCycle may be other than 0 only when isContinuous is false.
[5] changeCycle must be positive or 0.
[6] A property stereotyped by PhSVariable must not be stereotyped by PhSConstant.
[7] Properties stereotyped by PhSVariable must have multiplicity 1 unless they are also stereotyped by MultidimensionalElement (see Subclause 11.5).
[8] Flow properties stereotyped by PhSVariable that are connected and matching must have opposite directions (in/out or out/in), the same type and multiplicity, and the same value for isContinuous on the applied stereotype.
[9] Flow properties stereotyped by PhSVariable that have in direction may connect to and match no more than one other flow property stereotyped by PhSVariable.
[10] A property stereotyped PhSVariable can redefine at most one other property and it must have the same name and type and must be stereotyped by PhSVariable.
[11] When a property stereotyped by PhSVariable with isContinuous=true redefines another property, the PhSVariable applied to the redefined property must have isContinuous=true.
[12] When a property stereotyped by PhSVariable with isContinuous=false redefines another property stereotyped by PhSVariable with isContinuous=false, the redefining property's changeCycle must be an integer multiple of the redefined property's changeCycle.

Notation

The stereotype label between guillemets is "phsVariable".

A compartment with the label "phs variables" may appear as part of a block definition to list the properties stereotyped by PhSVariable. The properties omit the "«phsVariable»" prefix.

A compartment with the label "physical interactions" may appear as part of a block definition to list flow properties typed by a block specialized from ConservedQuantityKind that has one conserved and one non-conserved PhSVariable (see Subclause 11.2.2).

A compartment with the label "signal flows" may appear as part of a block definition to list flow properties that have PhSVariable applied.

This page intentionally left blank.

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 'SysPhS' string as their language.

The SysPhS expression grammar includes a subset of Modelica's grammar, as follows:

- All terminal symbols
- 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*, *relational-operator*, *arithmetic-expression*, *add-operator*, *term*, *mul-operator*, *factor*, *primary*, *name*, *component-reference*, *function-call-args*, *function-arguments*, *function-argumentsnon-first*, *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 SysPhS expression 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 10.13).

The following non-terminal symbol is included in the SysPhS expression 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 SysPhS 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).

This page intentionally left blank.

9. Preprocessing SysML Models

9.1 Introduction

This clause defines processing of SysML models performed prior to translation to simulation platforms per Clause 10, to enable translations of SysML modeling patterns not covered in Clause 10. Subclause 9.2 covers associations blocks. Subclauses 9.3 through 9.5 address flow property and connector patterns. Preprocessing should be performed in the order of the subclauses below. In these subclauses, flow properties with PhSVariable applied or typed by blocks (indirectly) specializing ConservedQuantityKind are called *simulation flow properties*.

9.2 Replace connectors typed by association blocks with their internal structure

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 includes association blocks for modeling complex relationships, which are not available in simulation models. Connectors typed by association blocks must be replaced with the internal structure of their association blocks before translation to simulation platforms per the correspondences 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 association block in a SysML block definition diagram on the top, as well as a usage of it in an internal block diagram on the bottom. The top diagram shows an association block *FrictionAssociation* relating *Flanges*. The internal structure of *FrictionAssociation* has a part typed by *Friction* with two ports, each connected to a participant of the association. The lower diagram shows a connector typed by the association block between the flange of a mass and the flange of a ground. The connector has a connector property typed by *FrictionAssociation*.

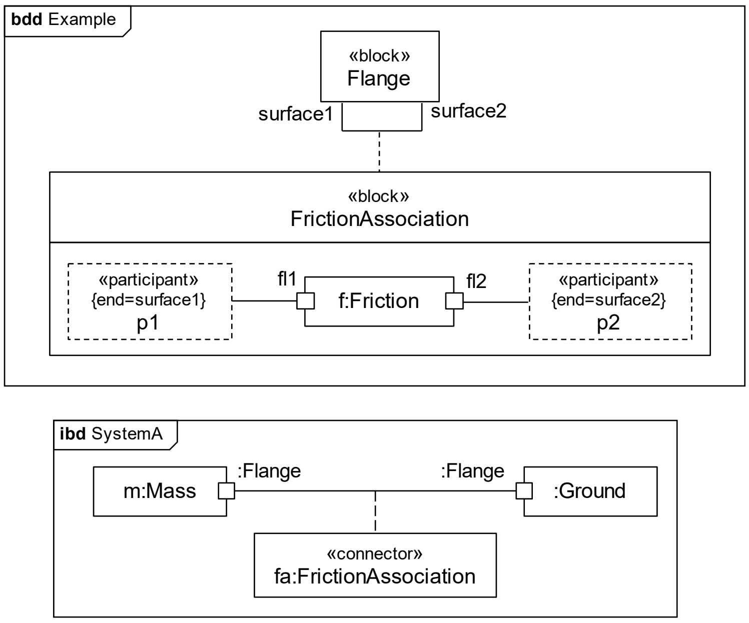


Figure 2: Association block with internal structure and connector properties in SysML

9.2.3 SysML model after processing

Connectors typed by association blocks, including their connector properties, are replaced by the internal structure of the association blocks. Figure 3 shows the content of Figure 2 after processing. The connector and its property *fa* in Figure 2 is replaced by the content of the association block *FrictionAssociation* (the connector and its 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. The block definition diagram in Figure 2 is not changed.

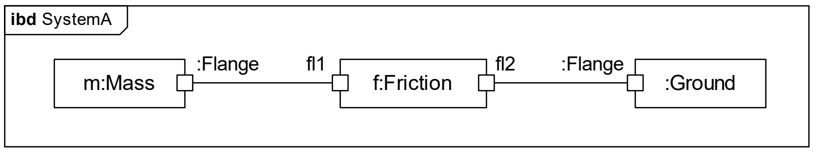


Figure 3: Association block with internal structure and connector properties in SysML

9.3 Non-simulation ports changed to parts

9.3.1 Purpose

SysML supports blocks typing ports that have other properties beside simulation flow properties, but simulation models do not. These ports must be changed into parts before translation to simulation platforms per Clause 10.

9.3.2 SysML model before processing

Figure 4 shows a port of type *Wheel*, which has a property *radius* that is not a simulation flow property.

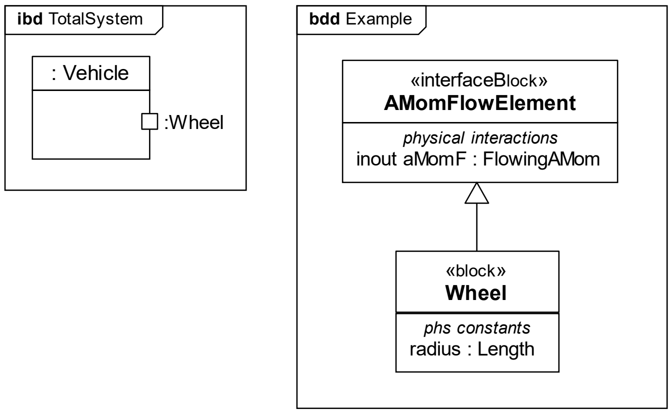


Figure 4: Association block with internal structure and connector properties in SysML

9.3.3 SysML model after processing

Ports typed by blocks that have other properties besides simulation flow properties (owned or inherited) are changed to regular parts. Figure 5 changes the port typed by *Wheel* in Figure 4 to a part. The property is not changed in any other way in this step, including connectors to it (external connectors to the property are addressed in later processing). The block definition diagram in Figure 4 is not changed.

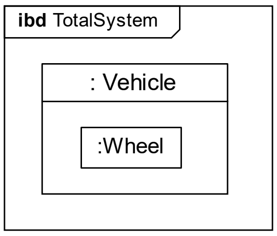


Figure 5: Association block with internal structure and connector properties in SysML

9.4 Separate blocks owning simulation flow properties, and typing parts and ports

9.4.1 Purpose

SysML blocks can have multiple flow properties on part and port types, but simulation models have flows only on port types, and only one per port for the correspondences in Clause 10. SysML blocks typing parts and ports can be the same or share properties by generalization, but simulation models use separate types for parts and ports. SysML connectors can link parts, but simulation models only link ports. Before translation to simulation platforms per Clause 10, SysML parts must be typed by blocks that have no simulation flow properties (owned or inherited), while ports must be typed by blocks owning exactly one simulation flow property and no others (owned or inherited), and connectors must only link ports.

9.4.2 SysML model before processing

Figure 6 shows an example that will be used to illustrate the processing steps in Subclause 9.4.3. *Block1* has two simulation flow properties (*sfp0* and *sfp1*), a PhSVariable (*sv*), and a port of type *Block2* (*p*). *Block2* has two simulation flow properties (*sfp2* and *sfp3*).

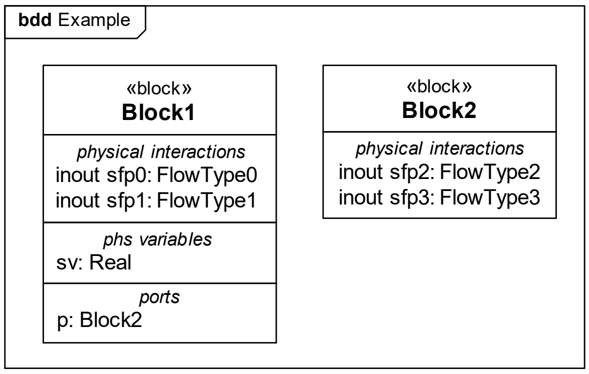


Figure 6: Association block with internal structure and connector properties in SysML

9.4.3 SysML model after processing

The model in Figure 6 is processed in six steps.

9.4.3.1 Move simulation flow properties to their own blocks

Simulation flow properties owned by blocks that also have non-simulation flow properties (owned or inherited) are moved to a new block and a generalization is added between the original block to the new one. The same is done for blocks that own multiple simulation flow properties and no other properties, except that one of the simulation flow properties remains on the original block. Figure 7 shows how simulation flow properties are moved from the blocks in Figure 6. The two simulation flow properties of *Block1* (*sfp0* and *sfp1*) are moved to separate blocks (*Sfp0Type* and *Sfp1Type*), both generalizing *Block1*. In *Block2*, the first simulation flow property (*sfp2*) is left in the block, while the second (*sfp3*) is moved to a new block (*Sfp3Type*) generalizing *Block2*.

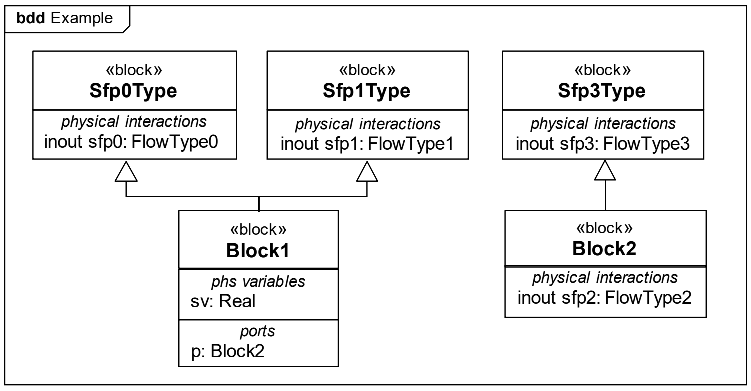


Figure 7: Association block with internal structure and connector properties in SysML

9.4.3.2 Add ports for simulation flow properties inherited to blocks that have non-simulation flow properties

Ports are added to blocks that have non-simulation flow properties (owned or inherited) for each simulation flow property that is inherited directly from a block that owns it, such as those added in Subclause 9.4.3.1. The port type is the block that owns the inherited simulation flow property. In Figure 7, *Block1* has non-simulation flow properties, as well as two simulation flow properties inherited directly from blocks that own them (*sfp0* and *sfp1*, inherited from *Sfp0Type* and *Sfp1Type*, respectively). Figure 8 adds two ports to *Block1* (*psfp0* and *psfp1*), typed by the two general blocks. *Block2* in Figure 7 is not changed, because it does not have non-simulation flow properties.

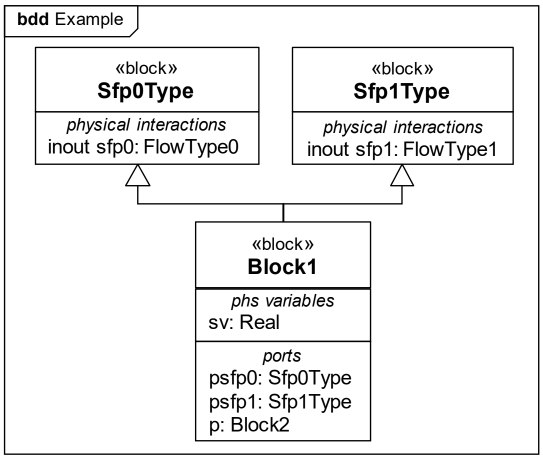


Figure 8: Port added to block that has non-simulation flow properties and inherits simulation flow properties in Figure 7

9.4.3.3 Split up ports typed by blocks that have inherited simulation flow properties

Ports are added for each simulation flow property that is inherited to a block's port types. The new ports are typed by the block owning the inherited simulation flow property. In Figure 8, *Block1* has a port typed by *Block2*, which has a simulation flow property inherited from *Sp3Type* (*sfp3*, see Figure 7). Figure 9 adds a new port to *Block1* (*psfp3*) typed by *Sp3Type*, because of that inherited property.

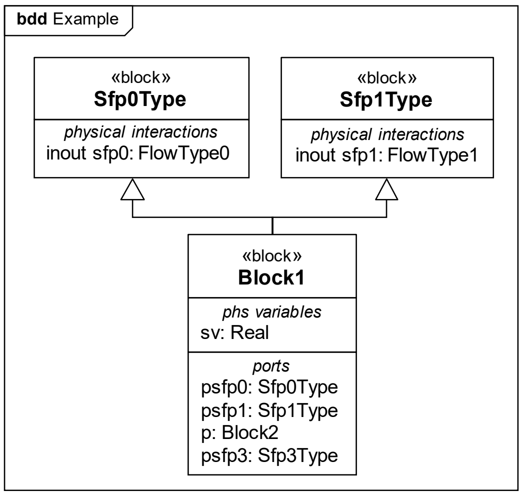


Figure 9: Port added alongside port typed by block with multiple simulation flow properties in Figure 8

9.4.3.4 Relink binding connectors that involve simulation flow properties moved to added ports

Binding connectors involving simulation flow properties that are moved to ports added in Subclauses 9.4.3.2 and 9.4.3.3 are relinked to their new locations. Specifically, after the processing in Subclause 9.4.3.1, binding connectors linked to, or through property paths containing, a simulation flow property inherited to a block that has non-simulation flow properties (owned or inherited) are relinked through the ports added in Subclause 9.4.3.2. Similarly, binding connectors linked to, or through property paths containing, simulation flow properties on blocks typing ports with multiple simulation flow properties are relinked through the ports added in Subclause 9.4.3.3. Figure 10 shows binding connectors before processing linked through simulation flow properties inherited to *Block1* (*sfp0* and *sfp1*), and bindings connectors linked through simulation flow properties of *Block2* (*p.sfp2* and *p.sfp3*). Figure 11 relinks these bindings through the ports added in Subclauses 9.4.3.2 and 9.4.3.3 (*psfp0.sfp0*, *psfp1.sfp1*, and *psfp3.sfp3*).

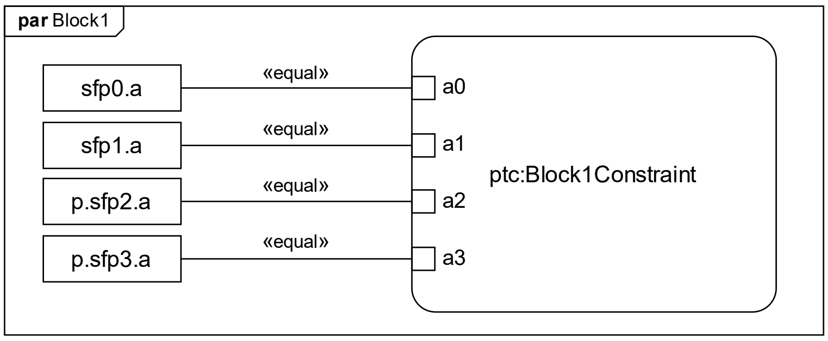


Figure 10: Bindings involving simulation flow properties before processing

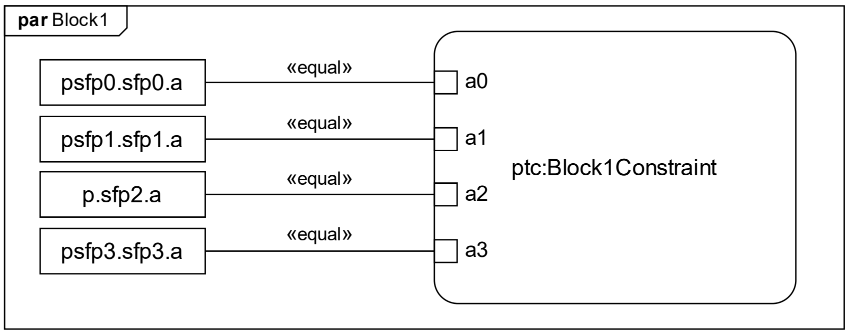


Figure 11: Bindings in Figure 10 relinked through ports added in subclauses 9.4.3.2 and 9.4.3.3

9.4.3.5 Replace or add connectors between properties typed by blocks that have simulation flow properties moved to added ports

Connectors to parts or ports typed by blocks that inherit simulation flow properties that are moved to ports added in Subclauses 9.4.3.2 and 9.4.3.3 are replaced by connectors to their new locations. Specifically, after the processing in Subclause 9.4.3.1, connectors to parts typed by blocks that inherit simulation flow properties are replaced by connectors to the ports added for these simulation flow properties in Subclause 9.4.3.2. Connectors are added linking the ports added for ports with multiple simulation flow properties in Subclause 9.4.3.3. In both cases, connectors are replaced or added only if the other end will have a matching simulation flow property (see Subclause 7.2.4), otherwise the connectors are deleted (this occurs if some of the simulation flow properties do not match before processing). Figure 12 shows two parts typed by *Block1* in Figure 6, before processing. A connector links the parts, and a second connector links their ports. Figure 13 replaces the first connector by two connectors between the ports *psfp0* and *psfp1*, respectively, added due to the inherited simulation flow properties fsp0 and fsp1, respectively. The figure also adds a connector between the ports added for the simulation flow property *psfp3* inherited to *Block2*.

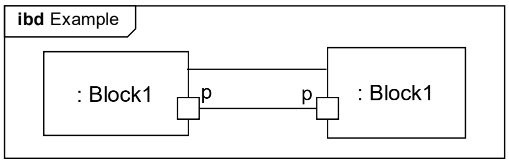


Figure 12: Connectors between parts and ports from Figure 6 before processing

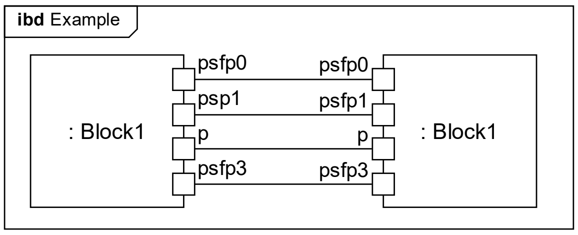


Figure 13: Connectors in Figure 12 replaced or added between ports added in Subclauses 9.4.3.2 and 9.4.3.3

9.4.3.6 Removing generalizations to blocks owning simulation flow properties

Now that all the port types needed for simulation have been created, some generalizations to blocks dedicated to simulation flow properties need be removed.

Generalizations to blocks that own simulation flow properties are removed unless the inherited properties are redefined in the special block. Figure 14 removes the generalizations in Figure 9 and Figure 7.

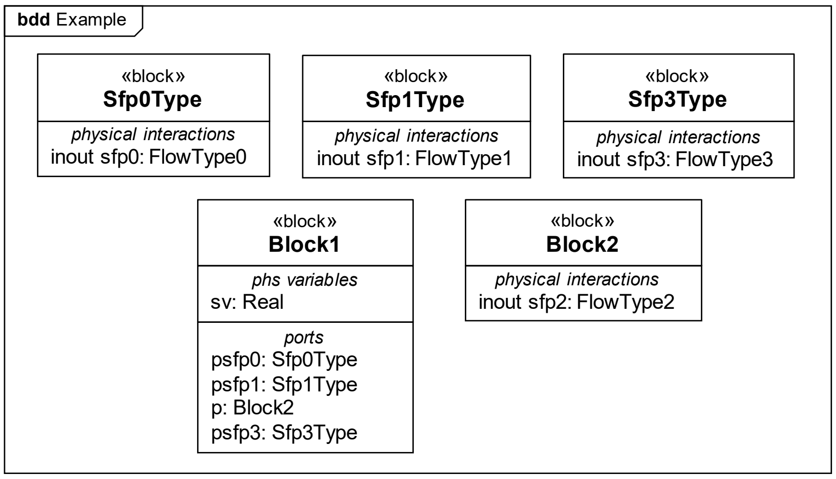


Figure 14: Generalizations in Figure 9 and Figure 7 removed

9.5 Reduce nesting of connector ends

9.5.1 Purpose

SysML supports connectors that link ports reached from the block owning the connector through a chain of other properties (property path), but some simulation models can only link ports reached through one property. These SysML connectors must be split up to link ports reached through only one property before translation to simulation platforms per Clause 10.

9.5.2 SysML model before processing

Figure 15 shows a connector linking a port (*z*) reached through a chain of two other properties (x and y). The length of the nested connector end property path at that end is 2.

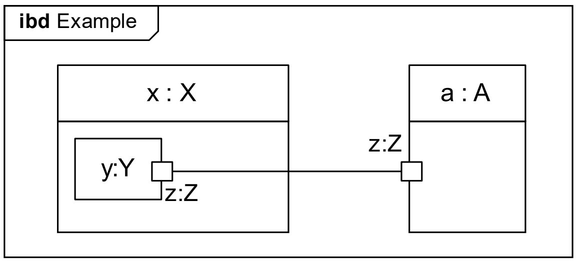


Figure 15: Connector linking port reached through two other properties

9.5.3 SysML model after processing

Connectors that link ports reached from the owner of the connector through a chain of other properties (SysML nested connector end property paths longer than 1) are relinked to an added intermediate port, and a connector added from that port (reducing the property path length to 1). Figure 16 adds a proxy port to x's type with the same type as *z*, and the connector in Figure 15 is relinked to the added port. A binding connector is added in *x*'s type between the new port and the original end of the connector. This procedure is repeated until connectors only link ports reached from the block owning the connector through one property.

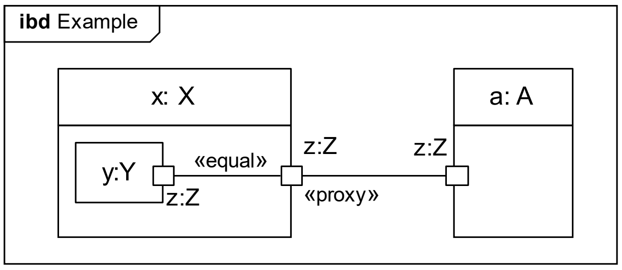


Figure 16: Connector in Figure 15 split by adding a proxy port and another connector

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:

- Many SysML capabilities are not supported on simulation platforms (some of these are supported by transforming SysML models before translation, see Clause 9).
- 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 10.12 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 10.13 covers translations for the expression language in Clause 8.

10.2 Root element

10.2.1 Purpose

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

10.2.2 SysML modeling

SysML root elements are packages, which are containers for model elements. Figure 17 shows a package *P* owning a block *B*.

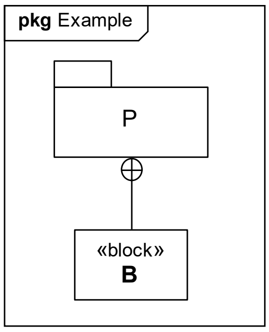


Figure 17: Package and model in SysML

10.2.3 Modelica modeling

SysML packages correspond to Modelica models defined as the root element of a file.

The following Modelica code corresponds to Figure 17. It has a model P owning a model B (see Subclause 10.3.3).

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

10.2.4 Simulink modeling

A SysML package corresponds to a Simulink library paired with a model, defined as root elements of separate files. The model is executed during simulation, referencing blocks defined in the library (see Subclause 10.3.4 about defining and referencing Simulink blocks). Only Simulink blocks defined in libraries can be referenced (reused), either by a library or a model. Models link together references to library blocks, corresponding to SysML connectors between parts (see Subclause 10.8.4).

The following Simulink codes in separate files correspond to Figure 17. The first has a library *P* and the second a model *M* (the names only appear in the file names). Both include a system, which the library uses to define a reusable block *B*.

```
<Library>
 <System>
   <Block Name="B">
   </Block>
 </System>
</Library>
```

```
<Model>
  <System>
  </System>
</Model>
```

10.2.5 Simscape modeling

SysML packages correspond to Simscape libraries compiled from directories of files with code corresponding to the elements in the package. Simscape files each contain a single element (see Subclauses 10.2.5 and 10.7.10) and are stored in directories named for the Simulink library that will contain the elements after the directory is compiled (the library is not specified in the files, there is no Simscape language element for it corresponding to SysML packages).

The package *P* in Figure 17 corresponds to a directory with “P” in its name. The directory has a file containing Simscape code corresponding to block B (see Subclause 10.3.5).

10.2.6 Summary

SysML	Modelica	Simulink	Simscape
Package	Model	Library and Model, each containing a system	Library (compiled from directory of element files)
Element owned by package	Element in model	Element in system	Element in library (compiled from element file)

10.3 Blocks and properties

10.3.1 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.

10.3.2 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 18 shows a SysML block *A* that contain one part *b1* of type *B*. *B* is also a SysML block.

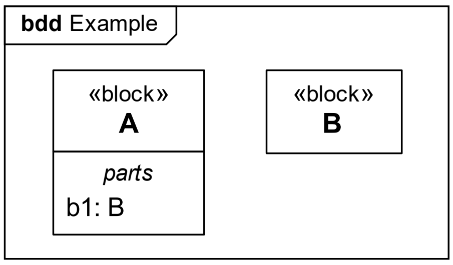


Figure 18: Block and part in SysML

10.3.3 Modelica modeling

Modelica is a human-readable textual language for physical interaction and signal flow modeling. It is class-oriented, like SysML, but with different terminology. Modelica includes various kinds of classes, four of which are used in this specification: models (corresponding to SysML blocks that do not type ports, see below, and to SysML packages, see Subclause 10.3.3), connectors (for physical interaction, see Subclause 10.7.8), types (for SysML value types, see Subclause 10.11.3) and blocks (for SysML state machines, see Subclause 10.12.3). SysML properties correspond to Modelica components.

The following Modelica example corresponds to the SysML block *A* in Figure 18. It has a Modelica model *A* corresponding to the SysML block *A*, with a component *b1* typed by Modelica model *B*, corresponding to the SysML property *b1* typed by block *B*.

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

It has a model A corresponding to the SysML block *A*, with a component *b1* typed by Modelica model *B*, corresponding to the SysML property *b1* typed by block *B*.

10.3.4 Simulink modeling

Simulink is a graphical language for signal flow modeling that has XML-based file format and an extension for physical interaction modeling (see Subclause 10.2.5). It is class-oriented to some extent, though not as much as the other simulation platforms used in this specification. Simulink has an abstraction called blocks that has many specializations, five of which are used in this specification: subsystems (corresponding to SysML blocks, see below), references (corresponding to SysML parts, see below), inports and outports (corresponding to SysML ports with in and out flow properties, see Subclause 10.7.5), and S-functions (corresponding to SysML constraint blocks, see Subclause 10.9.5). When used as a container, structural features are contained in a Simulink system. Simulink blocks are identified by an integer (SID) that is unique within its model or library. A SysML block and its parts correspond a Simulink block with a system containing blocks referencing other blocks (see Subclauses 10.4.4 and 10.5.4 about inherited features).

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 included), or to Simscape components (when Simscape is included).

The following example shows Simulink code corresponding to Figure 18. 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* from the same library *Example* (see Subclause 10.2.4 about libraries).

```
<Block BlockType="SubSystem" Name="A" SID="1">
  <System>
    <Block BlockType="Reference" Name="b1" SID="2">
      <P Name="Ports">[0,0]</P>
      <P Name="SourceBlock">Example/B</P>
    </Block>
</System>
</Block>
<Block BlockType="SubSystem" Name="B" SID="3">
  <System>
  </System>
</Block>
```

10.3.5 Simscape modeling

SysML parts correspond to Simscape member components (see Subclauses 10.4.5 and 10.5.5 about inherited features).

The following example shows Simscape code corresponding to blocks A and B in Figure 18. It has a component *A* containing a member component *b1* of type *B* from the same library *Example* (see Subclause 10.2.4 about libraries).

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

component B
end
```

10.3.6 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 Simulink code corresponds to block *A* in Figure 18. It has a subsystem block *A*, with a system that contains a reference *b* to the Simscape component *B*, (defined in Subclause 10.3.5), from the library *Example* (see Subclause 10.2.4 about libraries).

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

10.3.7 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

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.

Figure 19 shows a block *A* with a property *c1* of type *C*, and a block *B* that is a specialization of that block *A*.

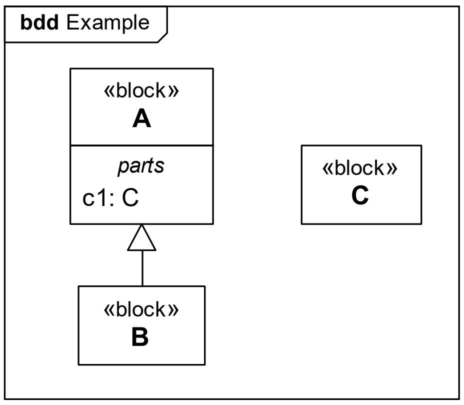


Figure 19: Generalization in SysML

10.4.3 Modelica modeling

SysML generalization corresponds to Modelica class extension, including multiple extensions of the same class.

The following Modelica code corresponds to Figure 19. 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 other blocks). Inherited features that are not redefined in SysML (see Subclause 10.5) correspond to newly defined (uninherited) features in Simulink blocks.

The following Simulink code corresponds to Figure 19. It has blocks *A* and *B*, each with a system containing a block *c1* that references block *C*. 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">Example/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">Example/C</P>
    </Block>
  </System>
</Block>
```

10.4.5 Simscape modeling

Simscape supports single generalization of components. SysML generalization corresponds to Simscape superclassing when the special SysML block has only one generalization and does not redefine any properties (see Subclause 10.5), otherwise, SysML generalization has no correspondence in Simscape, and inherited properties in SysML that are not redefined correspond to new (uninherited) component members in Simscape.

The following Simscape code corresponds to Figure 19. It has a component *A* with a member component *c1* typed by *C*, and the component *B* generalized by *A*.

```
component A
  nodes
    c1 = Example.C;
  end
end
component B < Example.A
end
```

10.4.6 Summary

SysML	Modelica	Simulink	Simscape
Generalization	Extend clause	N/A	Subclassing, when the special SysML block has only one generalization and does not redefine properties, otherwise, N/A.
Inherited features	Inherited components	Newly defined (uninherited) features	Inherited member components when the special SysML block has only one generalization and does not redefine properties, otherwise, new (uninherited) member components.

10.5 Property redefinition

10.5.1 Purpose

Classes that inherit features in systems and simulation models (see Subclause 10.4) can alter those features. For example, they can change the type of an inherited feature to a specialization of that type.

10.5.2 SysML modeling

In SysML, blocks can alter inherited properties by redefinition. Figure 20 shows a block *A* with a property *c1* of type *C*, and a block *B* specializing block *A*. *B* has a property c1 that redefines *C::c1* to be typed by *D*, a specialization of *C*.

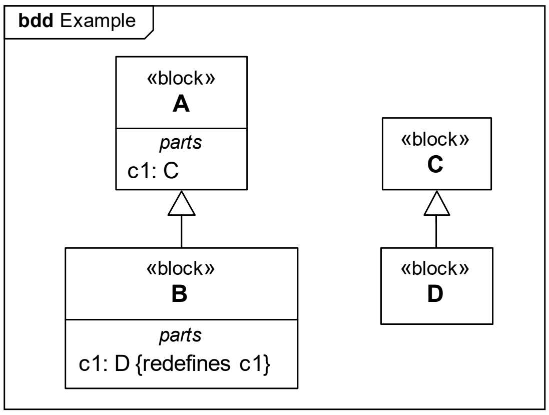


Figure 20: Property redefinition in SysML

10.5.3 Modelica modeling

Modelica supports alteration of inherited properties as SysML does, except that the property name cannot be changed. SysML redefined and redefining properties correspond to Modelica replaceable and redeclare components, respectively.

The following Modelica code corresponds to Figure 20. It has a model *A* with component *c1* indicated as replaceable, and a model *B* extending *A* with a component of the same name redeclaring it to alter the type (compare to Subclause 10.4.3).

```
model A
  replaceable C c1;
end A;

model B
  extends A;
  redeclare D c1;
end B;
```

10.5.4 Simulink modeling

Simulink does not support redefinition because it does not support generalization (see Subclause 10.4.4). The effect of SysML redefinition can be achieved by using Simulink correspondences for properties (see Subclause 10.2.4) that redefine inherited ones (see Subclause 10.4.4 about inherited properties that are not redefined).

The following Simulink code corresponds to Figure 20. It has block *A* and *B*, each with a system containing a block *c1*, one referencing block *C* and the other block *D* (compare to Subclause 10.4.4).

```
<Block BlockType="SubSystem" Name="A" SID="1">
  <System>
    <Block BlockType="Reference" Name="c1" SID="2">
      <P Name="Ports">[0,0]</P>
      <P Name="SourceBlock">Example/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">Example/D</P>
    </Block>
  </System>
</Block>
```

10.5.5 Simscape modeling

Simscape supports generalization (single, see Subclause 10.4.5), but not redefinition. The effect of SysML redefinition can be achieved by using Simscape correspondences for multiple generalization or inherited SysML properties that are redefined (see Subclause 10.4.5) and including correspondences for properties (see Subclause 10.2.5) that redefine inherited properties.

The following Simscape code corresponds to Figure 20. It has component *A* and *B*, each with a member component *c1*, one typed by component *C* and the other by *D* (compare to Subclause 10.4.5).

```
component A
  components
    c1 = Example.C;
  end
end

component B
  components
    c1 = Example.D;
  end
end
```

10.5.6 Summary

SysML	Modelica	Simulink	Simscape
Redefined property	Replaceable component	N/A	N/A
Property that redefines inherited property of the same name	Redeclare component	Reference, inport, outport, or connection block	Member component, variable, parameter, input, output, or node

10.6 PhSVariables and PhSConstants

10.6.1 Purpose

Simulation modeling specifies how numeric and boolean variable values can change in more detail than system models. 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.

10.6.2 SysML modeling

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

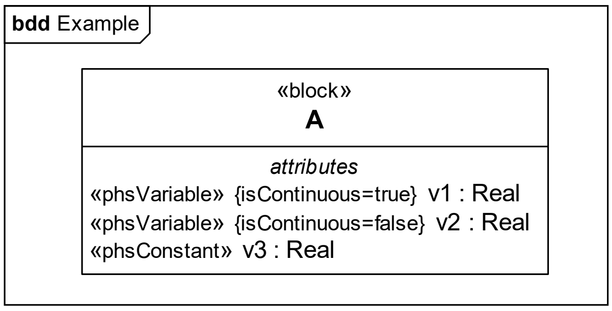

bdd Example
«block»
A
attributes
«phsVariable» {isContinuous=true} v1 : Real
«phsVariable» {isContinuous=false} v2 : Real
«phsConstant» v3 : Real

Figure 21: PhSVariable and PhSConstant in SysML

Figure 21 shows a block *A* with three properties: one continuous PhSVariable *v1*, one discrete PhSVariable *v2*, and one PhSConstant *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.

10.6.3 Modelica modeling

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

The following Modelica code corresponds to Figure 21. 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
```

10.6.4 Simulink modeling

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

10.6.5 Simscape modeling

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

The following Simscape code corresponds to Figure 21. 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
```

10.6.6 Summary

SysML	Modelica	Simulink	Simscape
Property stereotyped by PhSVariable, with isContinuous=true	Continuous component	N/A	Variable
Property stereotyped by PhSVariable, with isContinuous=false	Discrete component	N/A	N/A
Property stereotyped by PhSConstant	Parameter component	N/A	Parameter
Property type	Component type	N/A	Member type

10.7 Ports and Flow Properties

10.7.1 Purpose

Systems and simulation modeling describe interactions between system components. These interactions include exchanges of 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.

10.7.2 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. This correspondence assumes connectors are only between ports (see Subclause 9.4.2 about connectors between parts). Ports describe flows through them using flow properties, which specify the kind of things that flow by their type, as well as the direction of flow (in/out/inout).

The extension for simulation in Subclause 7.2 adds information to flow properties needed for simulation, in particular, flow rates and potentials to flow (conserved and non-conserved PhSVariables, respectively). Physical interaction uses both of these, while signal flow has semantics equivalent to potential to flow. PhSVariables for physical interactions are on blocks specialized from ConservedQuantityKind (see Subclause 11.2.2) typing flow properties. PhSVariables for signals are flow properties (a property with two stereotypes applied) that have a numeric or boolean type specifying the kind of signal.

Subclauses 10.7.3 through 10.7.6 cover signal flow modeling in SysML and simulation platforms, while Subclauses 10.7.7 through 10.7.10 cover physical interaction modeling.

10.7.3 SysML modeling, signal flow

When modeling signal flow, flow properties on port types must be:

- Stereotyped by a non-conserved PhSVariable.
- Typed by Real, Integer, Boolean, or one of their specializations.
- Either in or out.

Figure 22 shows an example signal flow application. The block *Spring* has two ports *u* and *y*, of type *RealInSignalElement* and *RealOutSignalElement* from the signal flow library (see Subclause 11.2.1), respectively. *RealInSignalElement* has an in flow property *rsig*, while *RealOutSignalElement* has the same property with an out direction.

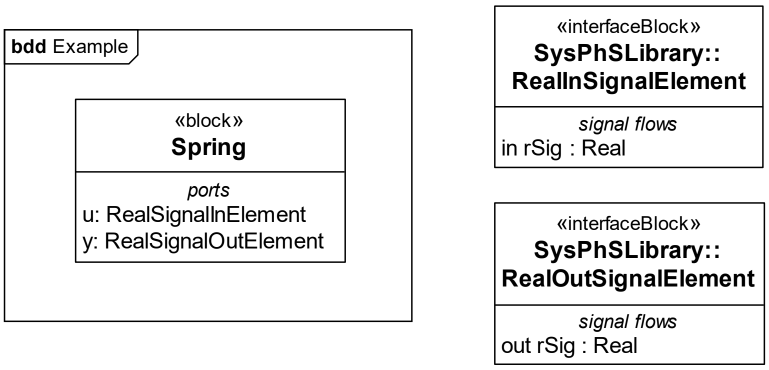


Figure 22: Ports for signal flow in SysML

10.7.4 Modelica modeling, signal flow

SysML ports with a type containing a flow property stereotyped by a non-conserved PhSVariable and typed by Real, Integer, or Boolean, or one of their specializations, correspond to Modelica components typed by the same data type. SysML flow properties have no corresponding constructs in Modelica, but the Modelica component corresponding to the SysML port has a direction given by the flow property.

The following Modelica code corresponds to Figure 22. 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;
```

10.7.5 Simulink modeling, signal flow

Simulink has several kinds of ports, three of which are used in this specification: inports, outports (for signal flow, corresponding to SysML ports typed by blocks with in or out flow properties that have PhSVariable applied, respectively, see below), and connection ports (for physical interaction, see Subclause 10.7.9). Simulink block definitions contain an array giving the number of each kind of port, with connection ports distinguished by whether they appear on the left or right of their blocks in Simulink diagrams. The number of inports and outports are given at the 1st and 2nd positions from the left, respectively, while the number of left and right connection ports are at the 6th and 7th positions, respectively. Trailing series of zeros on the right can be omitted.

SysML ports with a type containing a flow property stereotyped by a non-conserved PhSVariable and typed by Real, Integer, or Boolean, or one of their specializations, correspond to Simulink inports or outports, depending on the direction of the flow property.

The following Simulink code corresponds to Figure 22. It has a block *Spring*, with one inport *u* and one outport *y*. The Ports property of the block gives the port array, showing the number of inports and outports. The Port property of the inport or outport specifies the index of that inport or outport, which must be separately sequential integers for each kind of port, starting with 1.

```
<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>
```

10.7.6 Simscape modeling, signal flow

SysML ports with a type containing a flow property stereotyped by a non-conserved PhSVariable and typed by a Real, Integer, or Boolean, or one of their specializations, correspond to Simscape inputs or outputs, depending on the direction of the flow property.

The following Simscape code corresponds to Figure 22. It has a component *Spring*, with one input *u* and one output *y*, specifying that they should appear on the left and right sides of blocks referencing the spring in Simulink diagrams, respectively (see Subclauses 10.8.5 and 10.8.6). Left or right positioning does not restrict how components can be connected.

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

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

10.7.7 SysML modeling, physical interaction

When modeling physical interaction, flow properties of port types must be inout. This flow property must be typed by a block (indirectly) specializing ConservedQuantityKind (see Subclause 11.2.2), which contains conserved and non-conserved PhSVariables (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 *lMo* typed by *FlowingLMom* from the physical interaction library (see Subclause 11.2.2), which has one conserved PhSVariable *f* and one non-conserved PhSVariable *lV*.

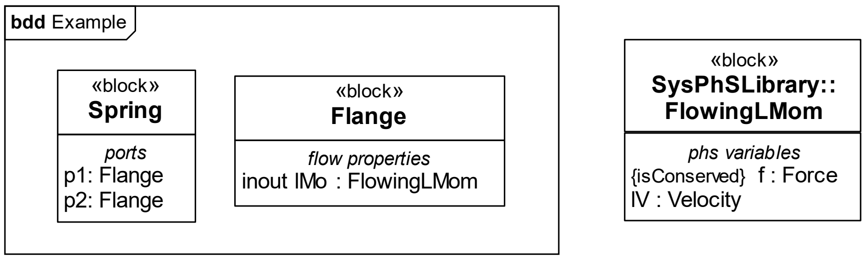


Figure 23: Ports for physical interaction in SysML

10.7.8 Modelica modeling, physical interaction

SysML ports with a type containing a flow property typed by a block (indirectly) specializing ConservedQuantityKind (see Subclause 11.2.2) correspond to Modelica components that have no direction specified, and SysML port types correspond to Modelica connectors. SysML flow properties have no corresponding constructs in Modelica, but PhSVariables on conserved quantity kind blocks correspond to Modelica components on connectors. PhSVariables on conserved quantity kind blocks correspond to Modelica components. Conserved PhSVariables correspond to Modelica flow components, while non-conserved PhSVariables correspond to regular Modelica components.

The following Modelica code corresponds to Figure 23. 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 *lV*.

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

10.7.9 Simulink modeling, physical interaction

Simulink supports connection ports for representing bidirectional flows, but they must be linked to Simscape nodes (see Subclauses 10.7.10 and 10.8.6).

The following Simulink code corresponds to Figure 11. It has a subsystem block *Spring* with connection ports *p1* and *p2*. Connection ports must be linked to nodes on Simscape components defined in the subsystem block (see Subclause 10.7.5 about left and right annotation and port arrays).

```
<Block BlockType="SubSystem" Name="Spring" SID="3">
  <P Name="Ports">[0, 0, 0, 0, 0, 1, 1]</P>
  <System>
    <Block BlockType="PMIOPort" Name="p1" SID="1">
      <P Name="Port">1</P>
      <P Name="Side">Left</P>
   </Block>
   <Block BlockType="PMIOPort" Name="p2" SID="2">
      <P Name="Port">2</P>
      <P Name="Side">Right</P>
   </Block>
</System>
</Block>
```

10.7.10 Simscape modeling, physical interaction

Simscape adds support for physical interaction ports to Simulink, called nodes. Nodes are typed by a domain, which corresponds to a SysML port type with an inout flow property typed by a block (indirectly) specializing ConservedQuantityKind (see Subclause 11.2.2). Conserved PhSVariables on these blocks correspond to Simscape balancing variables in domains.

The following Simscape code corresponds to Figure 23. It has a component *Spring*, with two nodes *p1* and *p2* of type *Flange* (Simscape nodes use left and right annotations in the same way inputs and outputs do, see Subclause 10.7.6). Flange is a domain from the package *CurrentLibrary*, with two variables: one non-balancing variable *lV*, and one balancing variable *f*.

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

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

10.7.11 Summary

SysML	Modelica	Simulink	Simscape
Port typed by block with an in flow property stereotyped by a non-conserved PhSVariable and typed by Real, Integer, Boolean or one of their specializations (signal flow)	Component typed by an equivalent data type	Inport	Input variable
Port typed by block with an out flow property stereotyped by a non-conserved PhSVariable and typed by Real, Integer, Boolean or one of their specializations (signal flow)	Component typed by an equivalent data type	Outport	Output variable
Port typed by block with an inout flow property typed by block (indirectly) specializing ConservedQuantityKind (physical interaction)	Component typed by connector	Connection port	Node typed by domain
Block (indirectly) specializing ConservedQuantityKind (physical interaction)	Connector	N/A	Domain
PhSVariables on blocks (indirectly) specializing ConservedQuantityKind (physical interaction)	Components of connector	N/A	Variables of domain

10.8 Connectors

10.8.1 Purpose

A connection between two connection points enables exchange of physical substances or signals between these parts.

10.8.2 SysML modeling

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

Figure 24 shows an example of SysML connectors. It has a block *Example* with two parts *s1* and *s2*, of types *SpringA* and *SpringB* , respectively, defined similarly to *Spring* in Figure 11, Subclause 10.7.7. The blocks *SpringA and SpringB* have two ports, *p1* and *p2* of type *Flange*, as defined in Figure 23. The figure shows a connector between the port *p2* of *s1*, and the port *p1* of *s2*.

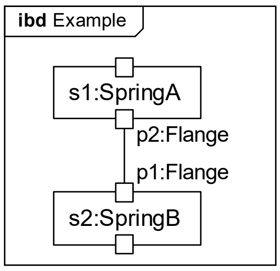


Figure 24: Connectors in SysML

10.8.3 Modelica modeling

SysML connectors correspond to Modelica connect equations, which link components typed by Modelica connectors. This depends on the correspondence between SysML port types and Modelica connectors (see Subclause 10.7.8).

The following Modelica code corresponds to Figure 24. It has a model *Example* with two components *s1* and *s2* of types *SpringA* and *SpringB*, respectively. The models *SpringA* and *SpringB* have two components *p1* and *p2* of type *Flange*, defined similarly to *Spring* in Subclause 10.7.8. *Model* contains a connect equation linking component *p2* of *s1* to component *p1* of *s2*.

```
model Example
  SpringA s1;
  SpringB s2;
equation
  connect(s1.p2, s2.p1);
end Example;
```

10.8.4 Simulink modeling, between blocks with no constraints

SysML connectors correspond to Simulink lines when:

- Simscape is not used with Simulink.
- Simscape is used with Simulink and the SysML connectors are owned by a block with no constraints involving PhSVariables and that link ports on blocks with no constraints involving PhSVariables, such as those in Subclause 11.3, SysML connectors correspond to Simulink lines (see Subclause 10.8.5 and 10.8.6 for other cases when Simscape is used with Simulink).

Simulink lines are directed from outports to inports.

The following Simulink code corresponds to Figure 24, assuming *SpringA* and *SpringB* do not have constraints involving PhSVariables. It has a subsystem block *Example* with two blocks *s1* and *s2* referring to the blocks *SpringA* and *SpringB, respectively,* and having one inport and one outport each, defined similarly to *Spring* in Subclause 10.7.5. A line is defined between the outport port of *s1* (*p2*) and the inport of *s2* (*p1*). Lines identify their end ports by the identifier of the block defining the port, followed by "#" and the kind of port ("in" and "out" for inports and outports, respectively, as shown below, or "lconn" and "rconn" for left and right connection ports, respectively, see Subclause 10.7.5), followed by a colon and the index of the port among those of that kind in the defining block (ports are all ordered).

```
<Block BlockType="SubSystem" Name="Example" SID="1">
  <P Name="Ports">[0,0]</P>
  <System>
    <Block BlockType="Reference" Name="s1" SID="2">
      <P Name="Ports">[1,1]</P>
      <P Name="SourceBlock">Library/SpringA</P>
    </Block>
    <Block BlockType="Reference" Name="s2" SID="3">
      <P Name="Ports">[1,1]</P>
      <P Name="SourceBlock">Library/SpringB</P>
    </Block>
    <Line>
      <P Name="Src">1#out:1</P>
      <P Name="Dst">2#in:1</P>
    </Line>
  </System>
</Block>
```

10.8.5 Simulink modeling, between blocks with constraints

When Simscape is used with Simulink, SysML connectors that are owned by a block with no constraints involving PhSVariables and that link ports on blocks with constraints involving PhSVariables (see Subclause 10.9) correspond to a type of Simulink line called connections.

The following Simulink code correspond to Figure 24, assuming *SpringA* and *SpringB* have constraints involving PhSVariables. It has a subsystem block *Example* with two blocks *s1* and *s2* referring to Simscape components *SpringA* and *SpringB*, respectively, defined similarly to *Spring* in Subclause 10.7.10. The springs have one left port (*p1*) and one right port (*p2*) each, linked by a line of type "Connection" (see Subclause 10.8.4 about defining the ends of lines).

```
<Block BlockType="SubSystem" Name="Example" SID="1">
  <P Name="Ports">[0,0]</P>
  <System>
    <Block BlockType="Reference" Name="s1" SID="2">
      <P Name="Ports">[0,0,0,0,0,1,1]</P>
      <P Name="SourceBlock">Library/SpringA</P>
      <P Name="SourceType">SpringA</P>
      <P Name="SourceFile">Library.SpringA</P>
      <P Name="ComponentPath">Library.SpringA</P>
      <P Name="ClassName">SpringA</P>
    </Block>
    <Block BlockType="Reference" Name="s2" SID="3">
      <P Name="Ports">[0,0,0,0,0,1,1]</P>
      <P Name="SourceBlock">Library/SpringB</P>
      <P Name="SourceType">SpringB</P>
      <P Name="SourceFile">Library.SpringB</P>
      <P Name="ComponentPath">Library.SpringB</P>
      <P Name="ClassName">SpringB</P>
    </Block>
    <Line LineType="Connection">
      <P Name="Src">1#rconn:1</P>
      <P Name="Dst">2#lconn:1</P>
    </Line>
  </System>
</Block>
```

10.8.6 Simulink modeling, between blocks that have constraints and blocks that do not

When Simscape is used with Simulink, SysML connectors that are owned by a block with no constraints involving PhSVariables and that link ports of a block with constraints involving PhSVariables (see Subclause 10.9) to ports of other blocks without constraints involving PhSVariables, such as those in Subclause 11.3, or vice versa, it is necessary to use an additional block between them to convert a regular Simulink signal into a Simscape signal, or vice versa. Specifically, a Simulink connection links a block with constraints (through ports) to or from the converter block, while a Simulink line connects the converter block to or from a block with no constraints.

The following Simulink code connects a Simulink block and a Simscape component, corresponding to Figure 24, assuming *SpringA* does not have constraints involving PhSVariables, while *SpringB* does. The code has a subsystem block *Example* with a block *s1* referring to Simulink block *SpringA* (defined similarly to *Spring* in Subclause 10.7.5), a block *tr1* converting regular signals to physical signals, a block *s2* referring to Simscape component *SpringB* (defined similarly to *Spring* in Subclause 10.7.10), a block *tr2* converting physical signals to regular signals, and a block *s3* also referring to Simulink block *SpringA*. Lines of type *Connection* link *s1*, *tr1*, *s2*, *tr2*, and *s3*.

```
<Block BlockType="SubSystem" Name="Example" SID="1">
  <P Name="Ports">[0,0]</P>
  <System>
    <Block BlockType="Reference" Name="s1" SID="1">
      <P Name="Ports">[1,1]</P>
      <P Name="SourceBlock">Library/SpringA</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">[0,0,0,0,0,0,1,1]</P>
      <P Name="SourceBlock">Library/SpringB</P>
      <P Name="SourceType">SpringB</P>
      <P Name="SourceFile">Library.SpringB</P>
      <P Name="ComponentPath">Library.SpringB</P>
      <P Name="ClassName">SpringB</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/SpringA</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>
 </System>
</Block>
```

10.8.7 Simscape modeling

When Simscape is used with Simulink, SysML connectors owned by a block with constraints involving PhSVariables correspond to Simscape connections.

The following Simscape code corresponds to Figure 24. It has a block *Example* with two components *s1* and *s2* of type *Spring A* and *SpringB*, defined similarly to Spring in Subclause 10.7.10, and a connection between *s1.p2* and *s2.p1*.

```
component Example
  components
    s1=Library.SpringA;
    s2=Library.SpringB;
  end
 connections
 connect(s1.p2, s2.p1);
 end
end
```

10.8.8 Summary

SysML	Modelica	Simulink (without Simscape)	Simulink (with Simscape)	Simscape
Connector between ports with in or out flow properties	Connect equation between components	Line between inport/outports	Connection line between connectors	Connect statement
Connector between ports with inout flow properties	Connect equation between components	N/A	Connection line between connectors	Connect statement

10.9 Blocks with constraints

10.9.1 Purpose

System behavior is represented in simulation models by expressions relating values of system properties. Simulating expressions involves computing an unknown variable from known variables.

10.9.2 SysML modeling

Simulation expressions correspond to constraint blocks in SysML. Constraint blocks are blocks that have parameters and constraint properties (properties typed by constraint blocks). Parameters are properties used in the equations, while constraints are equations.

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 10.9.3 through 10.9.6 cover signal flow modeling, while subclauses 10.9.7 through 10.9.10 cover physical interaction modeling.

10.9.3 SysML modeling, signal flow

Figure 25 shows an example constraint block for a signal flow application, using ports like those defined in Figure 22, Subclause 10.7.3, except in a system containing a spring attached to another object. The block *SpringMassSys* has a SysML constraint property *smsc* typed by *SMSConstraint*. The constraint block has six parameters, each bound to a property reachable from the spring mass system:

- *f* is bound to the signal coming in through port *u*, which has a type with an in flow property *rsig*
- *pos* is bound to the signal going out through port *y*, which has a type with an out flow property *rsig*
- *x* is bound to PhSVariable *position*
- *k* is bound to PhSConstant *springcst*
- *v* is bound to PhSVariable *velocity*
- *m* is bound to PhSConstant *mass*, the mass of the object attached to the spring.

The constraint block defines three constraints representing equations, written in the expression language specified in Clause 8.

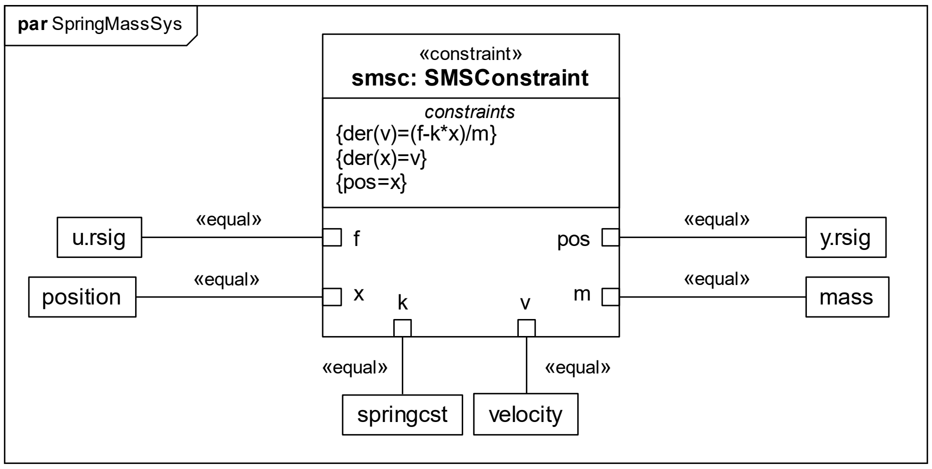


Figure 25: Constraint block for signal flow in SysML

10.9.4 Modelica modeling, signal flow

In a SysML block with constraint properties, the constraints correspond to the same equations in Modelica (assuming the expression language of Clause 8 is used in the constraint block), except the SysML parameters in those constraints correspond in Modelica to the properties they are bound to in SysML.

The following Modelica code corresponds to Figure 25. It has three equations from the constraint block. SysML parameter names are replaced in the Modelica equations according to the bindings in Figure 13: *f* is replaced by *u*, *pos* is replaced by *y*, *x* is replaced by *position*, *k* is replaced by *springcst*, *v* is replaced by *velocity*, *m* is replaced by *mass*.

```
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;
```

10.9.5 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 variables are accessible only inside an S-function (this is different from states in state machines, see Subclause 10.12). 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 corresponds to an output assignment.

SysML parameter names are used as variable names in the S-functions. SysML parameters bound to PhSConstants are replaced in S-functions by the value given for the PhS Constant.

Binding connectors involving ports with in or out flow properties correspond to Simulink lines (see Subclause 10.8.4) linking inports and outports to inputs and outputs of the S-function, respectively.

The following Simulink code corresponds to Figure 25. It has 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 S-function Spring_sc_SpringConstraint has a setup function indicating 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 the same substitutions performed as in Modelica (see Subclause 10.9.4).

```
<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
```

10.9.6 Simscape modeling, signal flow

Simscape supports signal flow by providing a way to specify input and output signals for components. SysML blocks with constraint properties correspond to equations in Simulink components, with the same substitutions as in Modelica (see Subclause 10.9.4). Simscape does not support discrete variables (compare to S-functions, see Subclause 10.9.5).

The following Simscape code corresponds to Figure 25. 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* (see Subclause 10.11.5 about units and Subclause 10.7.6 about left and right annotations). The component has equations connecting these variables: two equations that compute the derivative of the variables, and one that determines the output.

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

10.9.7 SysML modeling, physical interaction

Figure 26 shows an example constraint block for a signal flow application, using the port type defined in Figure 23, Subclause 10.7.7. It has a constraint block *SpringConstraint* with 8 parameters, each bound to a property reachable from the spring:

- Force and velocity at the two ends of the spring (*f1*, *v1*,*f2*, *v2*) are bound to the forces and velocities of conserved quantity kinds flowing through ports *p1* and *p2*, which have types with inout flow properties.
- Change in length of the spring (*x*) is bound to the PhSVariable *lengthchg*.
- Spring constant (*k*) is bound to the PhS Constant *springcst*.
- Force going through the spring and difference in velocities of the ends (*v*, *f*), are bound to the PhSVariables *forcethru* and *velocitydiff*, respectively.

The PhSVariables and PhS Constants above are defined on the block *Spring*, but not shown in Figure 11. The constraint block defines five constraints representing equations, written using the expression language specified in Clause 8.

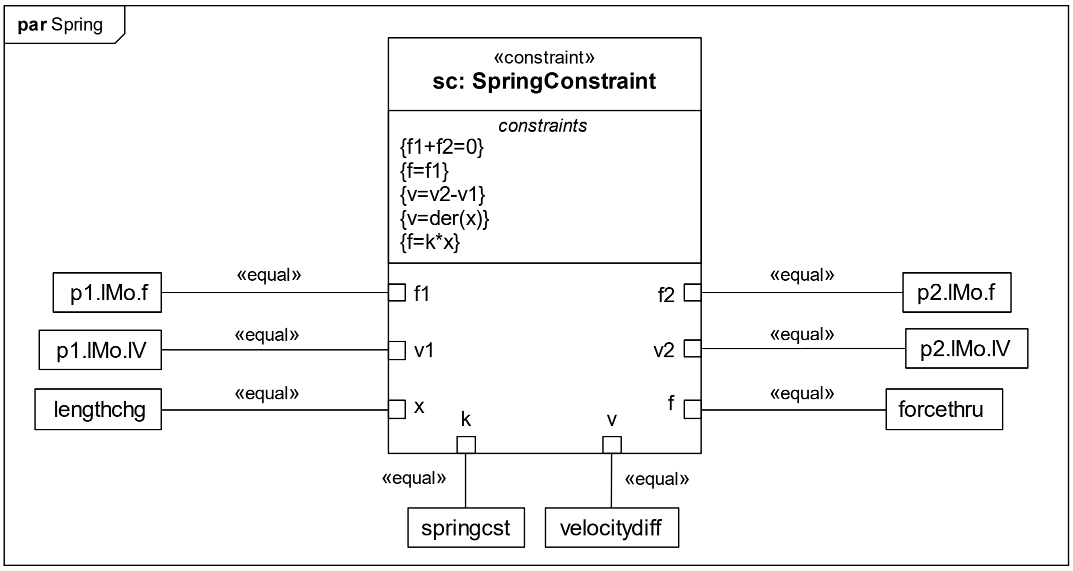


Figure 26: Constraint block for physical interaction in SysML

10.9.8 Modelica modeling, physical interaction

In a SysML block with constraint properties, the constraints correspond to the same equations 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 (and flow properties in SysML property paths leading to PhSVariables on conserved quantity kinds are omitted in Modelica, see Subclause 10.7.8).

The following Modelica code corresponds to Figure 26. It has five equations from the SysML constraint block. SysML parameter names are replaced in the Modelica equations according the bindings in Figure 14: *f1* is replaced by *p1.f*, *v1* is replaced by *p1.lV*, *x* is replaced by *lengthchg*, *k* is replaced by *springcst*, *v* is replaced by *velocitydiff*, *f* is replaced by *forcethru*, *v2* is replaced by *p2.v*, and *f2* is replaced by *p2.f*.

```
model Spring
  Flange p1;
  Flange p2;
  Real lengthchg;
  parameter Real springcst = "10";
  Real velocitydiff
  Real forcethru
equation
  p1.f+p2.f=0
  forcethru=p1.f;
  velocitydiff=p1.lV-p2.lV;
  velocitydiff=der(lengthchg);
  forcediff=springcst*lengthchg;
end Spring;
```

10.9.9 Simulink modeling, physical interaction

Physical interaction is modeled with the Simscape extension to Simulink, see Subclause 10.9.10.

10.9.10 Simscape modeling, physical interaction

For SysML blocks with constraint properties, the constraints correspond to the same equations in Simscape components (assuming the expression language of Clause 8 is used in constraint blocks), with the same substitutions in Simscape as in Modelica (see Subclause 10.9.8), followed by additional substitutions for balancing variables in Simscape domains (see Subclause 10.7.10 about domains). The additional substitutions are defined in Simscape branch statements, each introducing a new variable to substitute in equations (after the initial substitutions above) for each path to a balancing variable on a port.

The following Simscape code corresponds to Figure 26. It has five equations from the SysML constraint block. Note the additional variables defined by branch statements, which replace *p1.f* by *p1f* and *p2.f* by *p2f* in the equations (after the initial substitutions above).

```
component Spring
  variables
    forcethru={0,'N'};
    velocitydiff={0,'m/s'};
    lengthchg={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;
    forcethru=p1f;
    velocitydiff=p1.lV-p2.lV;
    velocitydiff=der(lengthchg);
    forcethru=springcst*lengthchg;
  end
end
```

10.9.11 Summary

SysML	Modelica	Simulink	Simscape
Constraint block, typing constraint properties	N/A	S-function	N/A
Constraint parameter bound to a property path that goes through an in flow property	N/A (SysML constraint parameter substituted in equations)	Input variable	N/A (SysML constraint parameter substituted in equations)
Constraint parameter bound to a property path that goes through an out flow property	N/A (SysML constraint parameter substituted in equations)	Output variable	N/A (SysML constraint parameter substituted in equations)
Constraint parameter bound to continuous PhSVariable	N/A (SysML constraint parameter substituted in equations)	Continuous state variable	N/A (SysML constraint parameter substituted in equations)
Constraint parameter bound to discrete PhSVariable	N/A (SysML constraint parameter substituted in equations)	Discrete state variable	N/A (SysML constraint parameter substituted in equations)
Constraint parameter bound to discrete PhSConstant	N/A (SysML constraint parameter substituted in equations)	Numeric or boolean value (substituted in equations)	N/A (SysML constraint parameter substituted in equations)
Constraint	Equation in the model corresponding to the SysML block 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 corresponding to the SysML block containing the constraint property (with substitution of parameters)

10.10 Default values and initial values

10.10.1 Purpose

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

10.10.2 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, whereas default values are only set when instances are created. Default and initial values can be changed after they are given to the instances.

Figure 27 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*.

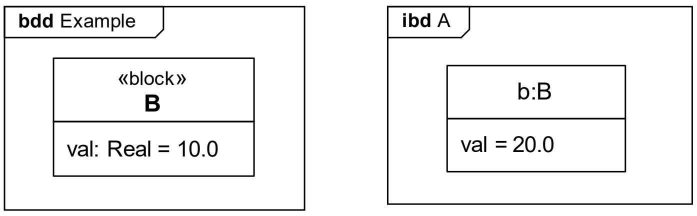


Figure 27: Default values and initial value in SysML

10.10.3 Modelica modeling

SysML default and initial values correspond to start values of Modelica components. Start values are marked as fixed, requiring the values be set at the beginning of the simulation (otherwise, simulators only take the values as suggestions, calculating their own start values to solve the equations).

The following Modelica code corresponds to Figure 15. 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;
```

10.10.4 Simulink modeling

Default values (or overriding initial values) of PhSVariables correspond to initial values of the corresponding S-functions variables (see Subclause 10.9.5), unless they are initial values for properties below the top-level system block or are for properties typed by blocks that have parts, whereupon they have the same correspondence with Simulink as redefined properties (see Subclause 10.5.4 and Subannex A.5.9).

The following Simulink code corresponds to Figure 15, assuming the PhSVariable *var* is bound to a constraint parameter (which corresponds to an S-function variable). 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). The properties *NumD works*, *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
```

10.10.5 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 initial values 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 block definition diagram in Figure 15. It 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 internal block diagram in Figure 15. 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>
```

10.10.6 Summary

SysML	Modelica	Simulink	Simscape
Default value	Start value (fixed)	S-function initial value	Member initial value (high priority)
Initial value	Start value (fixed)	N/A	Member assignment (high priority)

10.11 Data types and units

10.11.1 Purpose

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

10.11.2 SysML modeling

Data types in SysML are called value types. SysML numeric value types can be linked to units, where units are modeled with the SysML Unit block. These units are linked to value types that are generalized by SysML's numeric value types. Units and their symbols are from ISO 80000.

Figure 28 shows how a value type with units is defined in SysML, from the units library in Figure 20, Subclause 11.2.2 It has a value type *Force* that specializes the *Real* value type and has *newton* as unit. The *newton* unit has a symbol *N*.

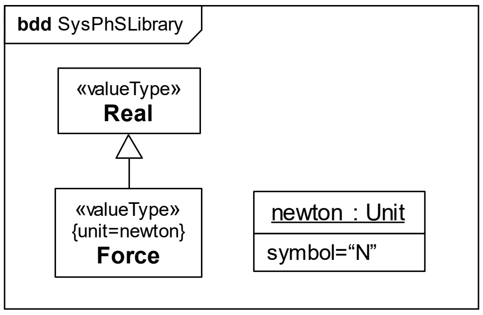


Figure 28: Units in SysML

10.11.3 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 28. It has a type *Force*, which extends *Real*, and the unit symbol *N* assigned to it.

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

10.11.4 Simulink modeling

Simulink inports and outports can have units. Simulink defines some unit symbols, and modelers can add their own. The following table shows correspondences between ISO 80000 and Simulink notation for unit operations when they differ.

Unit operation	ISO 80000	Simulink
Exponentiation	superscript (as in m^3)	caret (as in m^3)
Multiplication	· (as in N·m)	* (as in N*m)

The following table shows correspondences between ISO 80000 and Simulink notation for units when they differ.

ISO 80000	Simulink
Ω	ohm
°	deg
Å	ang
µ	u

The following Simulink code corresponds to Figure 16. It has an inport In1 with unit N, the symbol for Newton.

```
<Block BlockType="Inport" Name="In1" SID="1">
     <P Name="Unit">N</P>
</Block>
```

10.11.5 Simscape modeling

Unit symbols can be associated to variables and parameters in Simscape. Simscape uses the unit symbols defined in Simulink (see Subclause 10.11.4).

The following Simscape code corresponds to Figure 28. It has a variable *force* with an initial value of *0*, with the unit *N*, the symbol for Newton.

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

10.11.6 Summary

SysML	Modelica	Simulink	Simscape
Value type specializing Real, Integer, or Boolean with unit	Equivalent data type with unit symbol	N/A	N/A
Property typed by Real, Integer, Boolean or one of their specializations	Component typed by an equivalent data 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

10.12 State machines

10.12.1 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 10.9.5). 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.

10.12.2 SysML modeling

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

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

Figure 29 shows a block *Computer* with a simple state machine.

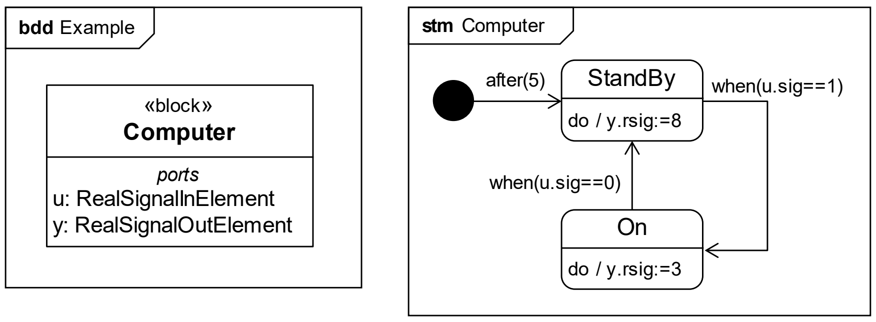


Figure 29: State machine in SysML

RealInSignalElement and *RealOutSignalElement* from the signal flow library (see Subclause 11.2.1), respectively. The state machine has one initial pseudostate, and two states *StandBy* and *On*. The transition from the initial pseudostate to *StandBy* has a relative TimeEvent with an expression indicating that the transition fires 5 seconds after the initial pseudostate is entered. 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.

10.12.3 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.
- Time events correspond to transition wait times.
- 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 29.

```
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 = true, enableTimer = true,
                                       waitTime = 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.

10.12.4 Simulink/StateFlow modeling

Simulink has an extension for state machines called Stateflow, providing some features of SysML state machines (StateFlow does not extend Simscape). 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 StateFlow code correspond to Figure 29.

```
<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 indicates 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.

10.12.5 Summary

SysML	Modelica	Simulink	Stateflow
Block with StateMachine as classifierBehavior	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
Relative TimeEvent	waitTime expression	N/A	after() statement

10.13 Mathematical expressions

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

SysPhS expression	MATLAB equivalent
‘if’ ... ‘then’ ... ‘elseif’ ... ‘then’ ... ‘else’ ... ‘end’ ‘if’	‘if’ ... ... ‘elseif’ ... ‘else’ ...
‘for’ ... ‘in’ ... ‘loop’ ... ‘end’ ‘for’	‘for’ ... ‘=’ ... ... [illegible]
‘=’	‘==’
‘<>’	‘~=’
‘not’	‘~’
‘and’	‘&&’
‘or’	‘||’
‘:=’	‘=’
‘div’	‘idivide’

11. Platform-independent Component Library

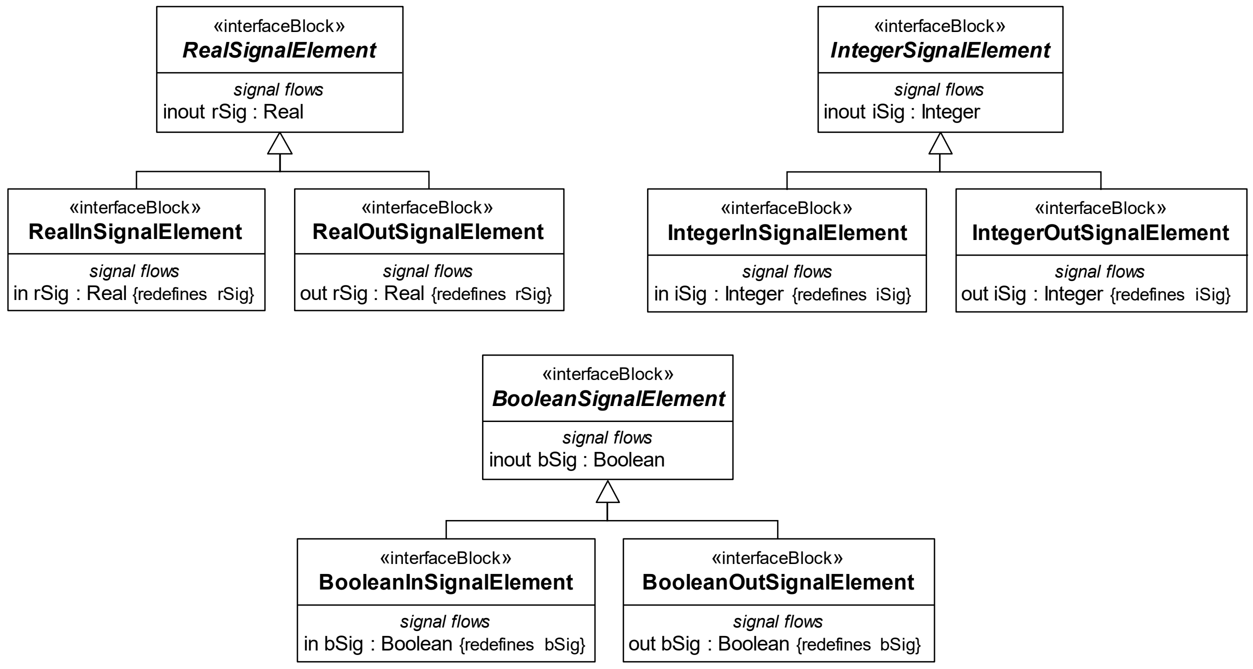
11.1 Introduction

Subclauses 11.2 and 11.3 define a platform-independent library of reusable blocks for component interaction and behavior, respectively. Subclause 0 defines value types with units used in Subclause 11.2.2. Subclause 11.5 defines a simulation platform extension used in Subclause 11.3.

11.2 Component interaction

11.2.1 Signal flow

This subclause defines elements for signal flow. They can be used as (generalizations of) system component blocks or port types. See Subclause 11.3.4 for additional signal flow elements.


«interfaceBlock» *RealSignalElement*
signal flows
inout rSig : Real
«interfaceBlock» RealInSignalElement
signal flows
in rSig : Real {redefines rSig}
«interfaceBlock» RealOutSignalElement
signal flows
out rSig : Real {redefines rSig}
«interfaceBlock» *IntegerSignalElement*
signal flows
inout iSig : Integer
«interfaceBlock» IntegerInSignalElement
signal flows
in iSig : Integer {redefines iSig}
«interfaceBlock» IntegerOutSignalElement
signal flows
out iSig : Integer {redefines iSig}
«interfaceBlock» *BooleanSignalElement*
signal flows
inout bSig : Boolean
«interfaceBlock» BooleanInSignalElement
signal flows
in bSig : Boolean {redefines bSig}
«interfaceBlock» BooleanOutSignalElement
signal flows
out bSig : Boolean {redefines bSig}

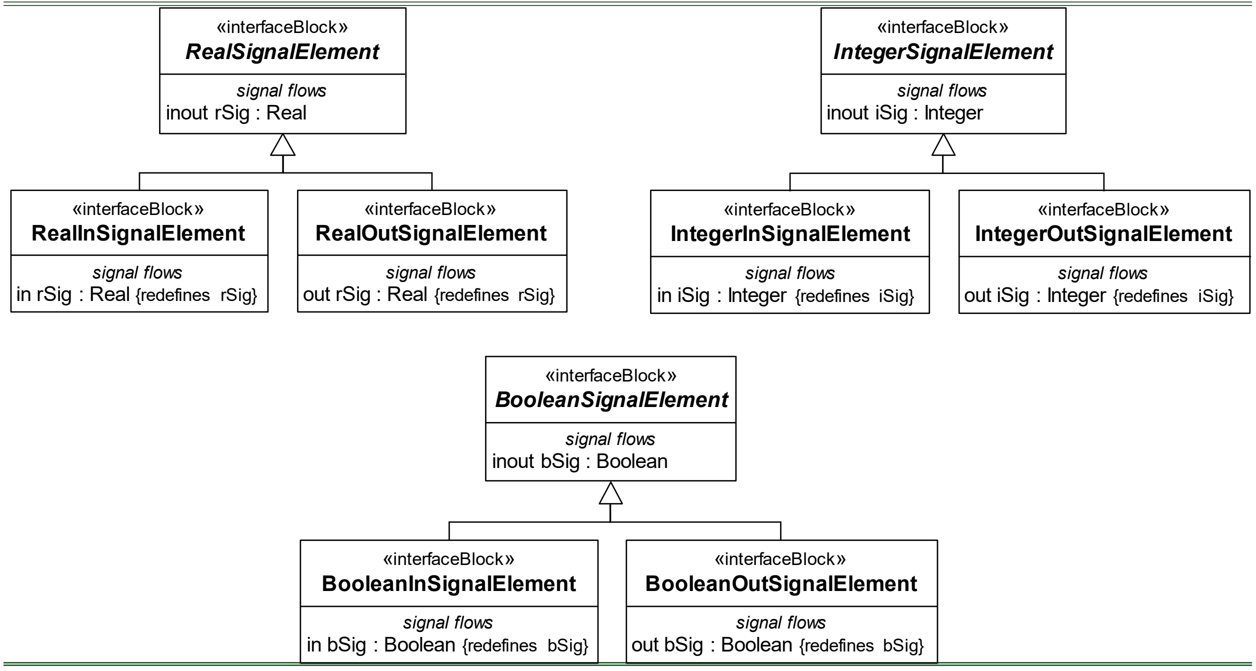

«interfaceBlock» *RealSignalElement*
signal flows
inout rSig : Real
«interfaceBlock» RealInSignalElement
signal flows
in rSig : Real {redefines rSig}
«interfaceBlock» RealOutSignalElement
signal flows
out rSig : Real {redefines rSig}
«interfaceBlock» *IntegerSignalElement*
signal flows
inout iSig : Integer
«interfaceBlock» IntegerInSignalElement
signal flows
in iSig : Integer {redefines iSig}
«interfaceBlock» IntegerOutSignalElement
signal flows
out iSig : Integer {redefines iSig}
«interfaceBlock» *BooleanSignalElement*
signal flows
inout bSig : Boolean
«interfaceBlock» BooleanInSignalElement
signal flows
in bSig : Boolean {redefines bSig}
«interfaceBlock» BooleanOutSignalElement
signal flows
out bSig : Boolean {redefines bSig}

Figure 30: Elements for signal flow

11.2.2 Physical interaction

This subclause defines elements for physical interaction (see Subclause 0 for and associated value types and units). Conserved quantity kinds are characteristics of physical substances that are not created or destroyed when exchanged between components. For example, charge is a characteristic of elementary physical particles that might cross the boundaries of an object. Conserved quantity kinds are modeled as blocks directly specializing the block ConservedQuantityKind, which specializes SysML QuantityKind, as shown in Figure 31. These can be conveyed by item flows and the type of item properties. Specializations of each conserved quantity kind (with names prefixed by "Flowing") are only used to type flow properties. They provide two PhSVariables describing the flows, one conserved (flow rate) and one non-conserved (potential to flow). For example, the flow rate of charge (current) must add to zero (be conserved) between components, while the potential to flow (voltage) must be the same (see Subclause 7.2.2). These variables only apply to conserved quantity kinds as they cross the boundary of components via flow properties, because they are defined with respect to the boundary (rate of crossing it or potential to cross it). The flow properties can be on blocks used as (generalizations of) part or port types, including interface blocks as shown at the bottom of Figure 31.

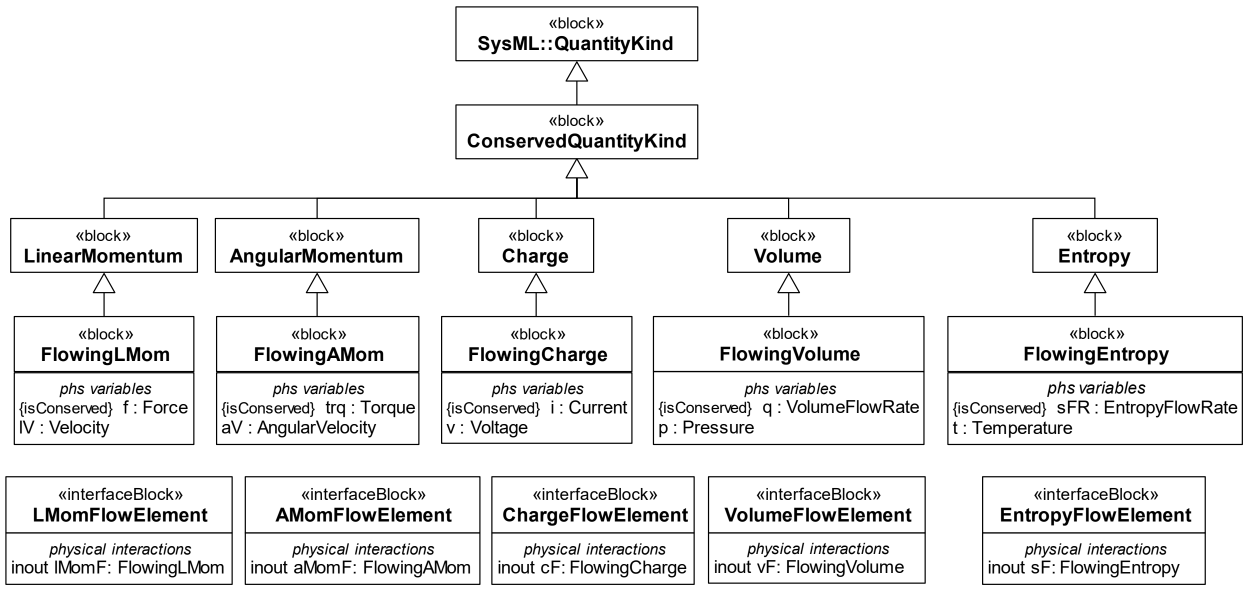


Figure 31: Elements for physical interaction

Constraints

[1] Blocks (indirectly) specializing ConservedQuantityKind that type flow properties must have one conserved and one non-conserved PhSVariable.
[2] Flow properties typed by blocks (indirectly) specializing ConservedQuantityKind must have direction inout and multiplicity 1.
[3] Flow properties typed by blocks (indirectly) specializing ConservedQuantityKind that are connected and matching must have the same type and multiplicity.

11.3 Component behavior

11.3.1 Introduction

This subclause defines SysML blocks corresponding to reusable components in the libraries of both Modelica and Simulink or its extensions. The semantics of these blocks are given by the corresponding elements in the Modelica libraries (which is the same semantics as in the libraries of Simulink or its extensions). The base classes and properties (including ports) of component blocks in this subclause have stereotypes from the simulation platform profile applied (see Subclause 11.5) to specify which simulation library elements correspond to them. For brevity, component blocks are described in tables, with each row defining one block.

The blocks in Subclauses 11.3.2 and 11.3.3 are for signal flow modeling. The columns of the tables are:

- *Component Block*: Name of the component block defined by the row.

  - *Simulink Block*: Value of the name property of the SimulinkBlock stereotype applied to the base class of the block defined by the row.
  - *Modelica Block*: Value of the name property of the ModelicaBlock stereotype applied to the base classof the block defined by the row is produced from this column by prepending "Modelica.Blocks."

- *Component Ports (Inputs* and *Outputs)*: Each line in each row of these columns gives the name of a~~a~~ component block port (these correspond to Simulink and Modelica ports and components, see Subclauses 10.7.5 and 10.7.4).

- *PhSConstants*: Each line in each row of this column gives the name of a property of the block defined by the row, corresponding to the same line in the two columns below.
  - *Simulink and Modelica Parameters*: Value of the name properties of SimulinkParameter and ModelicaParameter stereotypes, respectively, applied to the corresponding property on the same line in the PhSConstants column (the parameter stereotypes are specialized PhSConstants, see Subclause 11.5). Lines that have no corresponding property on the same line in the PhSConstants column, if any, give other parameters needed to obtain the same behavior in Simulink and Modelica, with the value of the parameter preceded by an equals sign.

- *Platform Behavior*: Tells whether the behaviors of the Simulink and Modelica library elements are supposed to yield the same value or not, when this can be determined from the platform library specifications. Values are considered the same when they are equal or the numerical difference is small.

Simulation platform data specified in the Component Ports (Input and Output), PhSConstants, and platform Parameters columns are scalar, unless marked with a V (vector) or an M (matrix). Component input ports for scalars are typed by RealSignalInElement, IntegerSignalInElement, or BooleanSignalInElement, while component output ports for scalars are typed by RealSignalOutElement, IntegerSignalOutElement, or BooleanSignalOutElement (see Subclause 11.2.1). Component input ports for vectors are typed by specializations of RealVectorSignalInElement, while component output ports for vectors are typed by specializations of RealVectorSignalOutElement (see Subclause 11.5.3). Component PhSConstants (SimulinkParameters and ModelicaParameters) for vectors and matrices have MultidimensionalElement applied, with dimension * and *,*, respectively (see Subclause 11.5.2.4). Models using component library blocks that have vector and matrix properties should specify initial values using instance specifications, with slots satisfying the constraints specified in Subclause 11.5.2.4.

The blocks in Subclause 11.3.4 are for electrical components. The columns of the table are explained in that~~t~~ subclause.

11.3.2 Real-valued components

11.3.2.1 Introduction

Simulation platform data specified in the Component ports (Inputs and Output), PhSConstants, and platform Parameters columns in this subclause are Real, unless otherwise indicated.

11.3.2.2 Continuous components

Component Block	Simulink Block	Modelica Block	Component Port (Inputs)	Component Port (Output)	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
Integrator	Integrator	Continuous.Integrator	u	y	init	InitialCondition	y_start	Same
Derivative	Derivative	Continuous.Derivative	u	y				Different
StateSpace	StateSpace	Continuous.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	TransferFcn	Continuous.TransferFunction	u	y	num (V) denom (V)	Numerator (V) Denominator (V)	b (V) a (V)	
FixedDelay	Transport Delay	Nonlinear.FixedDelay	u	y	delay	DelayTime InitialOutput=0	delayTime	Different
VariableDelay	Variable Transport Delay	Nonlinear.VariableDelay	u delayTime	y	delayMax	MaximumDelay InitialOutput=0 VariableDelayType =Variable time delay ZeroDelay=on	delayMax	Different

11.3.2.3 Discrete components

Component Block	Simulink Block	Modelica Block	Component Port (Inputs)	Component Port (Outputs)	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
StateSpace	DiscreteState Space	Discrete.StateSpace	u (V)	y (V)	A (M) B (M) C (M)	A (M) B (M) C (M)	A (M) B (M) C (M)	Same
TransferFunction	Discrete TransferFcn	Discrete.TransferFunction	u	y	numerator (V) denominator (V)	Numerator (V) Denominator (V)	b (V) a (V)	Same
UnitDelay	UnitDelay	Discrete.UnitDelay	u	y	initialCondition	InitialCondition	y_start	Same

11.3.2.4 Non-linear components

Component Block	Simulink Block	Modelica Block	Component Port (Inputs)	Component Port (Outputs)	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
Saturation	Saturate	Nonlinear.Limiter	u	y	upper lower	UpperLimit LowerLimit	uMax uMin	Same (min AND max mandatory)
Dynamic Saturation	Reference	Nonlinear.VariableLimiter	limit1 u limit2	y		SourceBlock= simulink/Discontinuities /Saturation Dynamic SourceType=Saturation Dynamic		Same
DeadZone	DeadZone	Nonlinear.DeadZone	u	y	lower upper	LowerValue UpperValue	uMin uMax	Same
RateLimiter	RateLimiter	Nonlinear.SlewRateLimiter	u	y	rising falling	RisingSlewLimit FallingSlewLimit	Rising Falling	Different

11.3.2.5 Mathematical components

Component Block	Simulink Block	Modelica Block	Component Port (Inputs)	Component Port (Outputs)	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
Gain	Gain	Math.Gain	u	y	gain	Gain	k	Same
Product	Product	Math.Product	u1	y		Inputs=**		Same
Division	Product	Math.Division	u1	y		Inputs=*/		Same
Addition	Sum	Math.Add	u1	y		Inputs=++		Same
Subtraction	Sum	Math.Add	u1	y		Inputs=+-		Same
Abs	Abs	Math.Abs	u	y				Same
Exp	Math	Math.Exp	u	y		Operator=exp		Same
Log	Math	Math.Log	u	y		Operator=log		Same
Log10	Math	Math.Log10	u	y		Operator=log10		Same
Sign	Signum	Math.Sign	u	y				Same
Sqrt	Sqrt	Math.Sqrt	u	y				Same
Sin	Trigonometry	Math.Sin	u	y		Operator=sin		Same
Cos	Trigonometry	Math.Cos	u	y		Operator=cos		Same
Tan	Trigonometry	Math.Tan	u	y		Operator=tan		Same
Asin	Trigonometry	Math.Asin	u	y		Operator=asin		Same

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

11.3.2.6 Sources and sinks

Component Block	Simulink Block	Modelica Block	Component Port (Inputs)	Component Port (Output)	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
Constant	Constant	Sources.Constant		y	k	Value	k	Same
SineWave	Sin	Sources.Sine		y	amplitude offset frequency phase	Amplitude Bias Frequency Phase	amplitude offset freqHz phase	Same
Clock	Clock	Sources.Clock		y				Same
Pulse	DiscretePulse Generator	Sources.Pulse		y	amplitude period width	Amplitude Period PulseWidth PhaseDelay	amplitude period width	Same
Step	Step	Sources.Step		y	startTime after	Time After Before=0	startTime height	Same
RealScope	Scope	Interaction.Show.RealValue	numberPort					
BooleanScope	Scope	Interaction.Show.BooleanValue	activePort					

11.3.2.7 Routing components

Multiplicities not equal to 1 for flow properties stereotyped by PhSVariable (signal flows) on Component Ports (Inputs and Outputs) are shown between square brackets. These flow properties have MultidimensionalElement applied, with dimension equal to the multiplicity of the flow property (see Subclause 11.5.2.4). Inputs with multiplicities of 2, 3, 4, 5, 6 are typed by RealVectorSignal2InElement, RealVectorSignal3InElement, RealVectorSignal4InElement, RealVectorSignal5InElement, RealVectorSignal6InElement, respectively. Outputs with multiplicities of 2, 3, 4, 5, 6 are typed by RealVectorSignal2OutElement, RealVectorSignal3OutElement, RealVectorSignal4OutElement, RealVectorSignal5OutElement, RealVectorSignal6OutElement, respectively.

Component Block	Simulink Block	Modelica Block	Component Port (Inputs)	Component Port (Output)	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
Mux2	Mux	Routing.Multiplex2	u1 u2	y [2]		Inputs=2		Same

Mux3	Mux	Routing.Multiplex3	u1 u2	y [3]		Inputs=3		Same
Mux4	Mux	Routing.Multiplex4	u1 u2 u3	y [4]		Inputs=4		Same
Mux5	Mux	Routing.Multiplex5	u1 u2 u3	y [5]		Inputs=5		Same
Mux6	Mux	Routing.Multiplex6	u1 u2 u3 u4	y [6]		Inputs=6		Same
Demux2	Demux	Routing.DeMultiplex2	u [2]	y1 y2		Outputs=2		Same
Demux3	Demux	Routing.DeMultiplex3	u [3]	y1 y2		Outputs=3		Same
Demux4	Demux	Routing.DeMultiplex4	u [4]	y1 y2 y3		Outputs=4		Same
Demux5	Demux	Routing.DeMultiplex5	u [5]	y1 y2 y3		Outputs=5		Same
Demux6	Demux	Routing.DeMultiplex6	u [6]	y1 y2 y3		Outputs=6		Same

Switch			u1 u2 u3	y		Criteria = u2~=0 Threshold=0		Same

11.3.3 Logical components

Simulation platform data specified in the Component ports (Inputs and Output) and platform Parameters columns in this subclause are Boolean, unless marked with an R (real).

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

11.3.4 Electrical components

Blocks in this subclause are for physical interaction between electrical components, some including signal flow of electrical quantities. The columns are the same as in Subclausess 11.3.2 and 11.3.3, except:

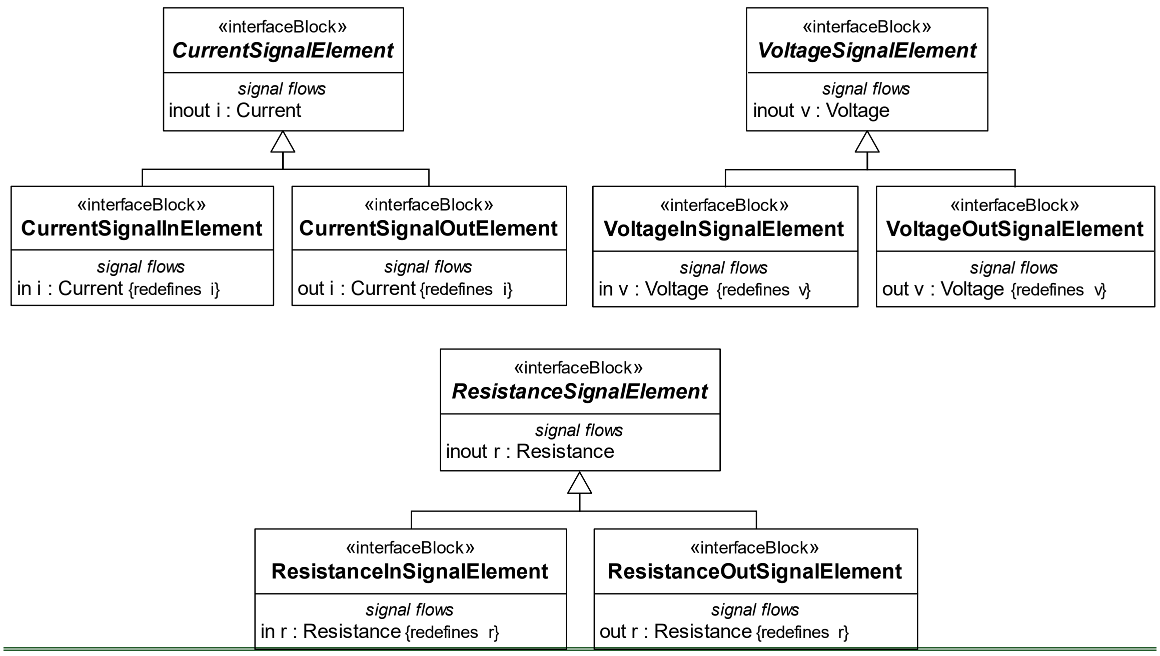
- Values of the name property of the SimulinkBlock and ModelicaBlock stereotypes applied to the base class of the block defined by each row are produced from these columns by prepending "foundation.electrical." for SimulinkBlocks and "Modelica.Electrical.Analog." for ModelicaBlocks. SimulinkBlocks in this table are Simscape library elements (an extension of Simulink, see Clause 10.1)
- ~~There is only one column for component ports, because they are mostly bidirectional, typed by FlowingChargeElement (see Subclause 11.2.2). Some components have additional unidirectional ports, typed by signal elements defined in Figure 32. Each line in the Component Ports column gives the name of a port. The corresponding lines in the Simulink Ports and Modelica Ports columns give the port names on the respective platforms. The component port is stereotyped by SimulinkPort and/or ModelicaPort when the name is different on that platform (SimulinkPort is used for Simscape ports in this table). In this case, the platform name is given as the value of the name property of the respective SimulinkPort and/or ModelicaPort stereotype.~~ There is only one column for component ports, because they are mostly bidirectional, typed by FlowingChargeElement (see Subclause 11.2.2). Some components have additional unidirectional ports, typed by signal elements defined in Figure 32. Each line in the Component Ports column gives the name of a port. The corresponding lines in the Simulink Ports and Modelica Ports columns give the port names on the respective platforms. The component port is stereotyped by SimulinkPort and/or ModelicaPort when the name is different on that platform (SimulinkPort is used for Simscape ports in this table). In this case, the platform name is given as the value of the name property of the respective SimulinkPort and/or ModelicaPort stereotype.

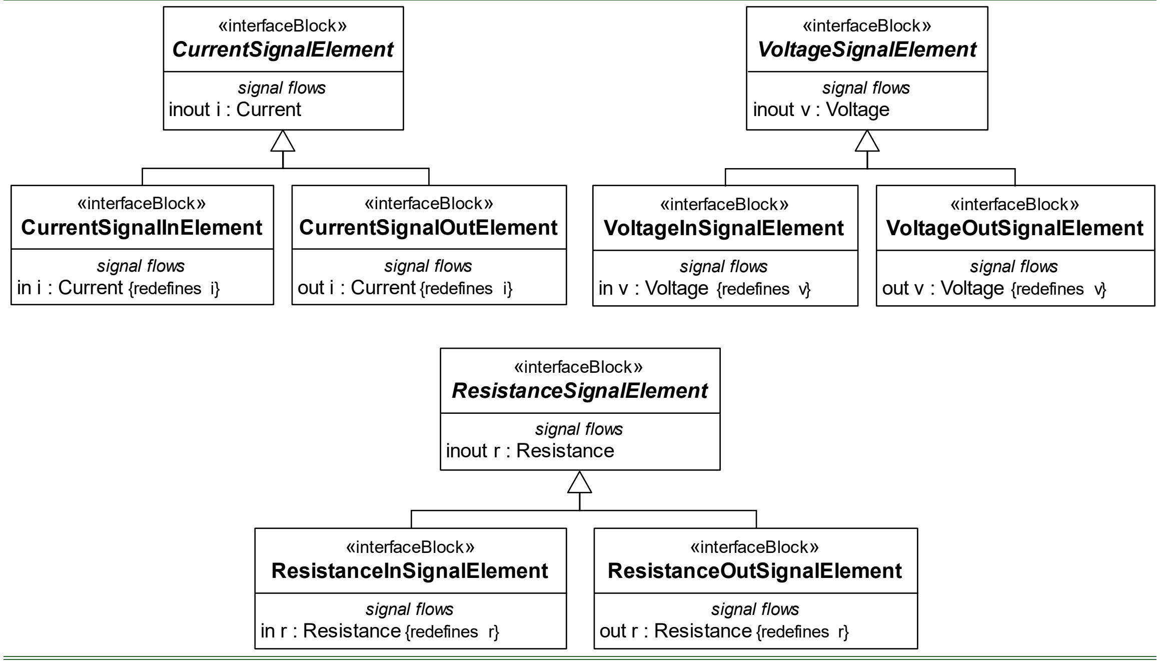
Component Block	Simulink Block	Modelica Block	Component Ports	Simulink Ports	Modelica Ports	PhSConstants	Simulink Parameters	Modelica Parameters	Platform Behavior
Ground	elements.reference	Basic.Ground	p	V	p				
Capacitor	elements.capacitor	Basic.Capacitor	p n	p n	p n	c : ~~Capacitance~~ Capacitance	c r=0 g=0	C	Same
Diode	elements.pwl_diode	Ideal.IdealDiode	p n	p n	p n	ron : ~~Resistance~~ ~~goff:Conductance~~ ~~vforward:Voltage~~ Resistance goff:Conductance vforward:Voltage	Ron Goff Vf	Ron Goff Vknee	
Ideal Transformer	elements.ideal _transformer	Ideal.IdealTransformer	p1 n1 p2 n2	p1 n1 p2 n2	p1 n1 p2 n2	n : ~~Real~~ Real	n	n	Same
Inductor	elements.inductor	Basic.Inductor	p n	p n	p n	l : ~~Inductance~~ Inductance	l r=0 g=0	L	Same
Infinite Resistance	elements.infinite _resistance	Ideal.Idle	p n	p n	p n				Same

OpAmp	elements.op_amp	Ideal.IdealOpAmp3Pin	p n out	p n out	in_p in_n out				Same
Resistor	elements.resistor	Basic.Resistor	p n	p n	p n	r : ~~Resistance~~ Resistance	R	R	Same
Variable Resistor	elements.variable _resistor	Basic.~~VariableResistor~~ VariableResistor	p n r : Resistance S~~ignalInElement~~ignalInElement	p n R	p n R				Same
CurrentSensor	sensors.current	Sensors.CurrentSensor	p n i : ~~Current~~ ~~SignalOutElement~~ Current SignalOutElement	p n I	p n i				Same
VoltageSensor	sensors.voltage	Sensors.VoltageSensor	p n v : Voltage SignalOutElement	p n V	p n v				Same
SignalCurrent	sources.controlled_ current	Sources.SignalCurrent	p n i : ~~Current~~ ~~SignalInElement~~ Current SignalInElement	p n iT	p n i				Same
SignalVoltage	sources.controlled_ voltage	Sources.SignalVoltage	p n v : ~~Voltage~~ ~~SignalInElemen~~Voltage SignalInElement	p n vT	p n v				Same
DCCurrent	sources.dc_current	Sources.ConstantCurrent	p n	p n	p n	i : ~~Current~~ Current	i0	I	Same
DCVoltage	sources.dc_voltage	Sources.ConstantVoltage	p n	p n	p n	v : ~~Voltage~~ Voltage	v0	V	Same

ACCurrent	sources.ac_current	Sources.SineCurrent	p n	p n	p n	amp :~~ Current~~ ~~phase : Real~~ ~~freq : Frequency~~ Current phase : Real freq : Frequency	amp shift frequency	I phase freqHz	Same
ACVoltage	sources.ac_voltage	Sources.SineVoltage	p n	p n	p n	amp : ~~Voltage~~ ~~phase : Real~~ ~~freq : Frequency~~ Voltage phase : Real freq : Frequency	amp shift frequency	V phase freqHz	Same

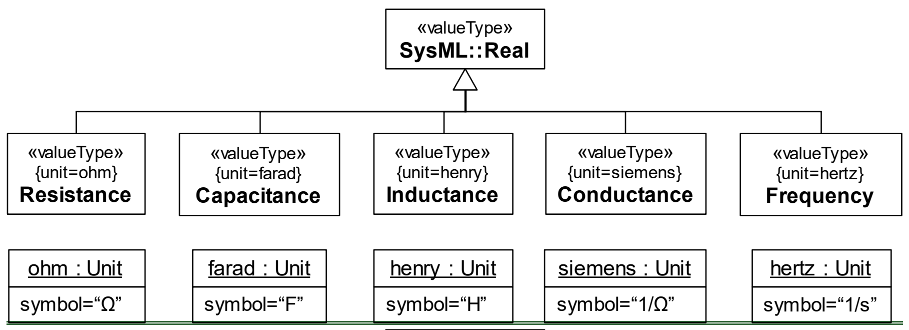
Figure 32 and Figure 33 give additional signal elements and value types with units for electrical modeling (see~~e~~ Figure 30 and Figure 34).


«interfaceBlock» CurrentSignalElement
signal flows
inout i : Current
«interfaceBlock» VoltageSignalElement
signal flows
inout v : Voltage
«interfaceBlock» CurrentSignalInElement
signal flows
in i : Current {redefines i}
«interfaceBlock» CurrentSignalOutElement
signal flows
out i : Current {redefines i}
«interfaceBlock» VoltageInSignalElement
signal flows
in v : Voltage {redefines v}
«interfaceBlock» VoltageOutSignalElement
signal flows
out v : Voltage {redefines v}
«interfaceBlock» ResistanceSignalElement
signal flows
inout r : Resistance
«interfaceBlock» ResistanceInSignalElement
signal flows
in r : Resistance {redefines r}
«interfaceBlock» ResistanceOutSignalElement
signal flows
out r : Resistance {redefines r}



«interfaceBlock» CurrentSignalElement
signal flows
inout i : Current
«interfaceBlock» VoltageSignalElement
signal flows
inout v : Voltage
«interfaceBlock» CurrentSignalInElement
signal flows
in i : Current {redefines i}
«interfaceBlock» CurrentSignalOutElement
signal flows
out i : Current {redefines i}
«interfaceBlock» VoltageInSignalElement
signal flows
in v : Voltage {redefines v}
«interfaceBlock» VoltageOutSignalElement
signal flows
out v : Voltage {redefines v}
«interfaceBlock» ResistanceSignalElement
signal flows
inout r : Resistance
«interfaceBlock» ResistanceInSignalElement
signal flows
in r : Resistance {redefines r}
«interfaceBlock» ResistanceOutSignalElement
signal flows
out r : Resistance {redefines r}

Figure 32: Elements for signal flows of electrical quantities



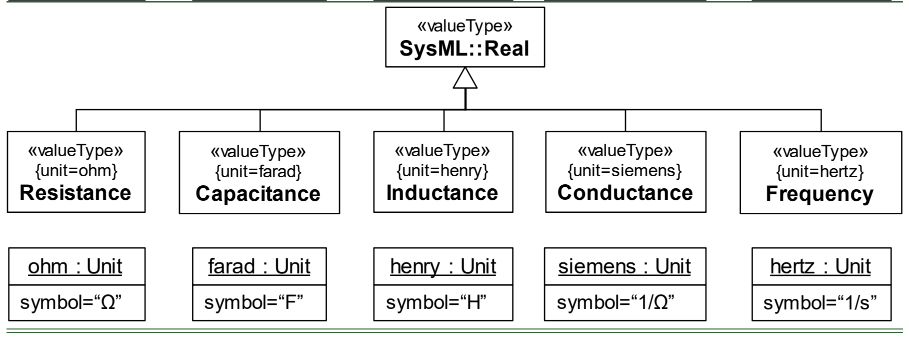


Figure 33: Value types and units for electrical quantities

11.4 Value types with units

This subclause defines value types with units for physical quantities. See Subclause 11.3.4 for additional value types.~~.~~

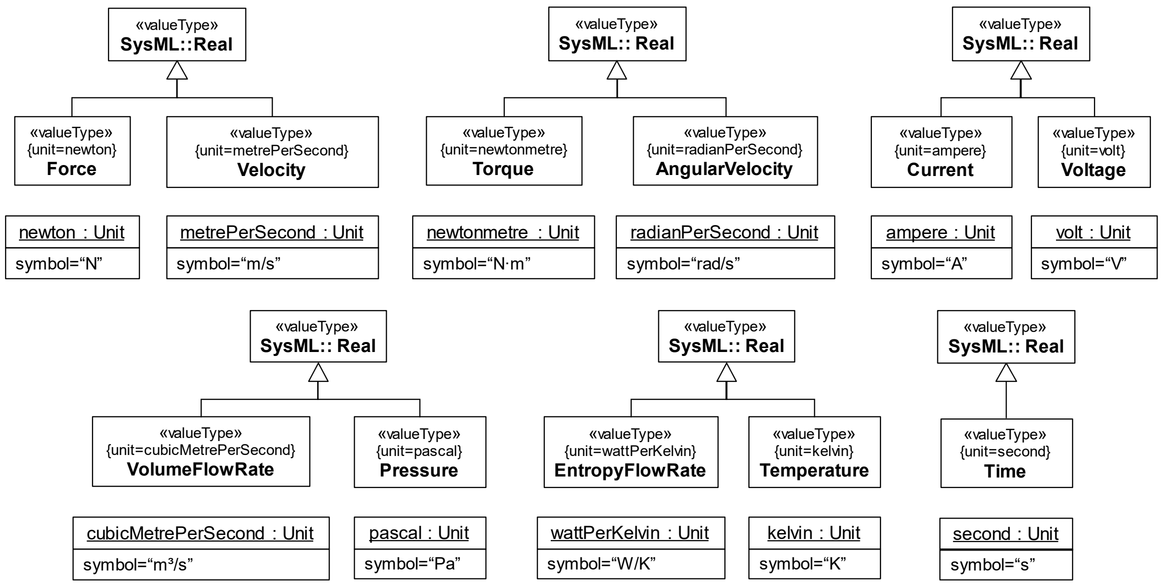


Figure 34: Value types and units for physical quantities~~s~~

11.5 Platform-dependent extension

11.5.1 Introduction

This subclause defines an extension of SysML used by that the platform-independent component library in Subclause 11.3. In this subclause, the Simulink library is taken as including the libraries of its extensions, for brevity.

11.5.2 Platform profile

This subclause defines stereotypes that Subclause 11.3 applies to the base classes and properties (including ports) of its blocks, to specify which library elements of Modelica and Simulink correspond to them.

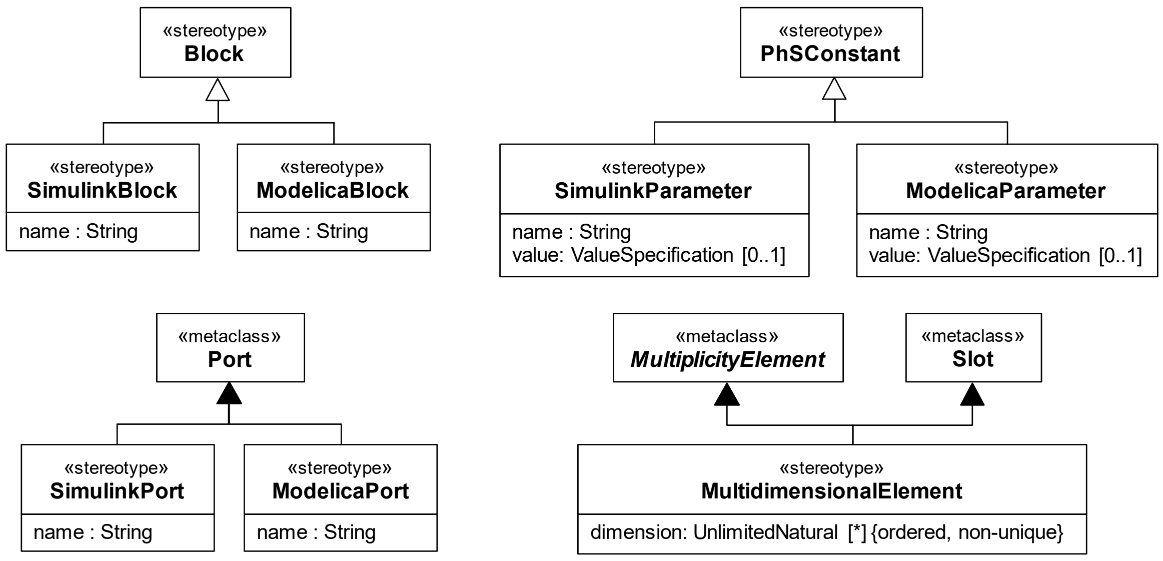


Figure 35: Simulation platform stereotypes

11.5.2.1 ModelicaBlock

Package: SysPhSLibrary
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.

11.5.2.2 ModelicaParameter

Package: SysPhSLibrary
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 port 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] The stereotyped port must be owned by a class stereotyped by ModelicaBlock.

11.5.2.3 ModelicaPort

Package: SysPhSLibrary
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 port 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] The stereotyped port must be owned by a class stereotyped by ModelicaBlock.

11.5.2.4 MultidimensionalElement

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

Attributes

- dimension: 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.

11.5.2.5 SimulinkBlock

Package: SysPhSLibrary
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.

11.5.2.6 SimulinkParameter

Package: SysPhSLibrary
isAbstract: No
Generalization: PhSConstant

Attributes

• name: String	Name of the parameter in the Simulink library corresponding to a parameter of a platform-independent component block
• value: ValueSpecification [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] The stereotyped property must be owned by a class stereotyped by SimulinkBlock.

11.5.2.7 SimulinkPort

Package: SysPhSLibrary
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

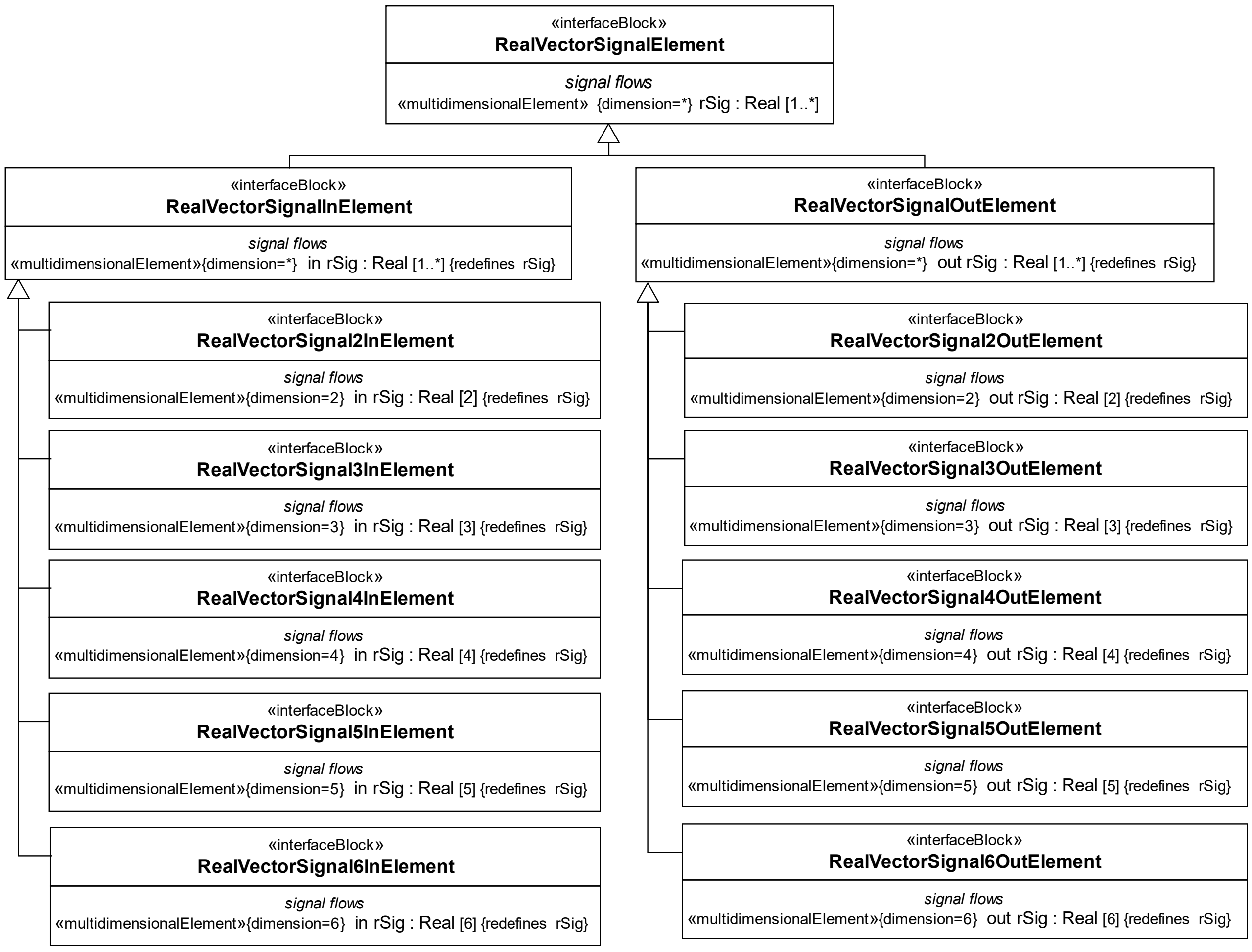
A port stereotyped by SimulinkPort has an equivalent in the Simulink library. The value of the name attribute gives the name of the corresponding port in the Simulink library.

Constraints

[1] The stereotyped port must be owned by a class stereotyped by SimulinkBlock.

11.5.3 Platform library

This subclause defines interface blocks used in Subclause 11.3.2 to specify vector signal flows (see Subclause 11.3.1).



«interfaceBlock»
RealVectorSignalElement
signal flows
«multidimensionalElement» {dimension=*} rSig : Real [1..*]
«interfaceBlock»
RealVectorSignalInElement
signal flows
«multidimensionalElement»{dimension=*} in rSig : Real [1..*] {redefines rSig}
«interfaceBlock»
RealVectorSignalOutElement
signal flows
«multidimensionalElement»{dimension=*} out rSig : Real [1..*] {redefines rSig}
«interfaceBlock»
RealVectorSignal2InElement
signal flows
«multidimensionalElement»{dimension=2} in rSig : Real [2] {redefines rSig}
«interfaceBlock»
RealVectorSignal3InElement
signal flows
«multidimensionalElement»{dimension=3} in rSig : Real [3] {redefines rSig}
«interfaceBlock»
RealVectorSignal4InElement
signal flows
«multidimensionalElement»{dimension=4} in rSig : Real [4] {redefines rSig}
«interfaceBlock»
RealVectorSignal5InElement
signal flows
«multidimensionalElement»{dimension=5} in rSig : Real [5] {redefines rSig}
«interfaceBlock»
RealVectorSignal6InElement
signal flows
«multidimensionalElement»{dimension=6} in rSig : Real [6] {redefines rSig}
«interfaceBlock»
RealVectorSignal2OutElement
signal flows
«multidimensionalElement»{dimension=2} out rSig : Real [2] {redefines rSig}
«interfaceBlock»
RealVectorSignal3OutElement
signal flows
«multidimensionalElement»{dimension=3} out rSig : Real [3] {redefines rSig}
«interfaceBlock»
RealVectorSignal4OutElement
signal flows
«multidimensionalElement»{dimension=4} out rSig : Real [4] {redefines rSig}
«interfaceBlock»
RealVectorSignal5OutElement
signal flows
«multidimensionalElement»{dimension=5} out rSig : Real [5] {redefines rSig}
«interfaceBlock»
RealVectorSignal6OutElement
signal flows
«multidimensionalElement»{dimension=6} out rSig : Real [6] {redefines rSig}

Figure 36: Elements for vector signal flow

Annex A - Examples (non-normative)

A1.1 Introduction

The following subannexes give example models for systems in various domains, using the simulation profile in Clause 7, the expression language in Clause 8, and libraries in Clause 11:

- Subannex A.2: Electric circuits (analog electrical interactions).
- Subannex A.3: Signal processing (manipulation of continuously varying numeric signals).
- Subannex A.4: Hydraulics (fluid interactions).
- Subannex A.5: Humidification (physical control example modeled with signal flows and state machines).
- Subannex A.6: Cruise Control System (control example modeled with physical interactions and signal flows).

Each section describes the system being modeled, then diagrams for internal structure, component types, properties, and constraints.

A1.2 Electric Circuits

A.2.1 Introduction

This subannex gives a model of an electric circuit as an example of physical interaction (flow of electric charge). It does not include any signal flows.

A.2.2 System Being Modeled

The electrical circuit has six components: ground, electrical source, inductor, capacitor, and two resistors, see Figure 37.

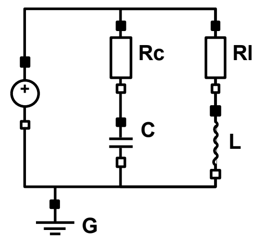


Figure 37: Electric circuit example

A.2.3 Internal Structure

Figure 38 shows the internal structure of a *Circuit* block. Part properties, typed by blocks defined in Subannex A.2.4, represent components of the system. They are connected through ports, which represent electrical pins, also defined in Subannex A.2.4. Item flows on connectors indicate that electricity (electric charge) passes through the ports and flows and between the parts. The diagram connects a voltage source in parallel with a resistor and capacitor in series, as well as a resistor and inductor in series.

SysML initial values specify property values for components used in internal block diagrams. Figure 38 shows initial values for resistance, capacitance, inductance, and source amplitude (properties defined in Subannex A.2.4). An alternative for specifying initial values of part properties in the *Circuit* block is to specialize it and redefine the part properties with default values for various configurations (see Subannex A.5.9).

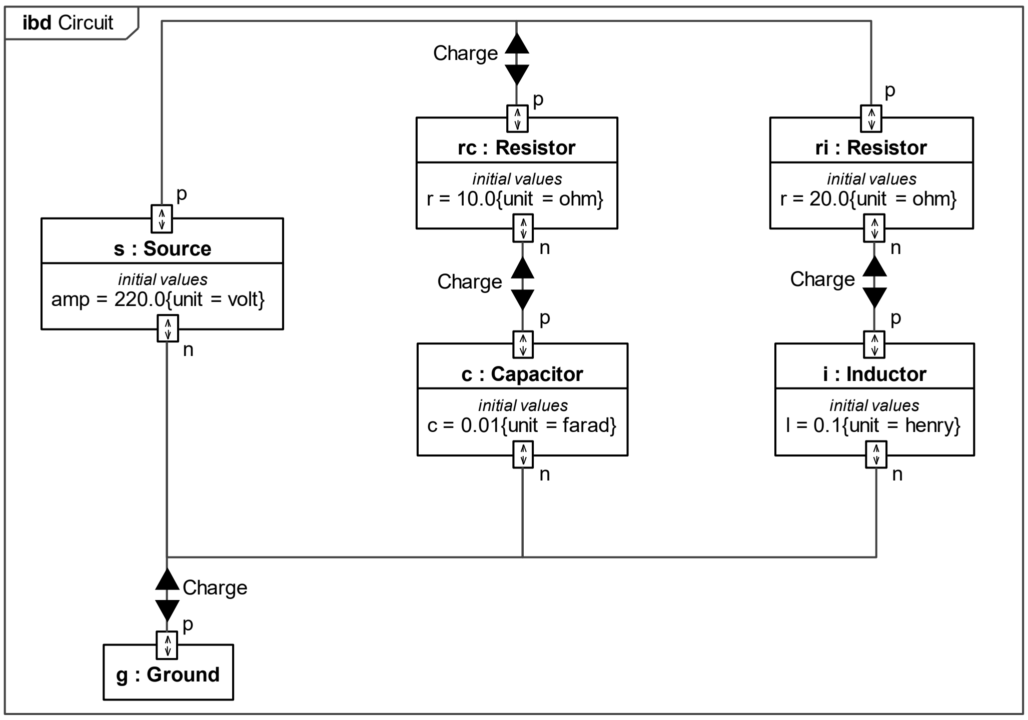


Figure 38: Internal structure of the circuit example

A.2.4 Blocks and Ports

Figure 39 shows block definitions for components of *Circuit* in Figure 39. Sources, inductors, conductors, and resistors each have one positive and one negative pin for electric charge to pass through. Since they are similar in this sense, a generalized *TwoPinElectricalComponent* component is defined with positive and negative pins, *p* and *n*, as ports. The ground has only one pin, which is positive. All ports are of type *ChargeFlowElement*, from the physical interaction library (see Subclause 11.2.2). Each component has its own behaviors, defined as constraints in A.2.6. Some electrical~~l~~ value types in Figure 39 are from the electrical components library (see Subclause 11.3.4). Alternatively, some~~e~~ components could be reused from that library also.

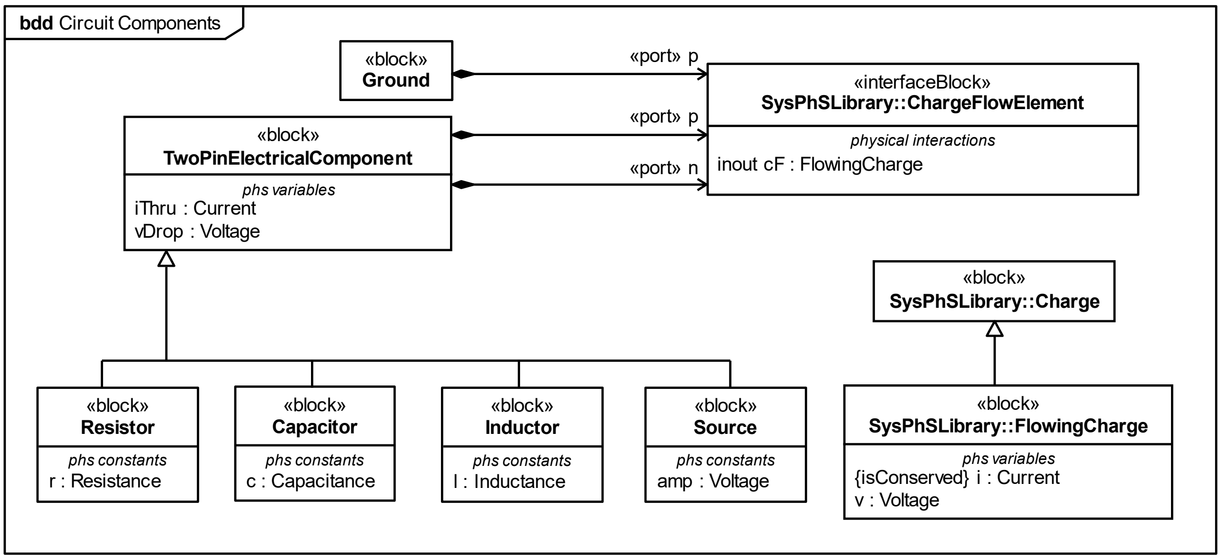


Figure 39: Electrical blocks, ports & component properties

A.2.5 Properties (variables)

Physical interaction is the movement of physical substances between system components, modeled in terms of conserved characteristics of the substances. In this example, electric charge is the conserved characteristic of electrons moving through the circuit. Movement of substances is described by numeric variables for flow rate and potential to flow of their conserved characteristics. In this example, movement of charge is described by a current variable for flow rate and a voltage variable for potential to flow. The flow rate variable is conserved (values on ends of the interaction sum to zero) and the potential variable is not (values on ends of the interaction are the same). This is modeled in three parts:

- Conserved physical characteristics are modeled as blocks directly specialized from *ConservedQuantityKind* in the physical interaction library (see Subclause 11.2.2), *Charge* in this example.
- Flow variables are modeled as properties with PhSVariable applied on specializations of conserved quantity kind blocks. In this example, the flow rate and potential PhSVariables are *i* and *v* on *FlowingCharge* (*i* marked as *isConserved*), respectively, typed by *Current* and *Voltage*, respectively, all from the physical interaction library.
- Flow in and out of components is modeled by ports typed by interface blocks that have flow properties typed by flowing conserved quantity kinds. In this example, ports are typed by *ChargeFlowElement* from the physical interaction library, which has a flow property *cF* typed by *FlowingCharge*, as shown in Figure 39.

Behavior of electrical components in this example is described by the amount of charge going in one pin and out the other (through the component) per unit time, and the difference in potentials between their positive and negative pins (across the component), given by the two properties *iThru* and *vDrop* on *TwoPinElectricalComponent*, respectively, shown in Figure 39. These two properties are typed by *Current* and *Voltage*, respectively, from the physical interaction library (see Subclause 11.2.2), and have the PhSVariable stereotype applied, specifying that their values might change during simulation.

The resistor, capacitor, inductor, and source have properties *r*, *c*, *l*, and *amp*, respectively, typed by *Resistance*, *Capacitance*, *Inductance*, and *Voltage*, respectively, and all with the PhSConstant stereotype applied, specifying that their values do not change during each simulation run.

A.2.6 Constraints (equations)

Equations define mathematical relationships between the values of numeric variables. Equations in SysML, are constraints in constraint blocks that use properties of the blocks (parameters) as variables. In this example, a constraint block *BinaryElectricalComponentConstraint* defines parameters and constraints common to resistors, inductors, capacitors, and sources, as shown in Figure 40. These specify that the voltage *v* across the component is equal to the difference between the voltage at the positive and negative pins. The current *i* through the component is equal to the current going through the positive pin. The sum of the current going through the two pins adds up to zero (one is the negative of the other), because the components do not create, destroy, or store charge. The constraints for the resistor, capacitor, and inductor specify the voltage/current relationship with resistance, capacitance, and inductance, respectively. The source constraint defines the circuit's electrical source. The ground constraint specifies that the voltage at the ground pin is zero. The source constraint defines the voltage across it as a sine wave with the parameter *amp* as its amplitude.

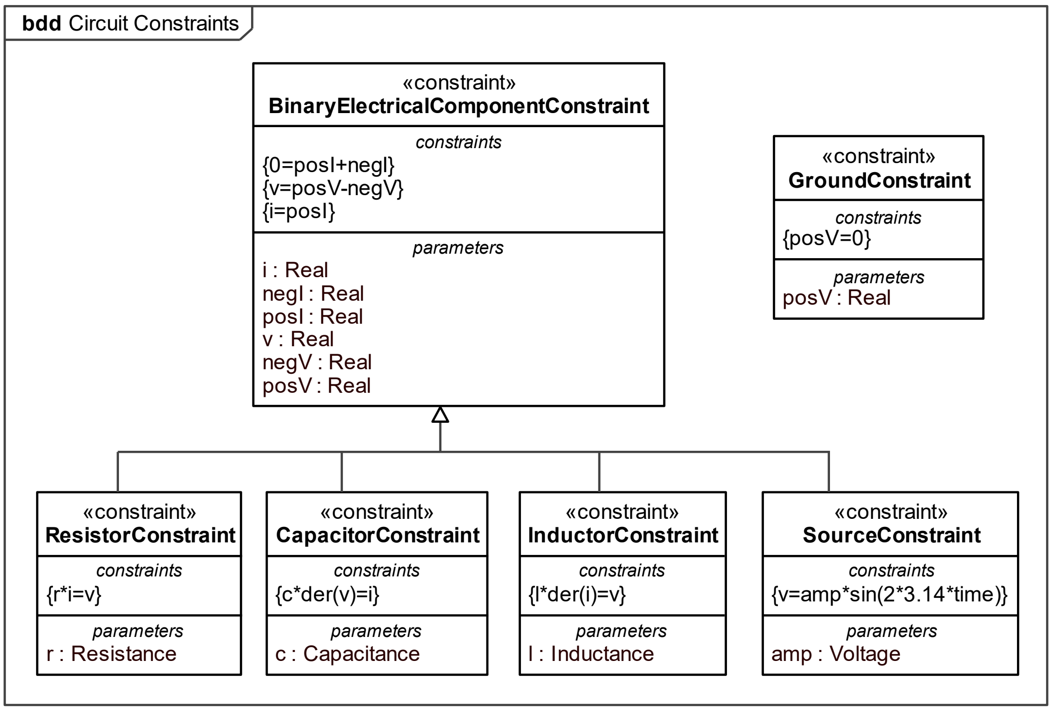


Figure 40: Circuit constraint blocks

A.2.7 Constraint Properties and Bindings

Equations in constraint blocks are applied to components using binding connectors in component parametric diagrams. Component parametric diagrams show properties typed by constraint blocks (constraint properties), as well as component and port simulation variables and constants. Binding connectors link constraint parameters to simulation variables and constants, indicating their values must be the same. Figure 41 through Figure 45 show parametric diagrams for resistors, capacitors, inductors, sources, and grounds, respectively.

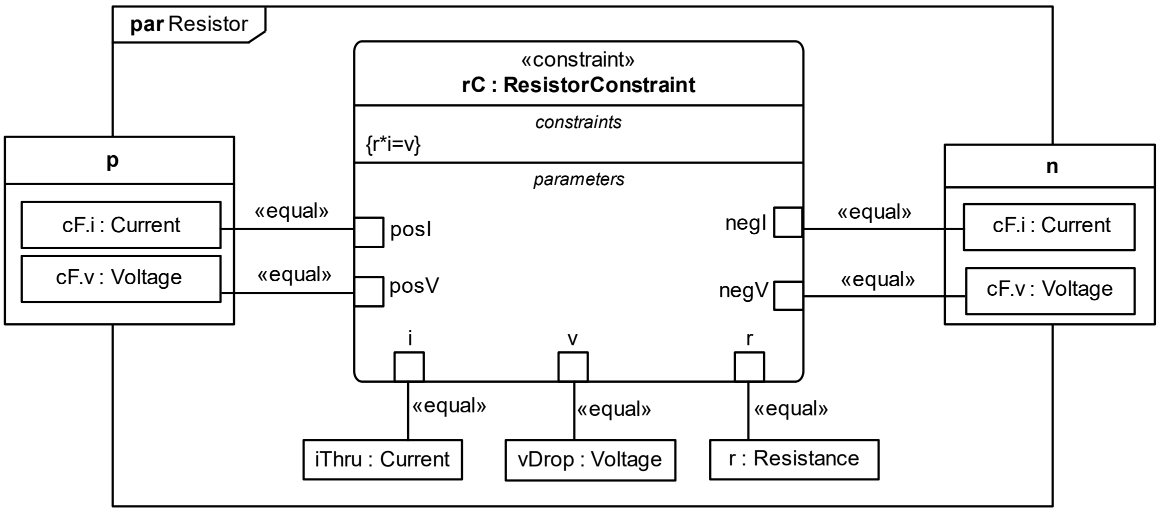


Figure 41: Parametric diagram applying the resistor constraint

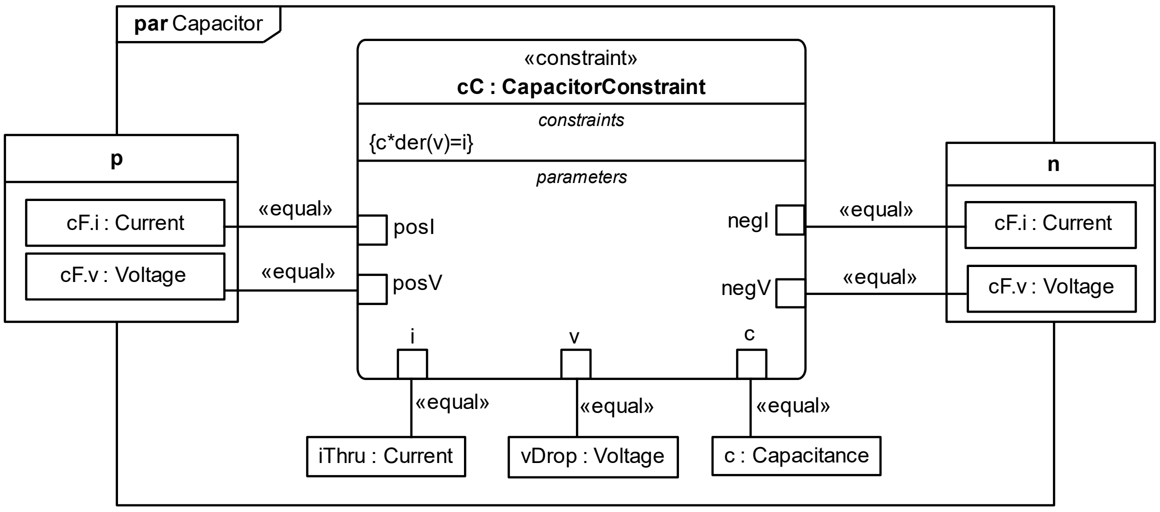


Figure 42: Parametric diagram applying the capacitor constraint

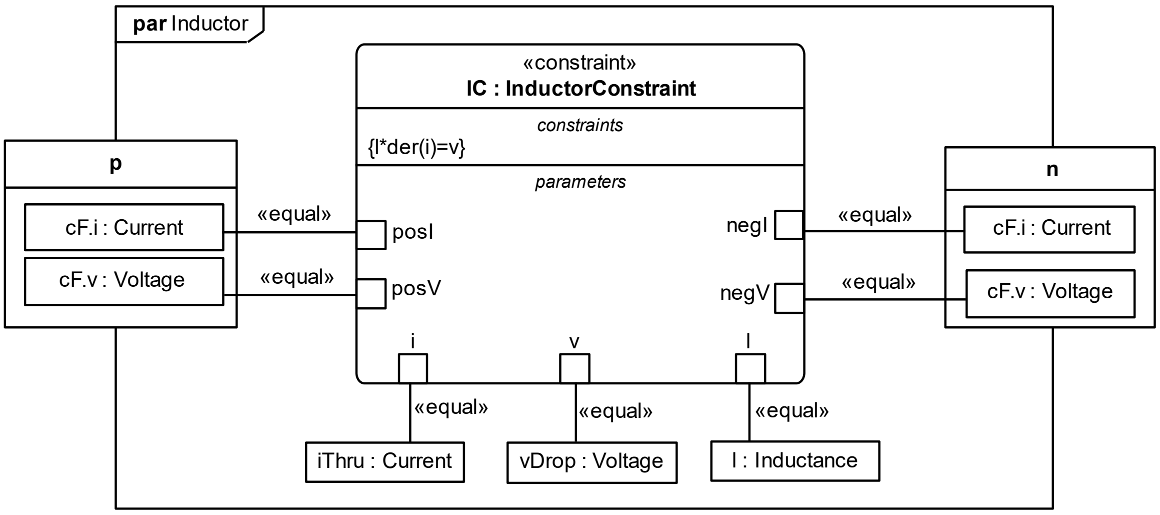


Figure 43: Parametric diagram applying the inductor constraint

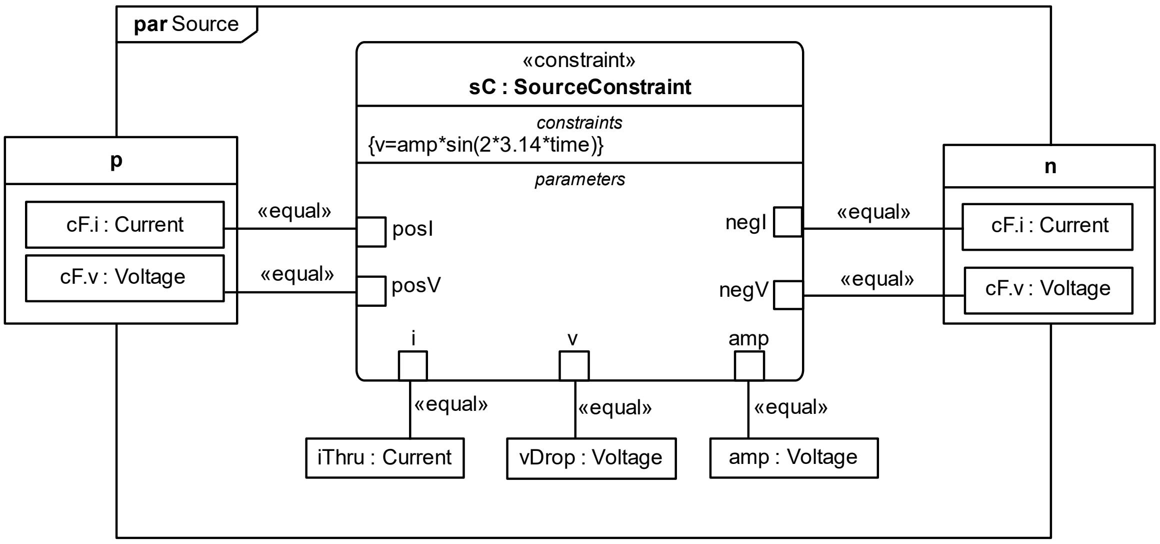


Figure 44: Parametric diagram applying the source constraint

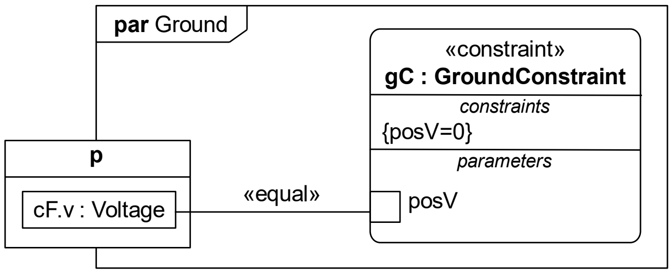


Figure 45: Parametric diagram applying the ground constraint

A1.3 Signal Processor

A.3.1 Introduction

This subannex gives a model of processing a sinusoidal variable as an example of signal flow. It does not include any physical interactions.

A.3.2 System Being Modeled

The signal processor and its testbed have a wave generator, an amplifier, high-pass and low-pass frequency filters, a mixer, and a signal sink, see Figure 46.

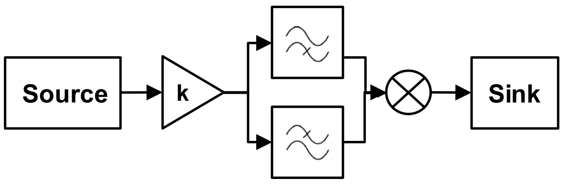


Figure 46: Signal processor example

A.3.3 Internal Structure

Figure 47 and Figure 48 show the internal structure of blocks *TestBed* and *SignalProcessor*, respectively. Part properties, typed by blocks defined in Subannex A.3.4, represent the components of the system. They are connected through ports, also defined in Subannex A.3.4, which represent signal outputs and inputs, also defined in Subannex A.3.4. Signals pass through ports in the direction shown by the arrows. Item flows on connectors indicate that the signals are real numbers.

Figure 47 connects a signal source to a signal processor, which it connects to a signal sink that displays the output. Figure 48 connects the signal processor input to an amplifier, the output of the amplifier to a high- pass filter in parallel with a low-pass filter, the outputs of the filters to a mixer, and the output of the mixer to the signal processor output. SysML initial values specify property values for components used in internal block diagrams. Figure 47shows an initial value for source amplitude *amp*, while Figure 48 shows initial values for amplifier signal gain *g* and filtering properties *xi* and *alpha* (defined in Subannex A.3.4). Simulink without Simscape does not have elements corresponding to initial values on parts below the top level system (see Subclause 10.10.4). Subannex A.5.9 shows SysML models that have the same effect as initial values and have corresponding elements in Simulink.

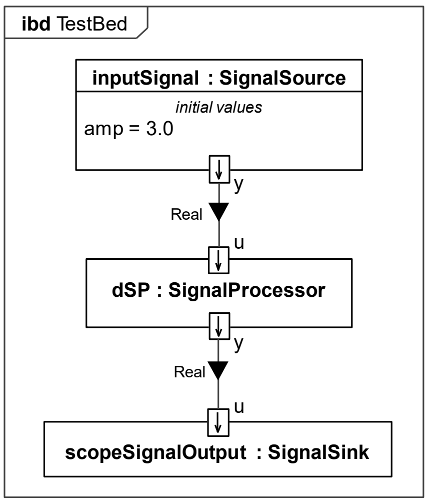


Figure 47: Internal structure of test bed from signal source to sink

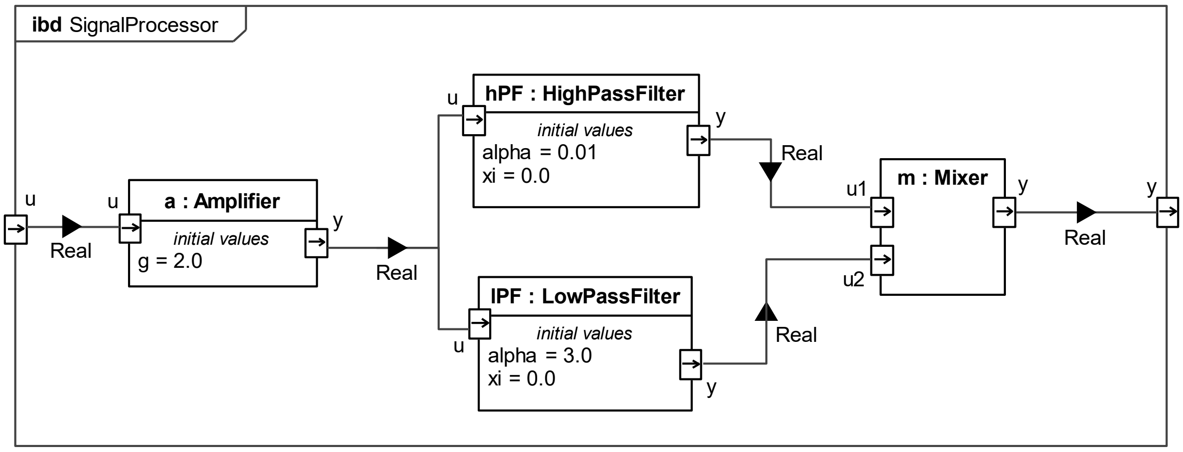


Figure 48: Internal structure of the signal processor

A.3.4 Blocks and Ports

Figure 49 and Figure 50 show block definitions for components of *TestBed* and *SignalProcessor* in Figure 47 and Figure 48, respectively. The output for *SignalSource* is named *y* and is typed by *RealSignalOutElement*, from the signal flow library (see Subclause 11.2.1). The input for *SignalSink* is named *u* and is typed by *RealSignalInElement*, also from the library. The signal processor has an input and output, transforming the signal from the source and passing it to the sink.

In Figure 50, amplifiers, low-pass filters, and high-pass filters, each have an input and an output. Since they are similar in this sense, a generalized *TwoPinSignalComponent* component has an input *u* and an output *y*. Mixers have inputs *u1* and *u2*, and an output *y*. Each kind of component has its own behaviors, defined as constraints in Subannex A.3.6. Alternatively, some of these components could be specified using the source and sink components library (see Subclause 11.3.2.7).

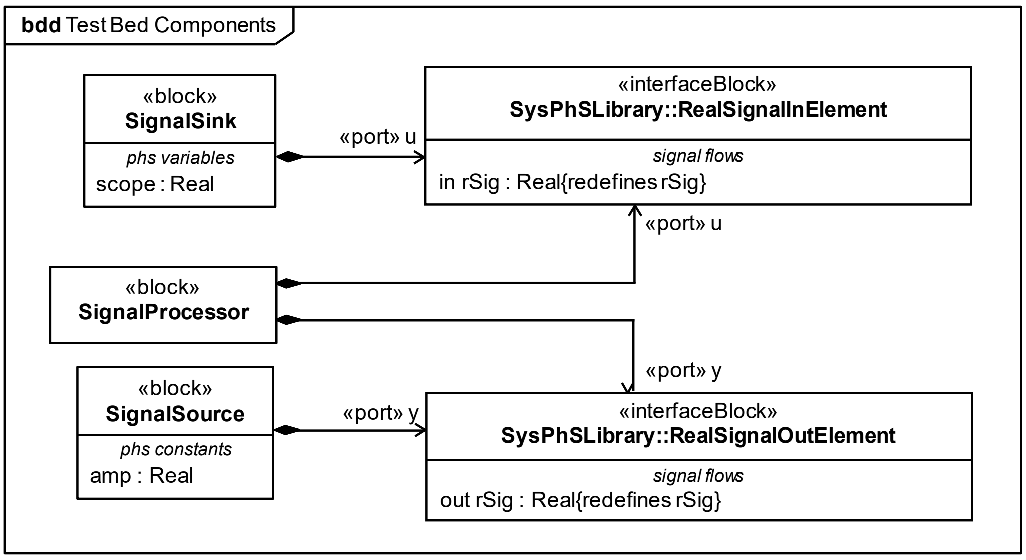


Figure 49: Total system (source to sink) blocks, ports, & component properties

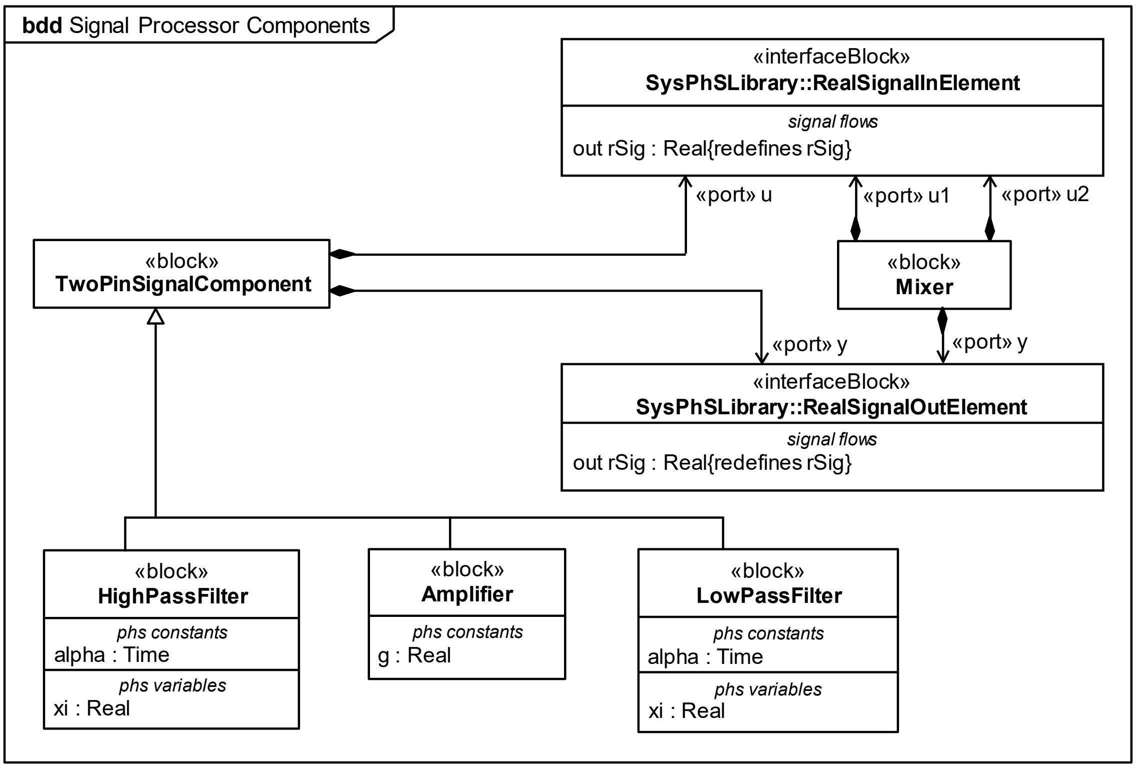


Figure 50: Signal processing system blocks, ports, & component properties

A.3.5 Properties (variables)

Signal flow is the movement of numbers between system components. These numbers might reflect physical quantities or not. In this example, they do not (see Subannex A.5 for an example where they do). Signals flowing in and out of components are modeled by ports typed by interface blocks that have flow properties typed by numbers. In this example, ports are typed by *RealSignalOutElement* and *RealSignalInElement* from the signal flow library (see Subclause 11.2.1), which both have a flow property *rSig* typed by *Real,* from SysML, as shown in Figure 49. This value type has no unit, reflecting that the signals are not measurements of physical quantities and do not follow conservation laws.

The amplifier, filters (high-pass and low-pass), signal source, and signal sink have properties *g, alpha* and *xi, amp,* and *scope,* respectively. The *amp, alpha* and *g* properties have the PhS Constant stereotype applied, specifying that their values are constant during each simulation run. The *xi* and *scope* properties have the PhSVariable stereotype applied, specifying that their values might vary during simulation.

A.3.6 Constraints (equations)

Equations define mathematical relationships between the values of numeric variables. Equations in SysML, are constraints in constraint blocks that use properties of the blocks (parameters) as variables. In this example, a constraint block *BinarySignalComponentConstraint* defines the parameters for one input (*ip*) and one output (*op*), common to amplifiers, low-pass filters, and high-pass filters, as shown in Figure 51. The amplifier, low-pass filer, and high-pass filter constraints show the input-output relationship of these components as the signal passes through them. The amplifier changes the signal strength by a factor *gain*, the low-pass filter eliminates the high-frequency components of the incoming signal, and the high-pass filter eliminates the low-frequency components of the signal. The mixer constraint specifies the relationship between its one output and the two inputs that come from the low-pass and high-pass filters. The constraint defines the output to be the average of the inputs. The source constraint specifies a sine wave signal with the parameter *amp* as its amplitude. The sink constraint displays (scopes) the output signal from the signal processor.

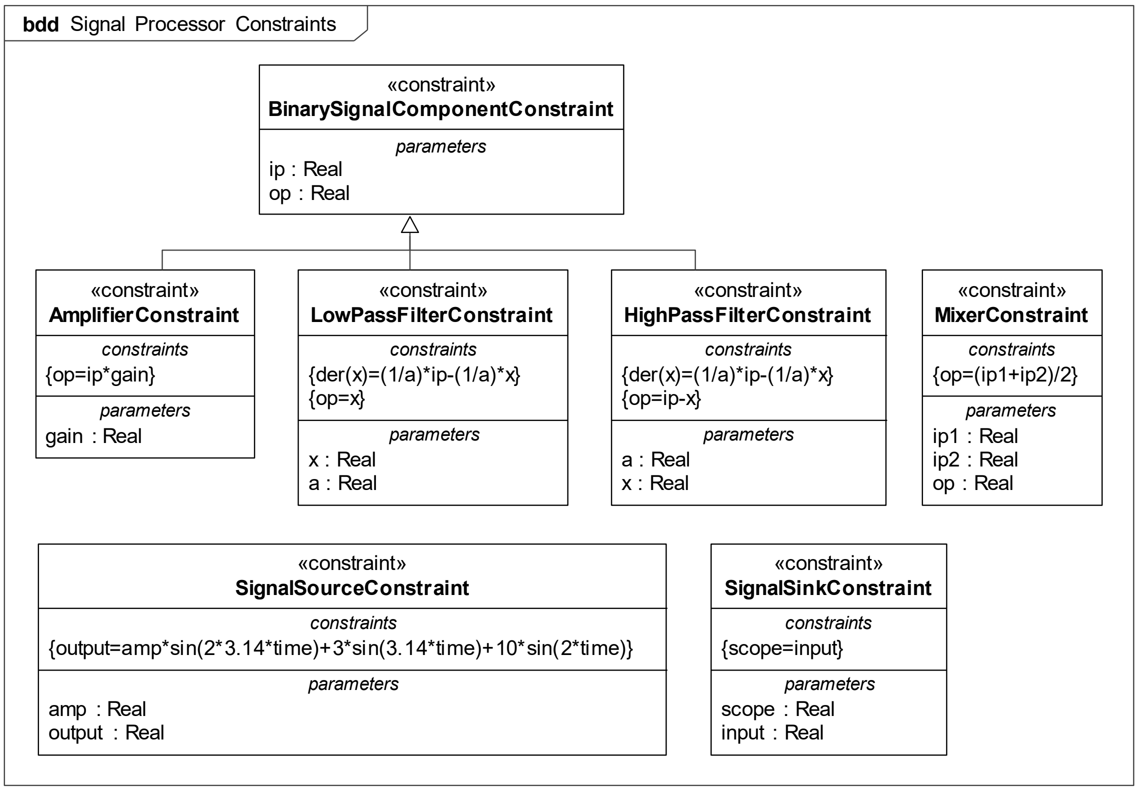


Figure 51: Signal processing system constraint blocks

A.3.7 Constraint properties and bindings

Equations in constraint blocks are applied to components using binding connectors in component parametric diagrams. Component parametric diagrams show properties typed by constraint blocks (constraint properties), as well as component and port simulation variables and constants. Binding connectors link constraint parameters to simulation variables and constants, indicating their values must be the same. Figure 52 through Figure 57 show parametric diagrams for the source, amplifier, high-pass filer, low-pass filter, mixer, and sink, respectively.

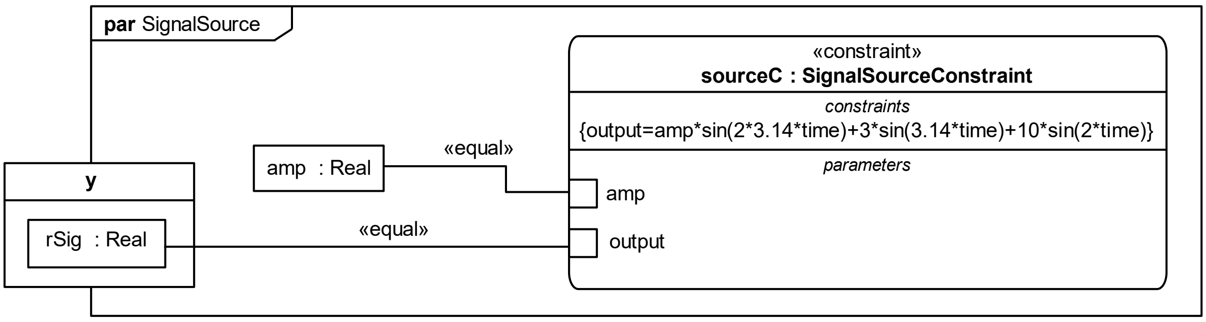


Figure 52: Parametric diagram applying signal source constraint

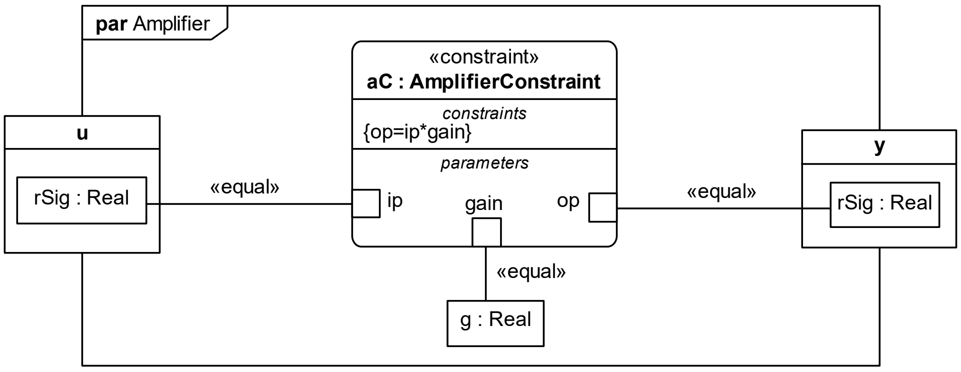


Figure 53: Parametric diagram applying the amplifier constraint

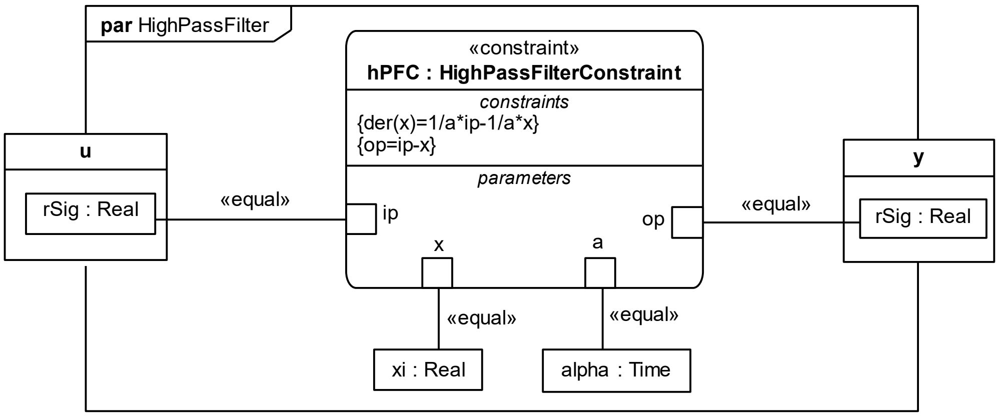


Figure 54: Parametric diagram applying the high-pass filter constraint

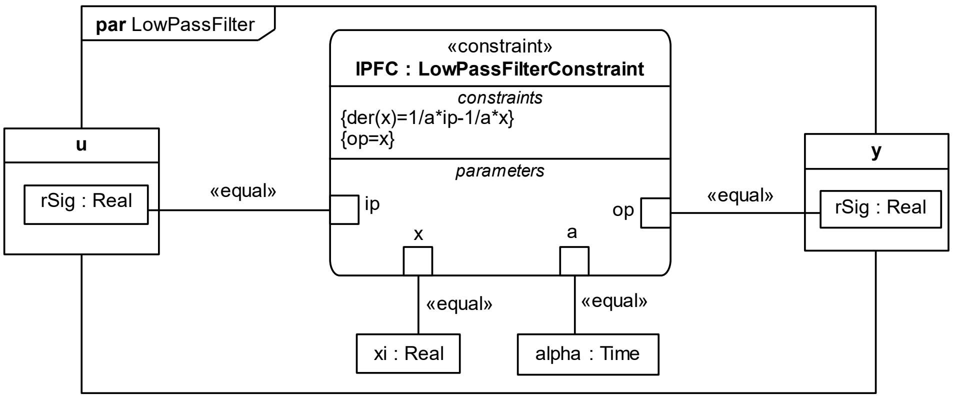


Figure 55: Parametric diagram applying the low-pass filter constraint

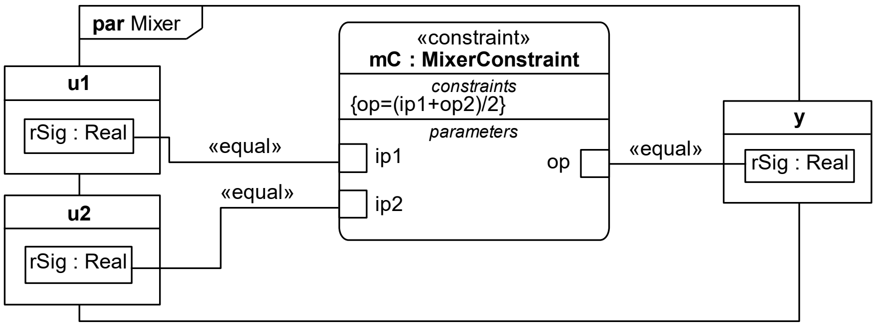


Figure 56: Parametric diagram applying the mixer constraint

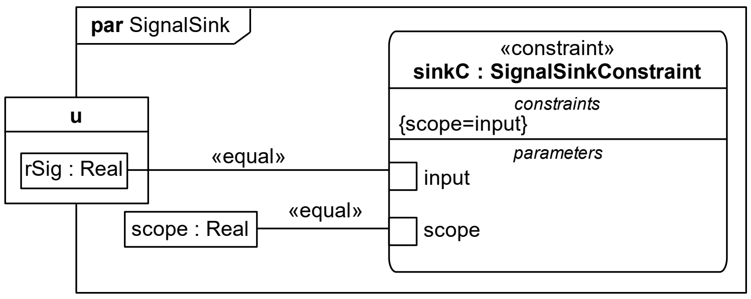


Figure 57: Parametric diagram applying the signal sink constraint

A1.4 Hydraulics

A.4.1 Introduction

This subannex gives a model of a simple hydraulic system as an example of physical interaction (fluid flow). It does not include any signal flows.

A.4.2 System Being Modeled

The hydraulic system has three components: two fluid reservoir tanks and a pipe for connecting these tanks, see Figure 58.

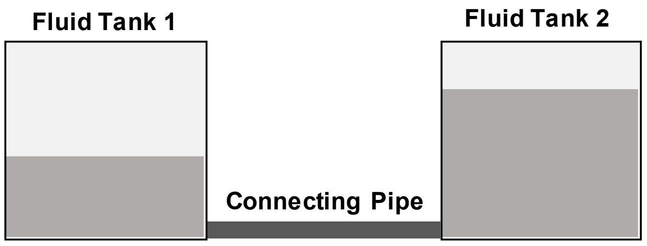


Figure 58: Hydraulics example

A.4.3 Internal Structure

Figure 59 shows the internal structure of a *ConnectedTanks* block. Part properties, typed by blocks defined in Subannex A.4.4, represent components in this system. They are connected to each other through ports, which represent openings in the tanks and pipe, also defined in Subannex A.4.4. Item flows on connectors indicate fluid passes through the ports and between the parts. The diagram connects a tank to each end of a pipe.

SysML initial values specify property values for components used in internal block diagrams. Figure 59 shows initial values for fluid density, gravity, tank surface area, pipe radius, pipe length, and dynamic viscosity of the fluid (properties defined in Subannex A.4.4). An alternative for specifying initial values of part properties in the *ConnectedTanks* is to specialize it and redefine the part properties with default values for various configurations (see Subannex A.5.9).

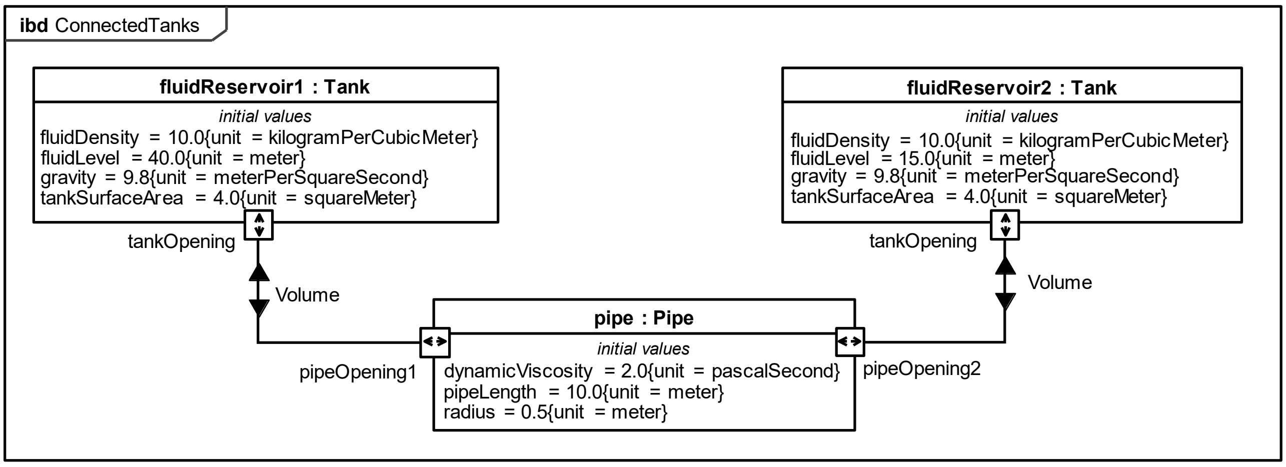

ibd ConnectedTanks
fluidReservoir1 : Tank
initial values
fluidDensity = 10.0{unit = kilogramPerCubicMeter}
fluidLevel = 40.0{unit = meter}
gravity = 9.8{unit = meterPerSquareSecond}
tankSurfaceArea = 4.0{unit = squareMeter}
tankOpening
fluidReservoir2 : Tank
initial values
fluidDensity = 10.0{unit = kilogramPerCubicMeter}
fluidLevel = 15.0{unit = meter}
gravity = 9.8{unit = meterPerSquareSecond}
tankSurfaceArea = 4.0{unit = squareMeter}
tankOpening
Volume
Volume
pipe : Pipe
initial values
dynamicViscosity = 2.0{unit = pascalSecond}
pipeLength = 10.0{unit = meter}
radius = 0.5{unit = meter}
pipeOpening1
pipeOpening2

Figure 59: Internal structure of hydraulics system

A.4.4 Blocks and Ports

Figure 60 shows block definitions for components of *ConnectedTanks* in Figure 59. Tanks and pipes have openings for fluid to pass through, one for tanks and two for pipes. The openings are represented by ports of type *VolumeFlowElement*, from the physical interaction library (see Subclause 11.2.2). Each type of component has its own behaviors, defined as constraints in A.4.6.

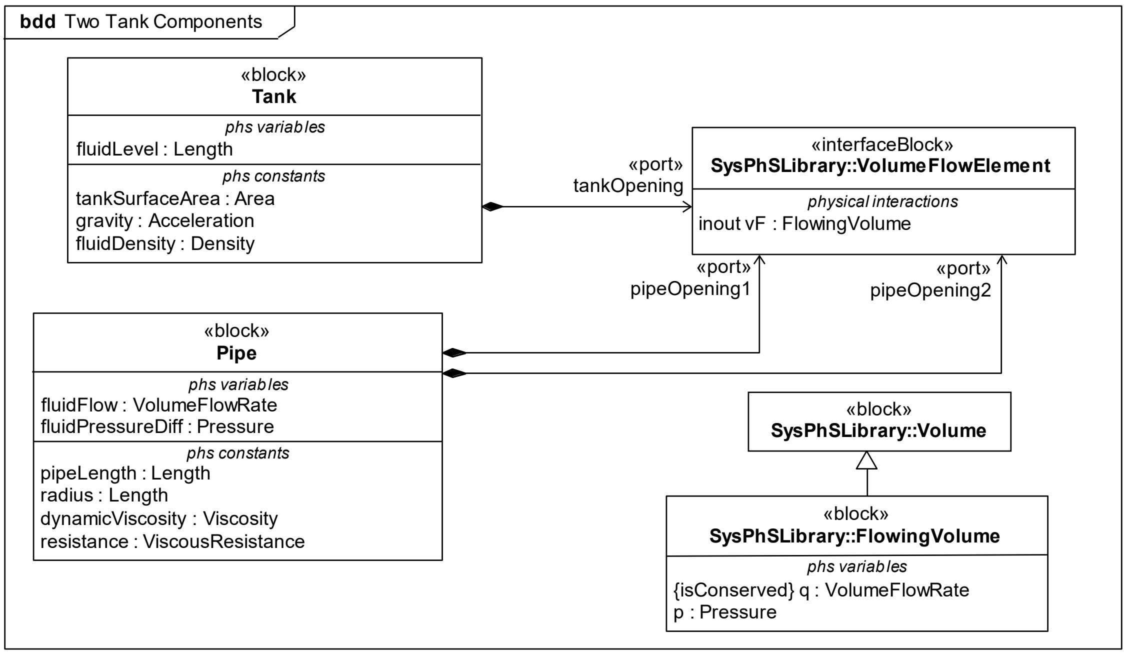


Figure 60: Hydraulics blocks, ports, & component properties

A.4.5 Properties (variables)

Physical interaction is the movement of physical substances between system components, modeled in terms of conserved characteristics of the substances. In this example, volume is the conserved characteristic of fluid moving between the tanks (fluids are substances that can be treated as volumes because they are incompressible, but otherwise do not resist deformation). Movement of substances is described by numeric variables for flow rate and potential to flow of their conserved characteristics. In this example, movement of volumes is characterized by a volume per unit time variable for the flow rate and a pressure variable for potential to flow. The flow rate variable is conserved (values on ends of the interaction sum to zero) and the potential variable is not (values on ends of the interaction are the same). This is modeled in three parts:

- Conserved physical characteristics are modeled as blocks directly specialized from *ConservedQuantityKind* in the physical interaction library (see Subclause 11.2.2), *Volume* in this example.
- Flow variables are modeled as properties with the PhSVariable stereotype applied on specializations of conserved quantity kind blocks. In this example, the flow rate and potential PhSVariables are *q* and *p* on *Flowing Volume* (*q* marked as *isConserved*), respectively, typed by *VolumeFlowR ate* and *Pressure*, respectively, all from the physical interaction library.
- Flows in and out of components are modeled by ports typed by interface blocks that have flow properties typed by flowing conserved quantity kinds. In this example, ports are typed by *VolumeFlowElement* from the physical interaction library, which has a flow property *vF* typed by *Flowing Volume*, as shown in Figure 60. The *Tank* block has a *tankOpening* port and the *Pipe* block has *pipeOpening1* and *pipeOpening2* ports, all typed by *VolumeFlowElement.*

Behavior of the pipe in this example is described by the fluid pressure and volume flow rate at the openings. The fluid pressure is given by the property *fluidPressureDiff* (difference in pressure between its two openings) and the volume flow rate is given by the property *fluidFlow* (the volume of fluid going in our out the openings per unit time). These two properties are typed by *Pressure* and *VolumeFlowRate*, respectively, from the physical interaction library (see Subclause 11.2.2), and have the PhSVariable stereotype applied, specifying that their values might vary during simulation.

The tank has properties *fluidLevel, tankSurfaceArea, gravity,* and *fluidD ensity* typed by *Length, Area, Acceleration,* and *Density,* respectively. The property *fluidLevel* has the PhSVariable stereotype applied, because the amount of fluid in the tank can vary during simulation, but the other properties have the

PhS Constant stereotype applied, specifying that their values do not change during each simulation run.

The pipe has properties *pipeLength, radius, dynamic Viscosity,* and *resistance* typed by *Length, Length, Viscosity,* and *ViscousResistance,* respectively, and all with the PhSConstant stereotype applied.

A.4.6 Constraints (equations)

Equations define mathematical relationships between the values of numeric variables. Equations in SysML, are constraints in constraint blocks that use properties of the blocks (parameters) as variables. In this example, constraint blocks *PipeConstraint* and *TankConstraint* define parameters and equations for pipes and tanks, respectively, as shown in Figure 61.

The pipe constraints specify that the pressure *pressureDiff* across it is equal to the difference of fluid pressures *opening1Pressure* and *opening2Pressure* at each end of the pipe. The fluid flow rate through the pipe, *fluidFlow,* is proportional to the pressure difference by the constant *resistance*, which depends on the geometric properties of the pipe as well as fluidic properties. The magnitude of fluid flow rate through the pipe *fluidFlow* is the same as the magnitude of flow rates *opening1FluidFlow* and *opening2FluidFlow* going through the pipe's openings, though the values differ in sign. The sum of the fluid flow rates going through the two pipe openings is zero (the fluid is assumed to be incompressible).

The tank constraints specify that the pressure in the tank, *pressure* depends on the height of the fluid level in the tank, *fluidHeight*, as well as properties of the fluid, *fluidDensity*. Also, the fluid flow in the tank, *fluidFlow,* is related to the change in the fluid height level *fluidHeight* over time and the cross-sectional surface area of the tank, *surfaceArea*.

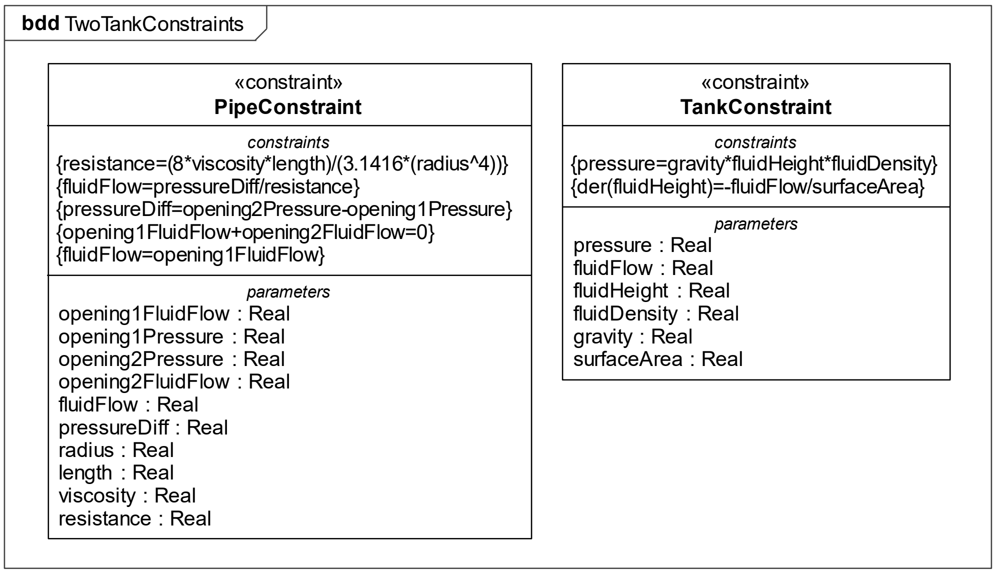


Figure 61: Hydraulics model constraint blocks

A.4.7 Constraint properties and bindings

Equations in constraint blocks are applied to components using binding connectors in component parametric diagrams. Component parametric diagrams show properties typed by constraint blocks (constraint properties), as well as component and port simulation variables and constants. Binding connectors link constraint parameters to simulation variables and constants, indicating their values must be the same. Figure 62 and Figure 63 show the parametric diagrams of the tank and the pipe, respectively.

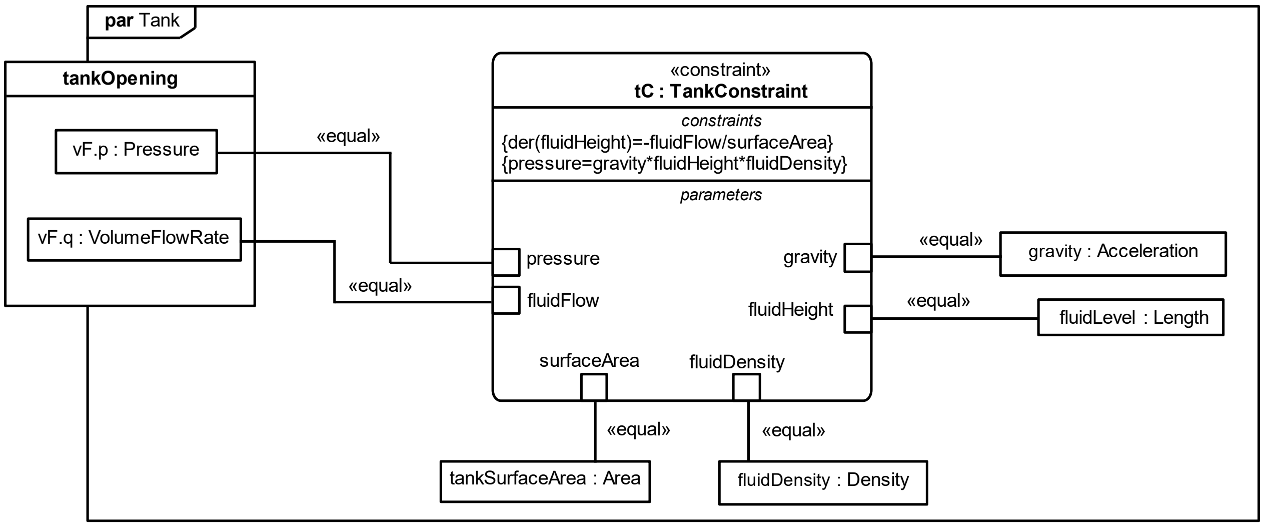


Figure 62: Parametric diagram applying the tank constraint

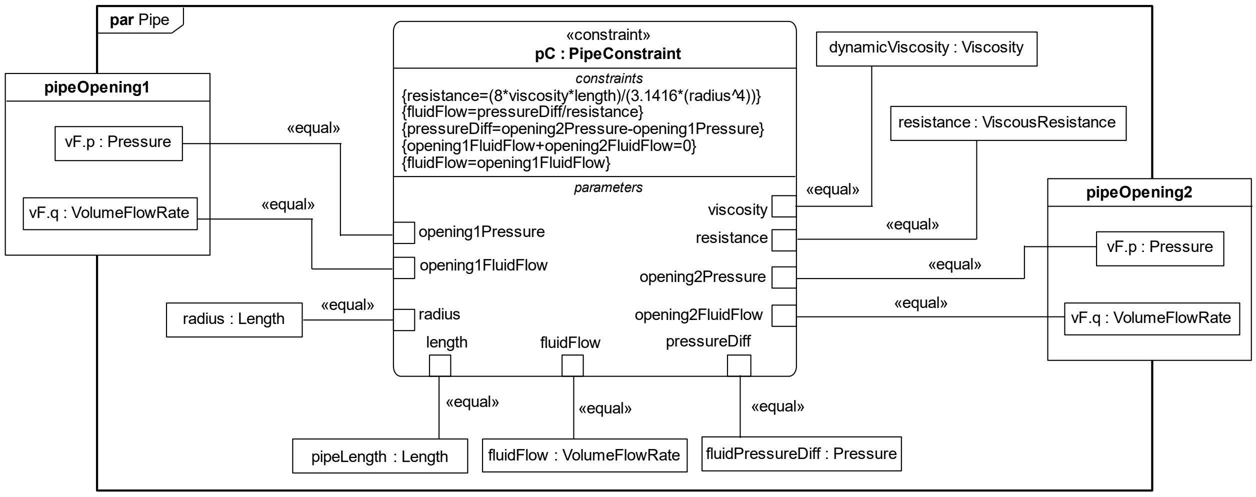

par Pipe
pipeOpening1
vF.p : Pressure
vF.q : VolumeFlowRate
«constraint»
pC : PipeConstraint
constraints
{resistance=(8*viscosity*length)/(3.1416*(radius^4))}
{fluidFlow=pressureDiff/resistance}
{pressureDiff=opening2Pressure-opening1Pressure}
{opening1FluidFlow+opening2FluidFlow=0}
{fluidFlow=opening1FluidFlow}
parameters
opening1Pressure
opening1FluidFlow
radius
length
fluidFlow
pressureDiff
viscosity
resistance
opening2Pressure
opening2FluidFlow
dynamicViscosity : Viscosity
resistance : ViscousResistance
pipeOpening2
vF.p : Pressure
vF.q : VolumeFlowRate
radius : Length
pipeLength : Length
fluidFlow : VolumeFlowRate
fluidPressureDiff : Pressure
«equal»

Figure 63: Parametric diagram applying the pipe constraint

A1.5 Humidifier

A.5.1 Introduction

This subannex gives a model of a room humidifier as an example of signal flows and state machines. Some signals in the example reflect physical quantities, but this is not physical interaction in the sense of physical substances with flow rates and potentials, as in Subannexes A.2 and A.4.

A.5.2 System Being Modeled

The total humidifier system has two main components: the humidified room and the humidifier, see Figure 64. The humidifier uses information about the room's humidity level to determine how much vapor to input to the room. The humidifier includes a water tank, a heater controller, and a vapor generation plant.

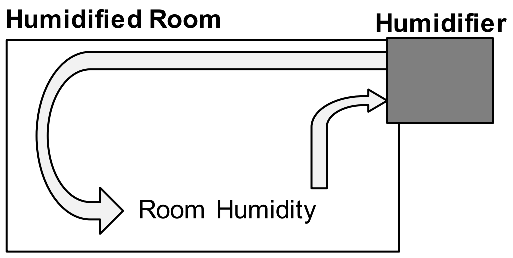


Figure 64: Total humidifier system example

A.5.3 Internal Structure

Figure 65 through Figure 71 show the internal structure of the total humidifier system and its components through seven nested internal block diagrams. The internal structure of the block *HumidifierSystem* shown in Figure 65 uses the blocks *HumidifiedRoom* and *Humidifier*. These two blocks have their own internal structures. The internal structure of *HumidifiedRoom* depicted in Figure 66 uses a block *RelativeHumidity*, which has an internal structure depicted in Figure 67. The internal structure of *Humidifier* in Figure 68 uses a block *VaporGenerationPlant*, which has an internal structure shown in Figure 69. The internal structure of *VaporGenerationPlant* uses blocks *Heating* and *Evaporation*, which have internal structures depicted in Figure 70 and Figure 71, respectively. The blocks used in these diagrams are introduced in Subannex A.5.4.

Part properties, typed by blocks defined Subannex A.5.4, represent the components of the system. They are connected to each other through ports, also defined in Subannex A.5.4, which represent signal outputs and inputs. Signals pass through ports in the direction shown by the arrows. Item flows on connectors indicate that the signals are real numbers.

Figure 65 connects the humidified room to the humidifier, showing vapor signals flowing from the humidifier to the room and humidity signals flowing from the room to the humidifier. Figure 66 directs vapor, saturation vapor pressure, and humidity signals flowing into the room to a relative humidity part that calculates the humidity flowing out of the room.

Figure 67 directs incoming vapor signals to a vapor pressure calculation part, which connects to the relative humidity calculation to output pressure signals. This figure also directs incoming saturation vapor pressure signals to the relative humidity calculation, as well as humidity signals to a humidity balance part, which connects to the relative humidity calculation to output a humidity change signal, which is directed to the output of this internal structure.

Figure 68 transforms humidity signals flowing to the humidifier into vapor signals flowing out of the humidifier. This is done using a heater control state machine, a usage scenario state machine, another controller state machine, information from the water tank's water volume, and information from the vapor generation plant. The state machines for the heater control, control, and usage scenario parts in Figure 68 are explained in A.5.8.

Figure 69 directs incoming heater power ratio signals to the vapor generation plant calculation part and incoming water fan signals to the radiation part. Connectors between the vapor generation plant calculation and radiation parts and the heating and evaporation parts result in vapor signal outputs from the evaporation part and temperature signal outputs from the heating part.

Figure 70 directs energy signals to the temperature increase part, which connects to the heating calculation to output temperature-increase signals, which is directed to the output of this internal structure. Figure 70 directs input energy and temperature signals to evaporation calculation parts, one of which outputs vapor signals for the internal structure.

Initial values for the properties of components in Figure 66 through Figure 71 in Subannex A.5.4 cannot be specified in internal block diagrams, as in the other subannexes, at least if Simulink is one of the platforms. Simulink without Simscape does not have elements corresponding to initial values on parts below the top- level system (see Subclause 10.10.4), and Simscape has no corresponding elements for state machines (see 10.12.4). Subannex A.5.9 shows how to get the effect of initial values in this example by specializing blocks and redefining their properties with default values.

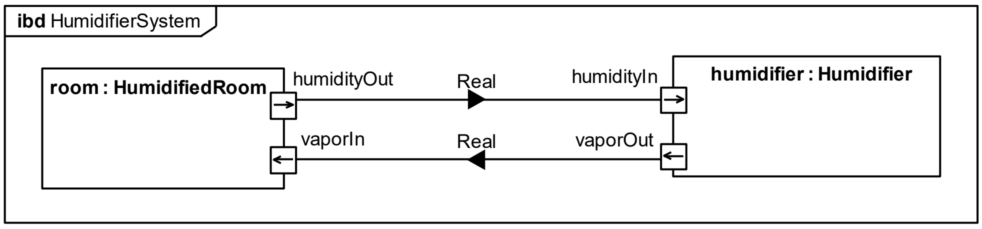


Figure 65: Internal structure of the total humidifier system

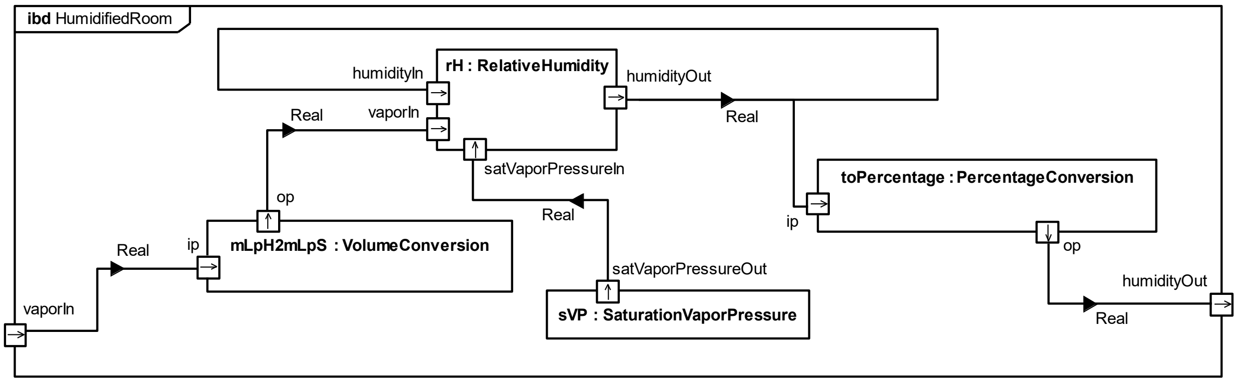


Figure 66: Internal structure of the humidified room

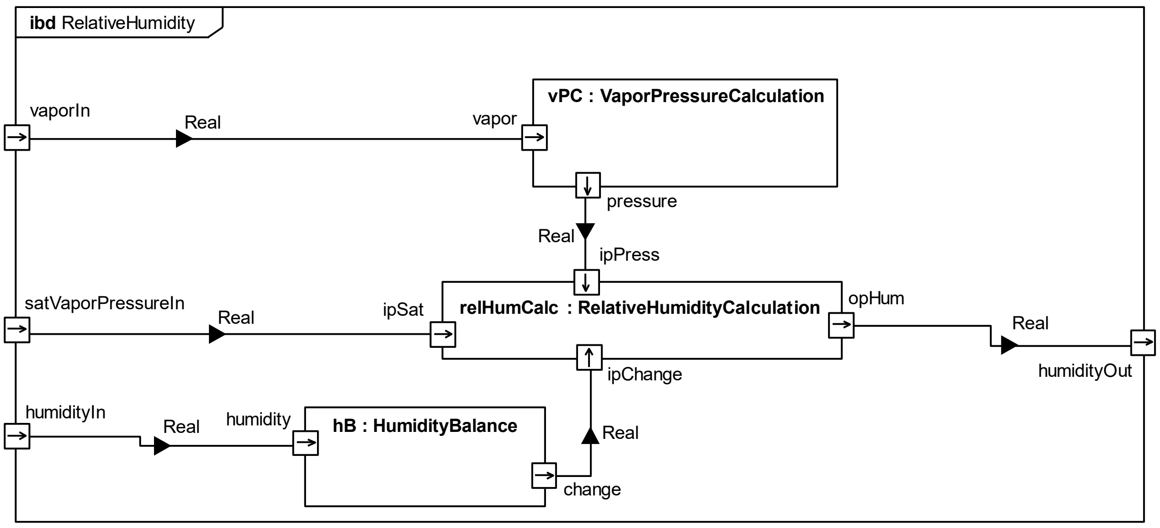


Figure 67: Internal structure of relative humidity

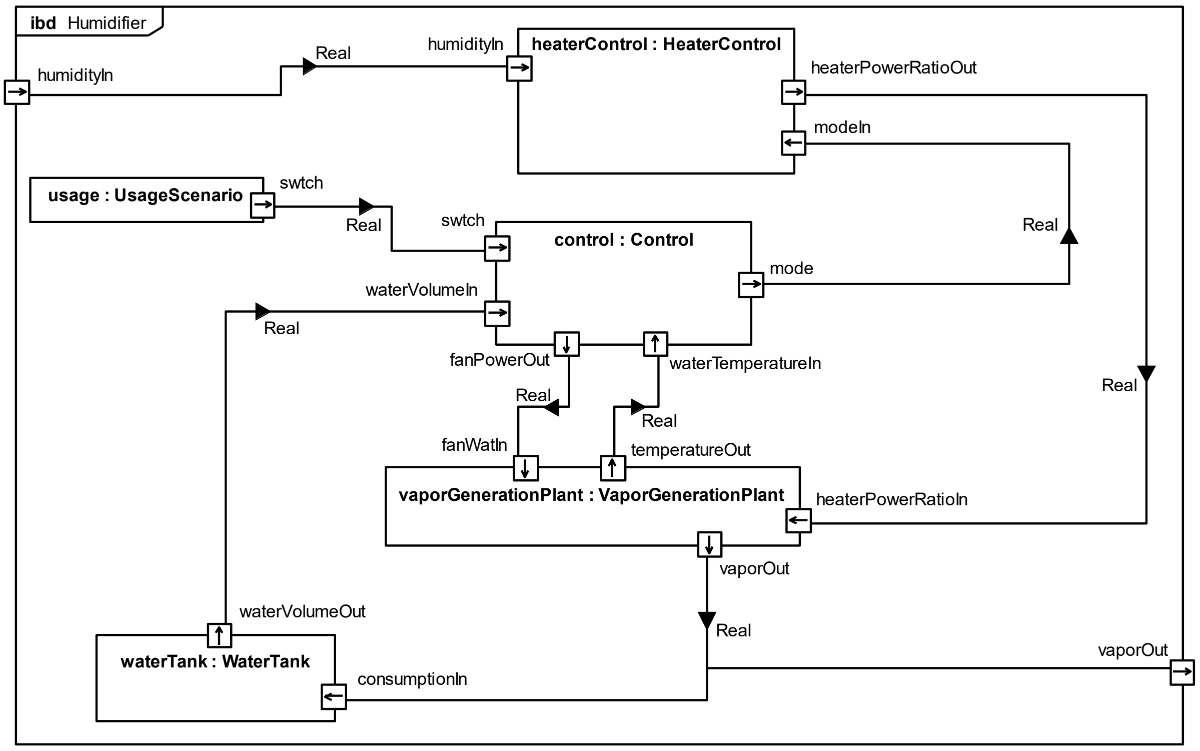


Figure 68: Internal structure of the humidifier

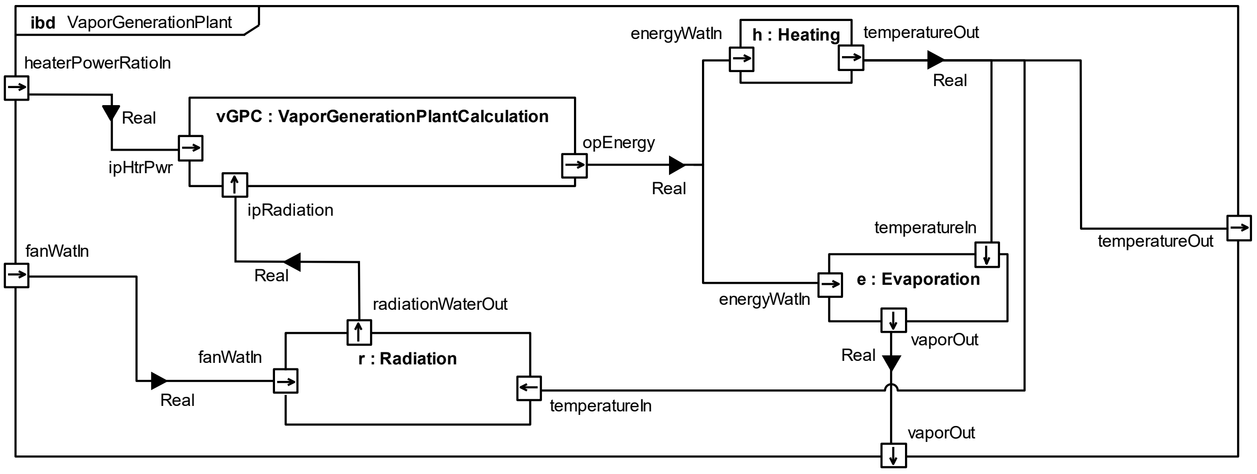


Figure 69: Internal structure of the vapor generation plant

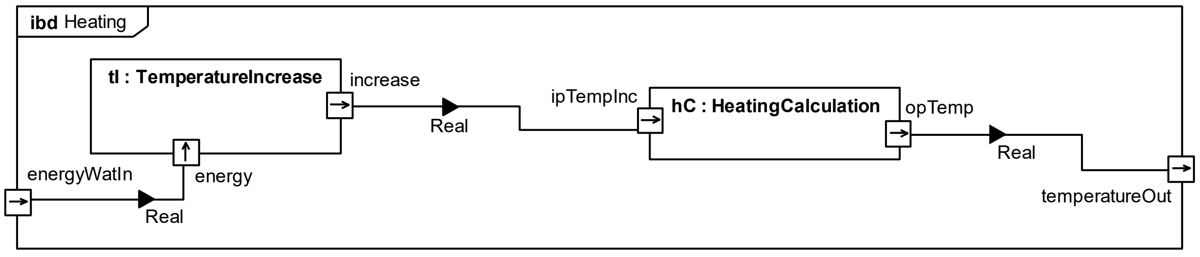


Figure 70: Internal structure of heating

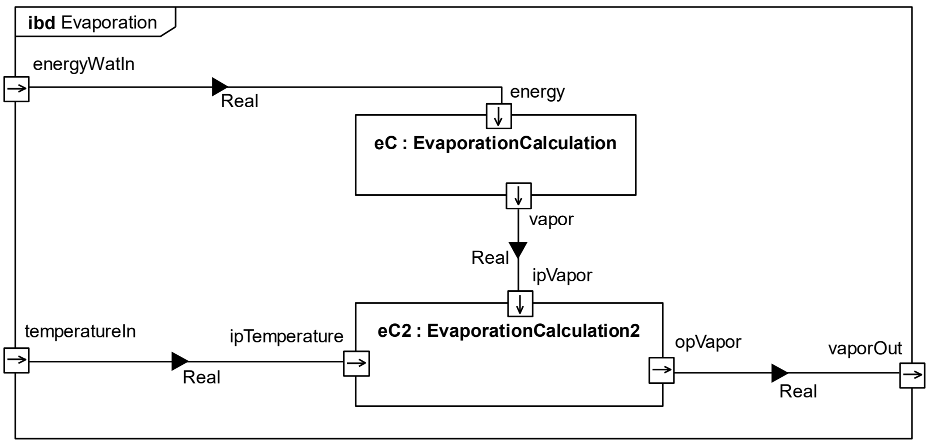


Figure 71: Internal structure of evaporation

A.5.4 Blocks and ports

Figure 72 through Figure 78 show block definitions for component used in the internal block diagrams shown in Figure 65 through Figure 71, respectively (one each for the total humidifier system, humidified room, relative humidity, humidifier, vapor generation plant, heating, and environment components). All ports are typed by *RealSignalInElement* from the signal flow library (see Subclause 11.2.1). A tilde (~) next to a port name indicates that it receives signals (conjugated port type), otherwise the port sends signals (the tilde normally appears before the type name, after a colon, but port types are omitted from the figures for brevity, because they are all the same; compare to the signal port types in Subannex A.3. Component blocks that do not have internal block diagrams in A.5.3 have their behaviors defined as constraints in Subannex A.5.6.

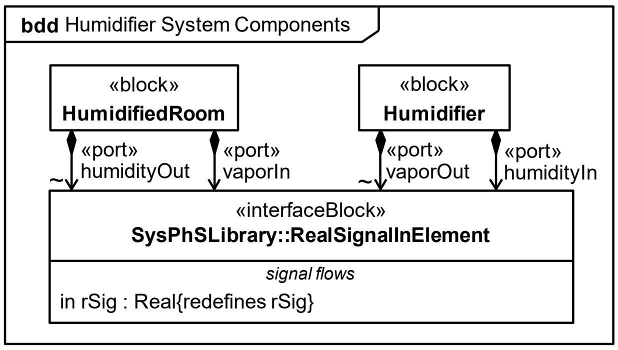


Figure 72: Total humidifier system blocks, ports, & component properties

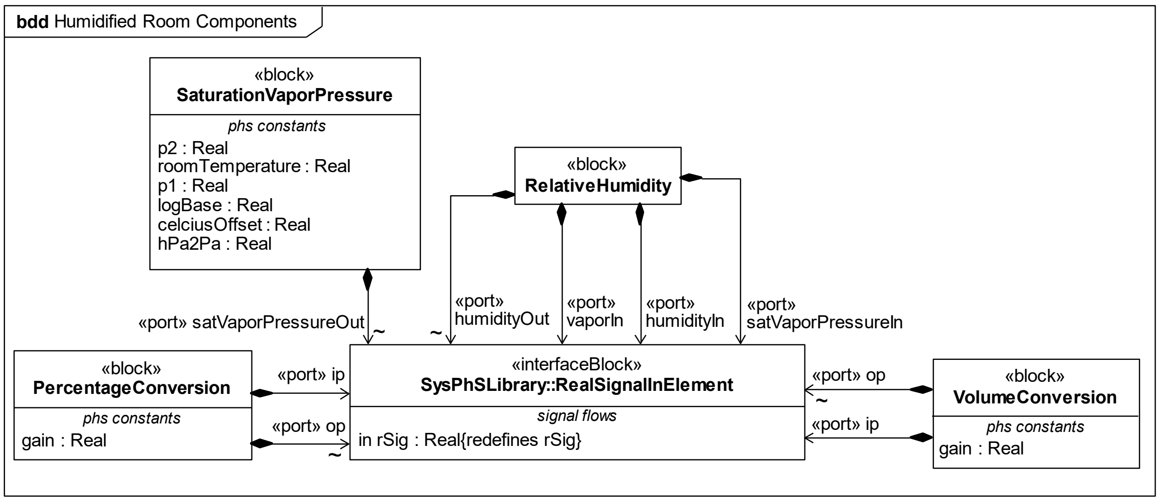


Figure 73: Humidified room blocks, ports, & component properties

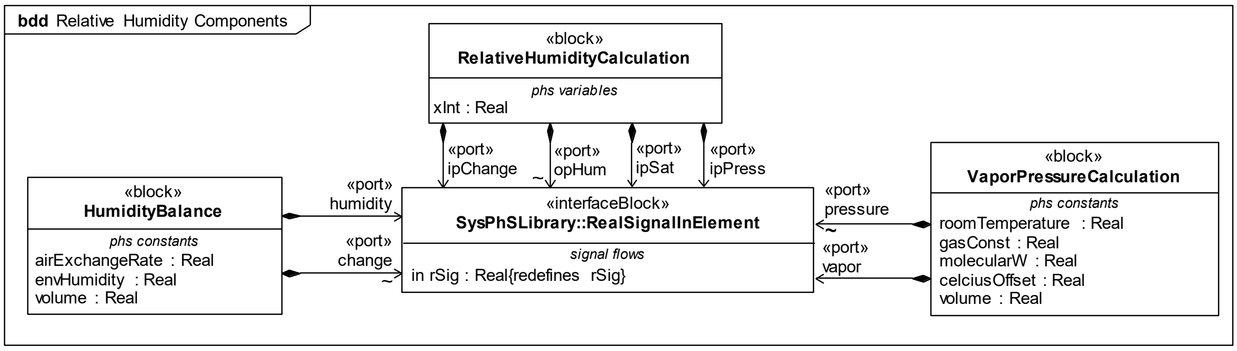


Figure 74: Relative humidity blocks, ports, & component properties

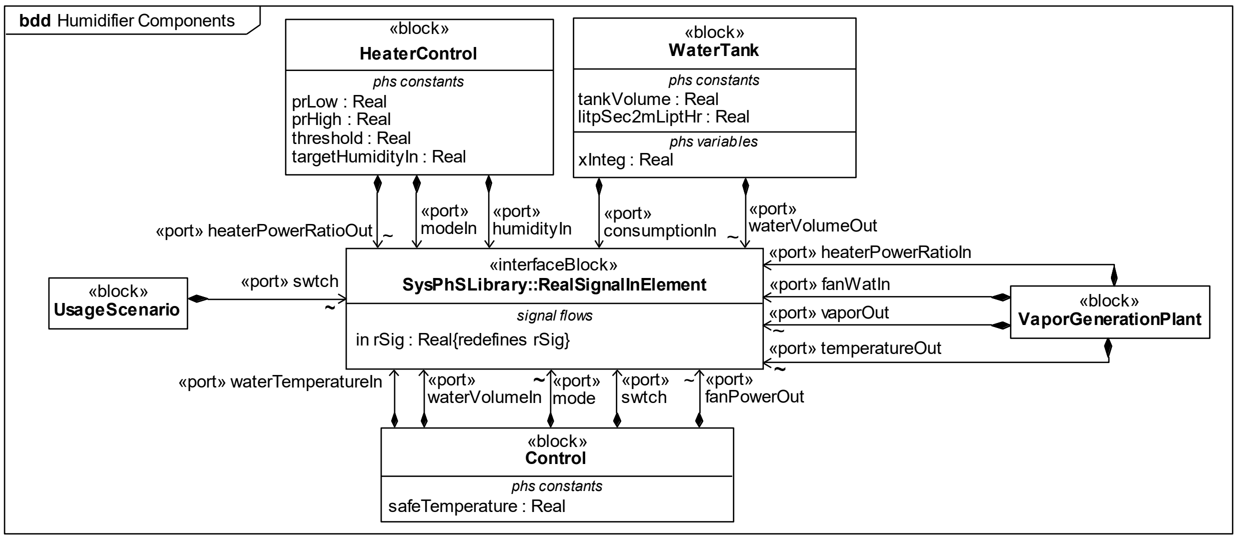


Figure 75: Humidifier blocks, ports, & component properties

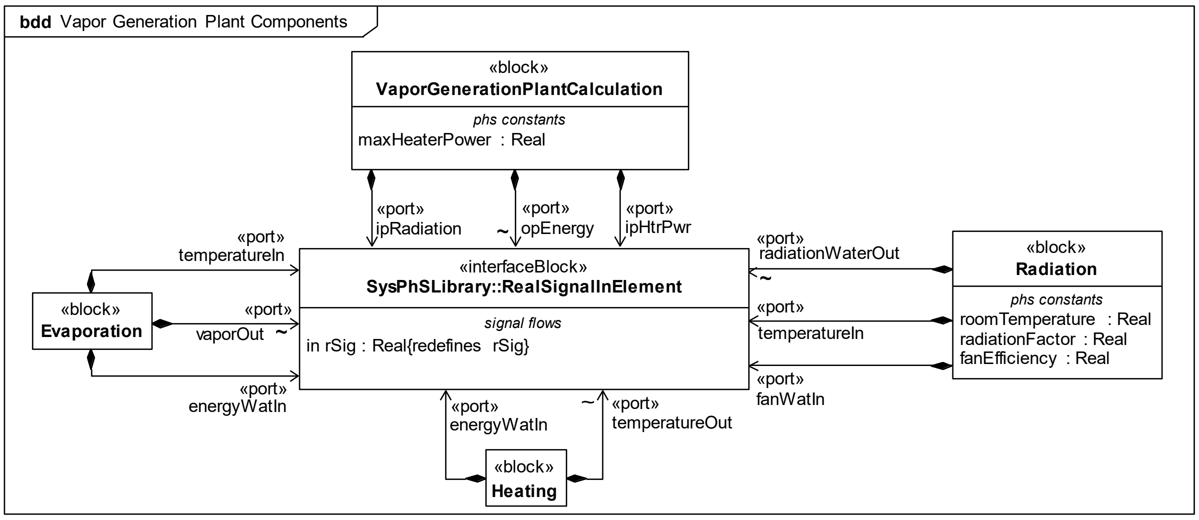


Figure 76: Vapor generation plant blocks, ports, & component properties

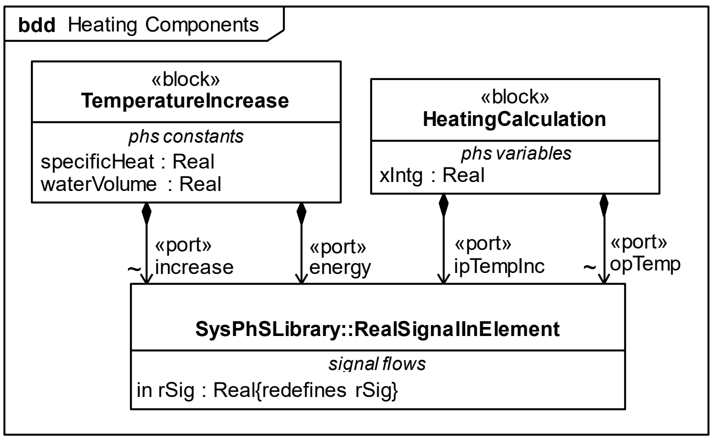


Figure 77: Heating blocks, ports, & component properties

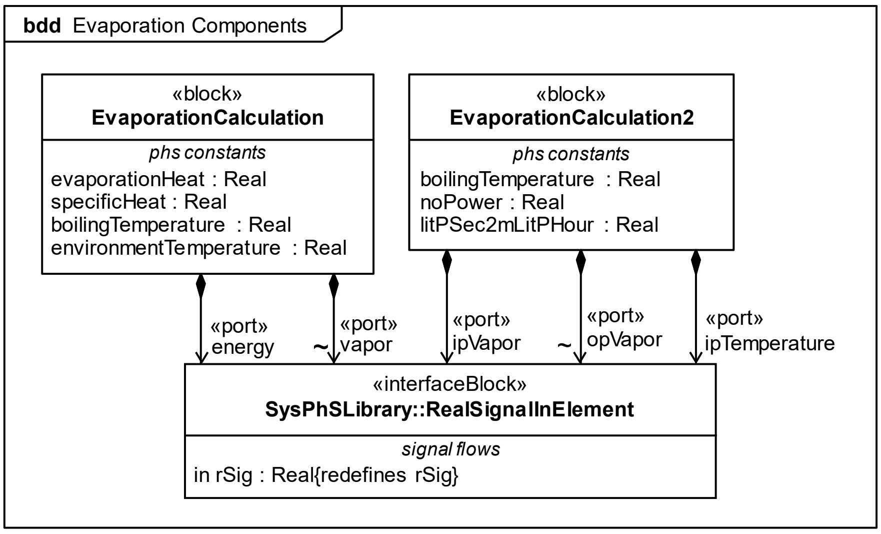


Figure 78: Evaporation blocks, ports, & component properties

A.5.5 Properties (variables)

Signals flow is the movement of numbers between system components. These numbers might reflect physical quantities or not. In this example, they do (see Subannex A.3 for an example where they do not). Signals flowing in and out of components are modeled by ports typed by interface blocks that have flow properties typed by numbers. In this example, ports are typed by *RealSignalInElement* from the signal flow library (see Subclause 11.2.1), which has a flow property *rSig* typed by *Real,* from SysML, as shown in Figure 72. This value type has no unit, even when they reflect physical quantities, and the values do not follow conservation laws.

The blocks *RelativeHumidityCalculation* (Figure 74)*, WaterTank* (Figure 75), and *HeatingCalculation* (Figure 77) have properties with PhSVariable stereotypes applied, specifying that the value of these properties may vary during simulation. The blocks *Saturation VaporPressure* (Figure 73)*, PercentageConversion* (Figure 73)*, VolumeConversion* (Figure 73), *HumidityBalance* (Figure 74)*, VaporPressureCalculation* (Figure 74)*, WaterTank* (Figure 75)*, HeaterControl* (Figure 75)*, Control* (Figure 75), *Radiation* (Figure 76)*, VaporGenerationPlantCalculation* (Figure 76)*, TemperatureIncrease* (Figure 77)*, EvaporationCalculation2* (Figure 78)*,* and *EvaporationCalculation* (Figure 78)*,* have properties with PhS Constant stereotypes applied, specifying that the value of these properties are constant during each simulation run.

A.5.6 Constraints (equations)

Equations define mathematical relationships between the values of numeric variables. Equations in SysML, are constraints in constraint blocks that use properties of the blocks (parameters) as variables. In this example, the constraint blocks in Figure 79 each define parameters and constraints for a component block in Figure 73 through Figure 78: *VolumeConversion, PercentageConversion,* and *Saturation VaporPressure* in Figure 73; *RelativeHumidityCalculation, VaporePressureCalculation,* and *HumidityBalance* in Figure 74; *WaterTank* in Figure 75; *Radiation* and *VaporGenerationPlantCalculation* in Figure 76; *HeatingCalculation* and *TemperatureIncrease* in Figure 77; and *EvaporationCalculation* and *EvaporationCalculation2* in Figure 78. The constraint blocks have the name of their components with the suffix "*-Constraint*" added. The constraints specify manipulation of signals between inputs and outputs of their component block.

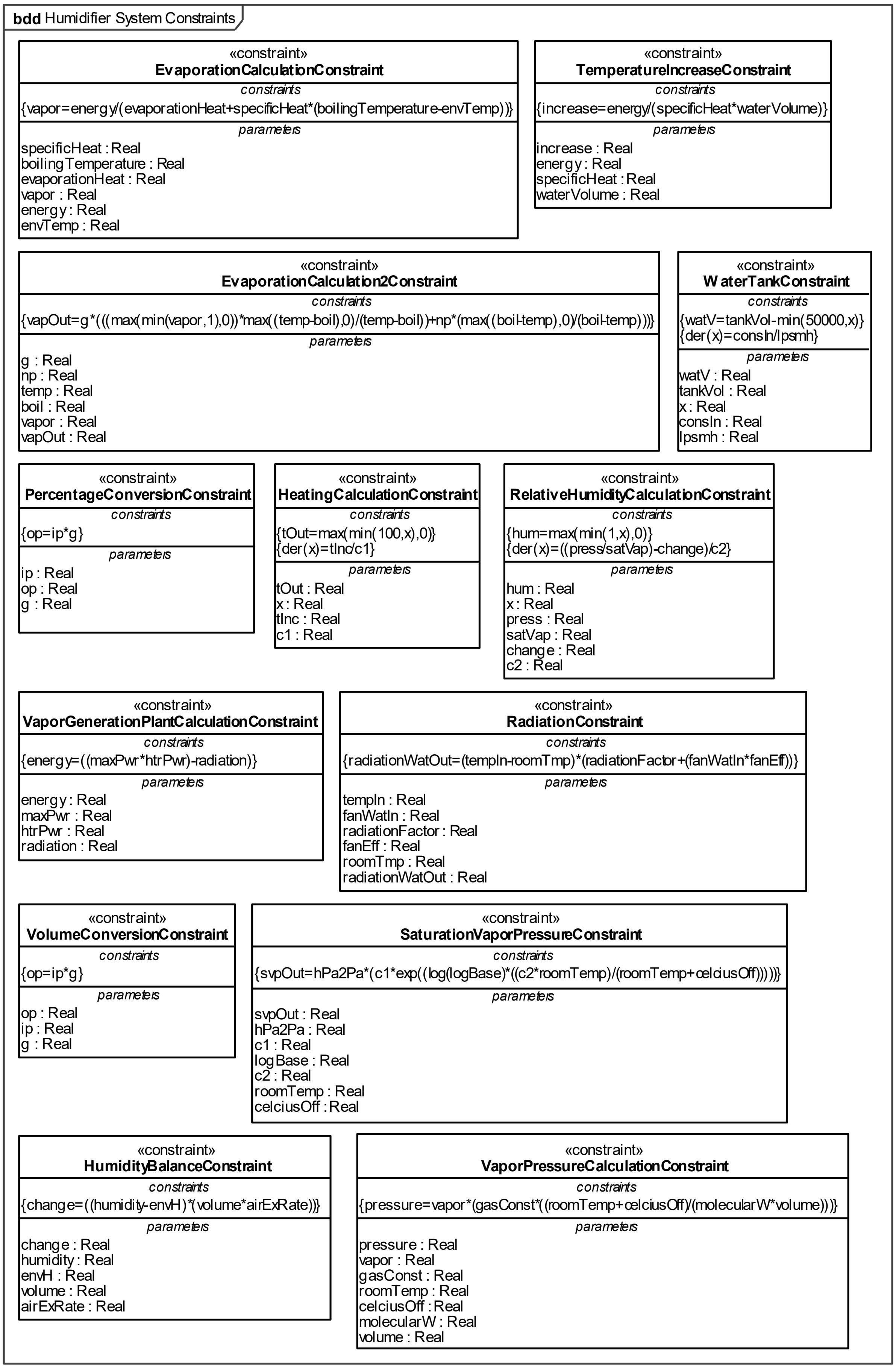


Figure 79: Humidifier constraint blocks

A.5.7 Constraint Properties & Bindings

Equations in constraint blocks are applied to components using binding connectors in component parametric diagrams. Component parametric diagrams show properties typed by constraint blocks (constraint properties), as well as component and port simulation variables and constants. Binding connectors link constraint parameters to simulation variables and constants, indicating their values must be the same. Figure 80 through Figure 92 show the parametric diagrams for the blocks *VolumeConversion, PercentageConversion, Saturation VaporPressure, HumidityBalance, RelativeHumidityCalculation, VaporPressureCalculation, VaporGenerationPlantCalculation, Radiation, HeatingCalculation, TemperatureIncrease, EvaporationCalculation, EvaporationCalculation2,* and *WaterTank,* respectively.

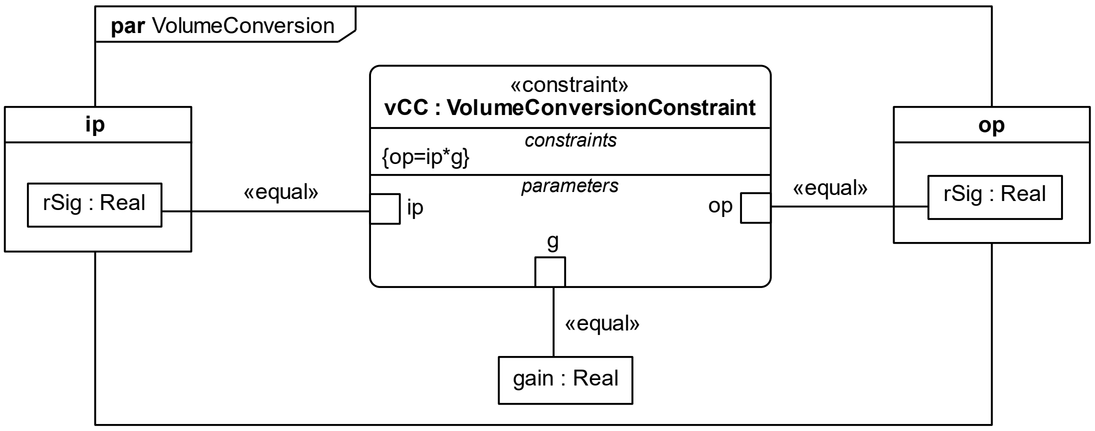


Figure 80: Parametric diagram applying the volume conversion constraint

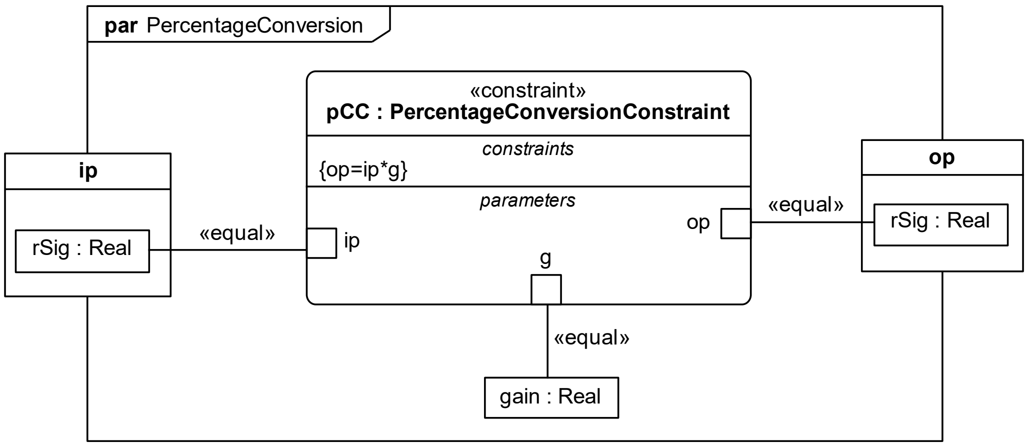


Figure 81: Parametric diagram applying the percentage conversion constraint

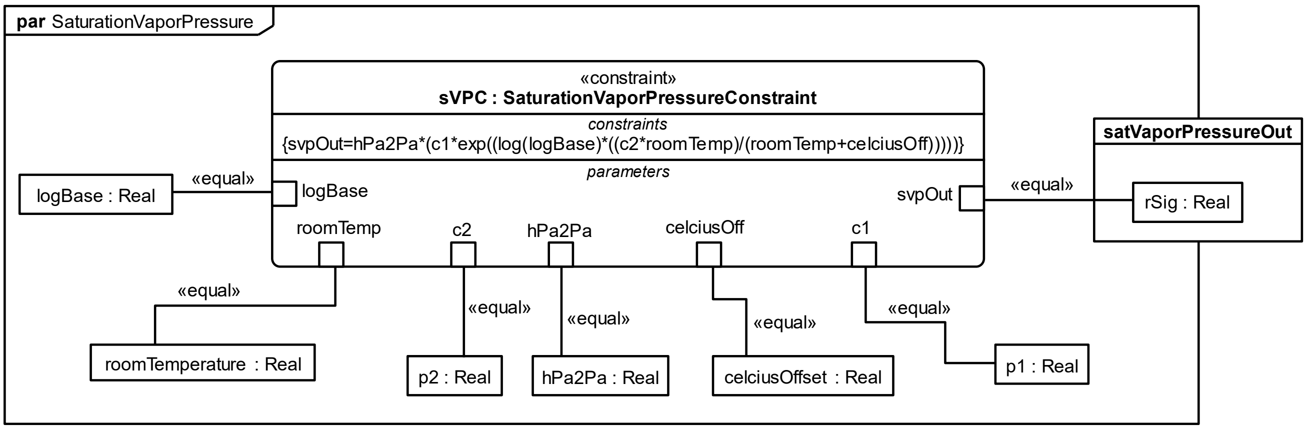


Figure 82: Parametric diagram applying the saturation vapor pressure constraint

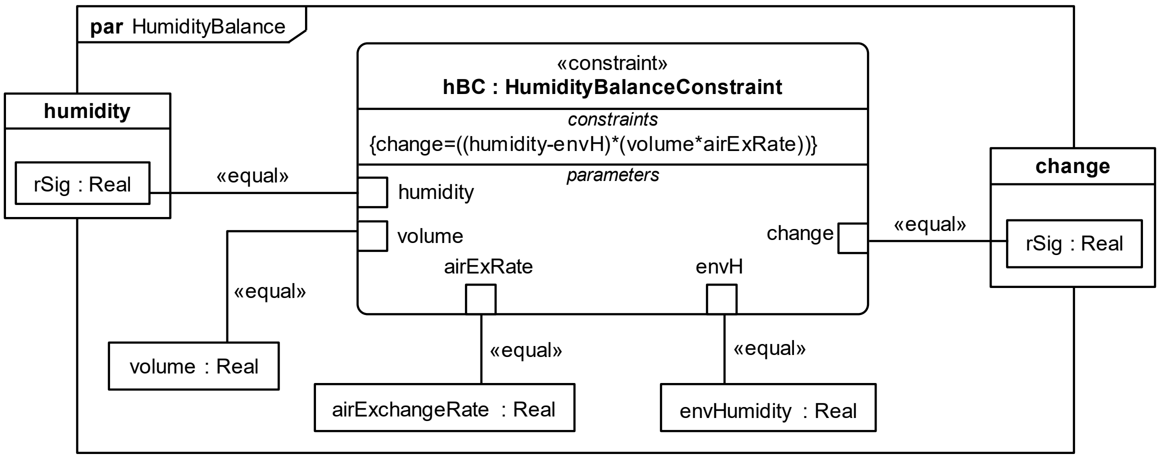


Figure 83: Parametric diagram applying the humidity balance constraint

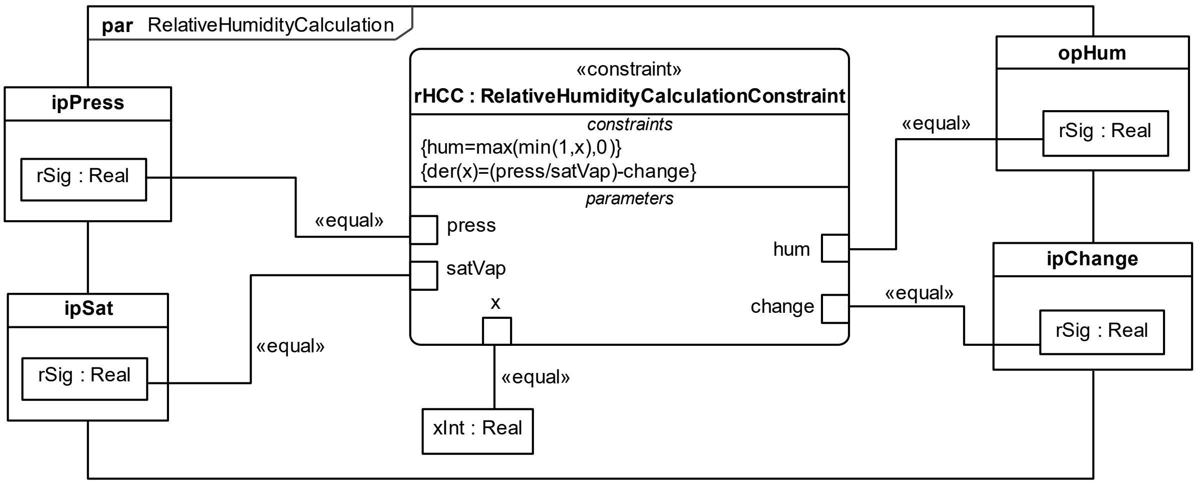


Figure 84: Parametric diagram applying the relative humidity calculation constraint

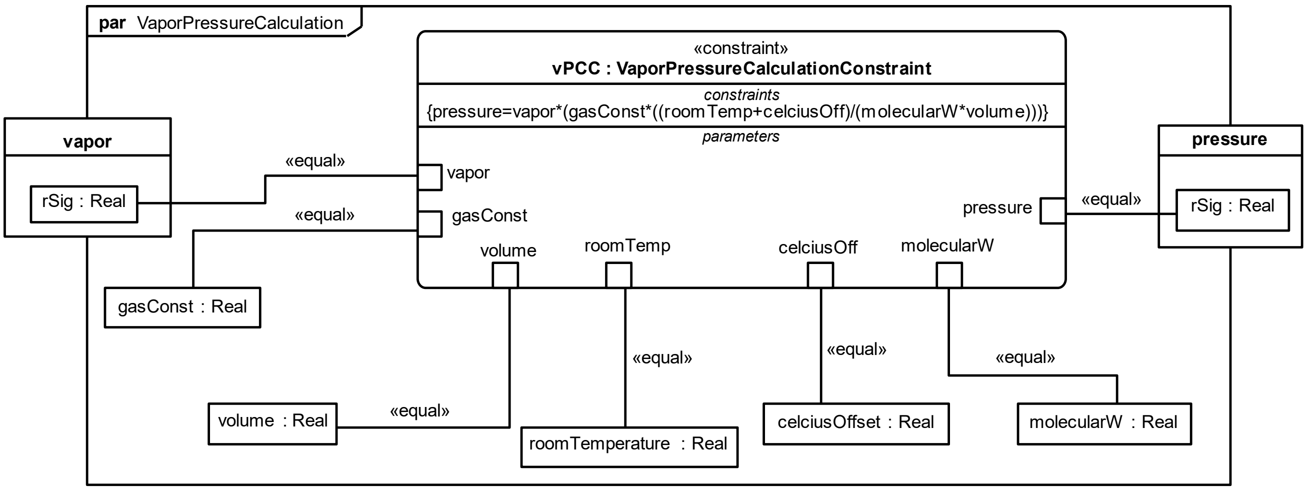


Figure 85: Parametric diagram applying the vapor pressure calculation constraint

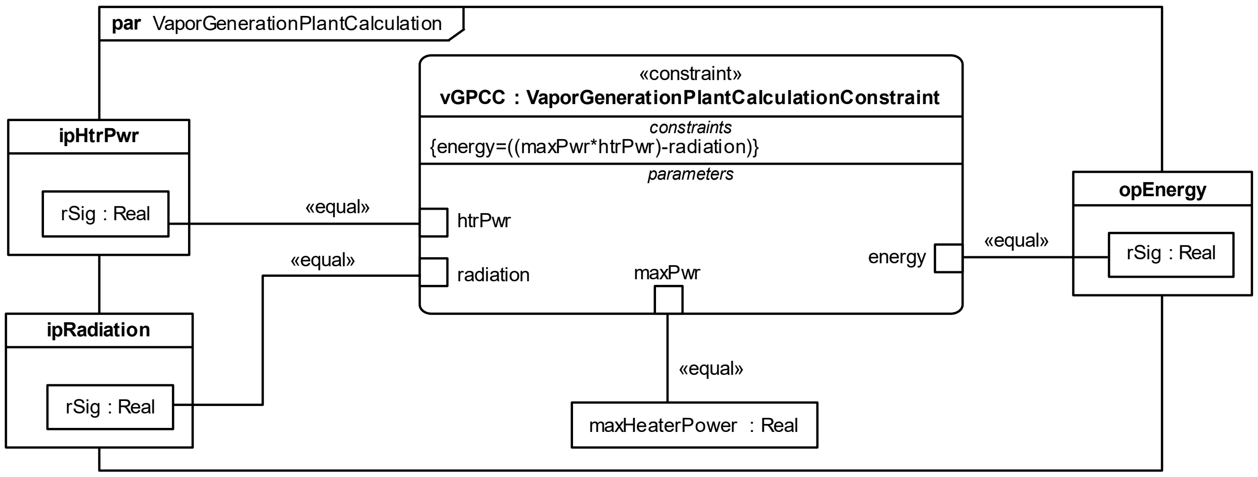


Figure 86: Parametric diagram applying the vapor generation plant calculation constraint

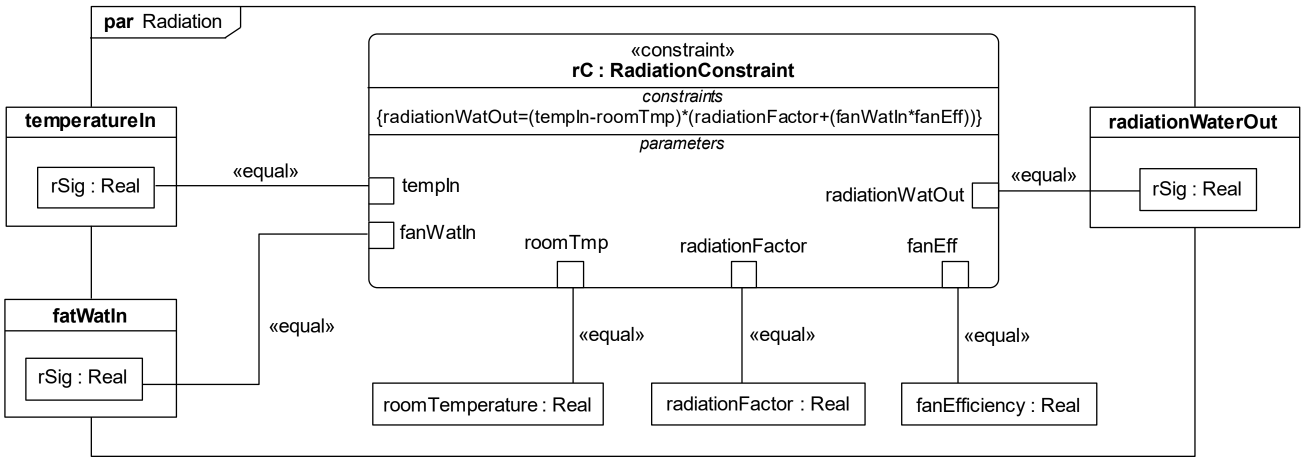


Figure 87: Parametric diagram applying the radiation constraint

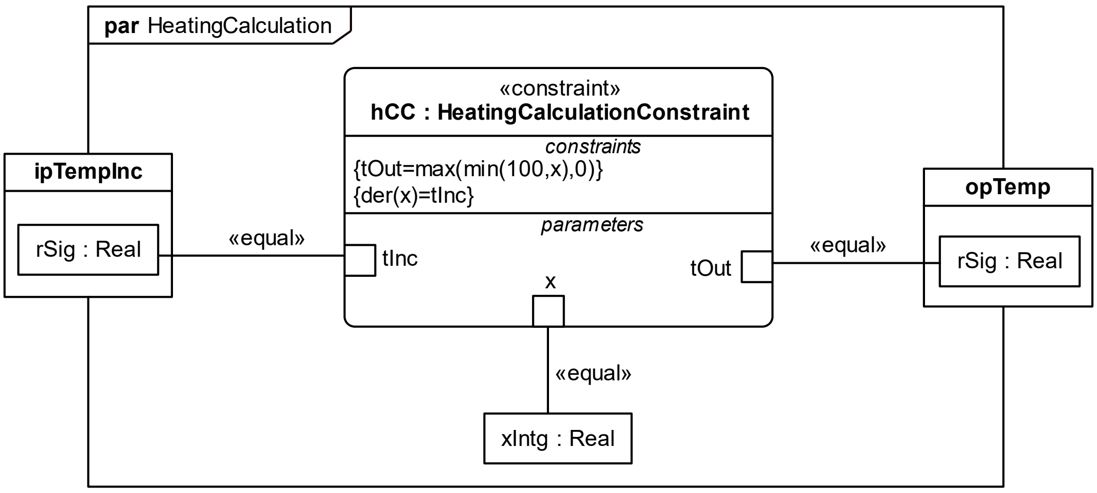


Figure 88: Parametric diagram applying the heating calculation constraint

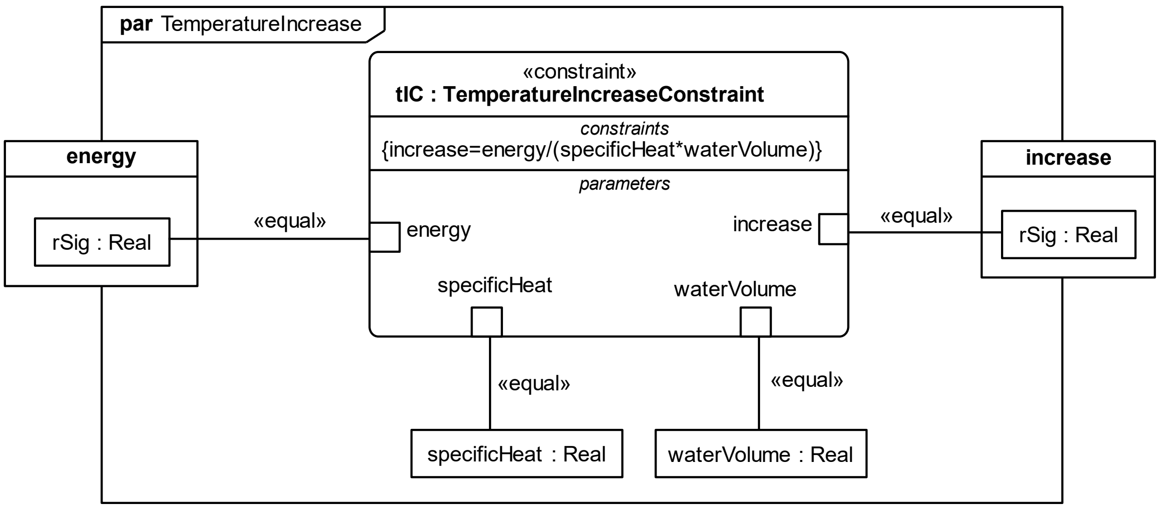


Figure 89: Parametric diagram applying temperature increase constraint

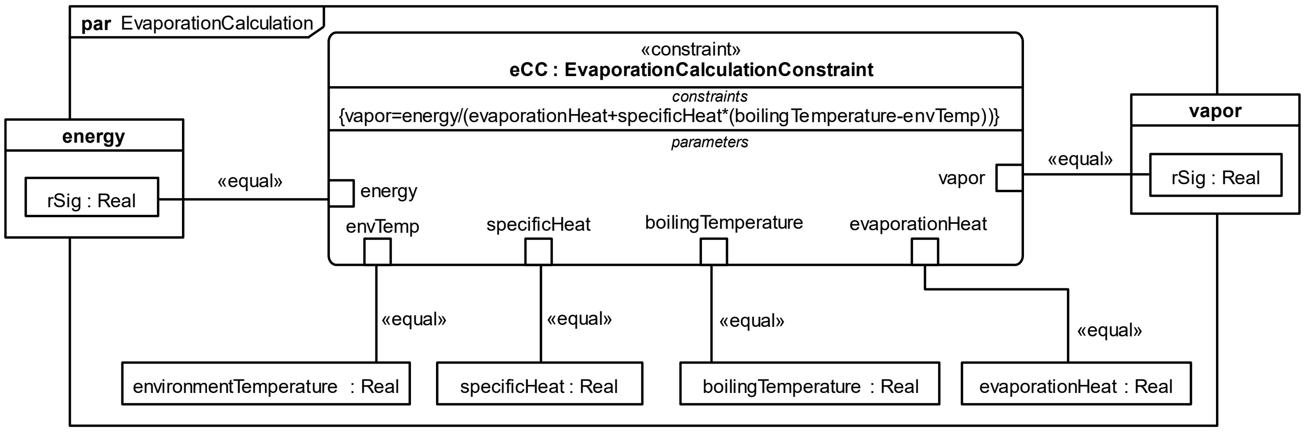


Figure 90: Parametric diagram applying the evaporation calculation constraint

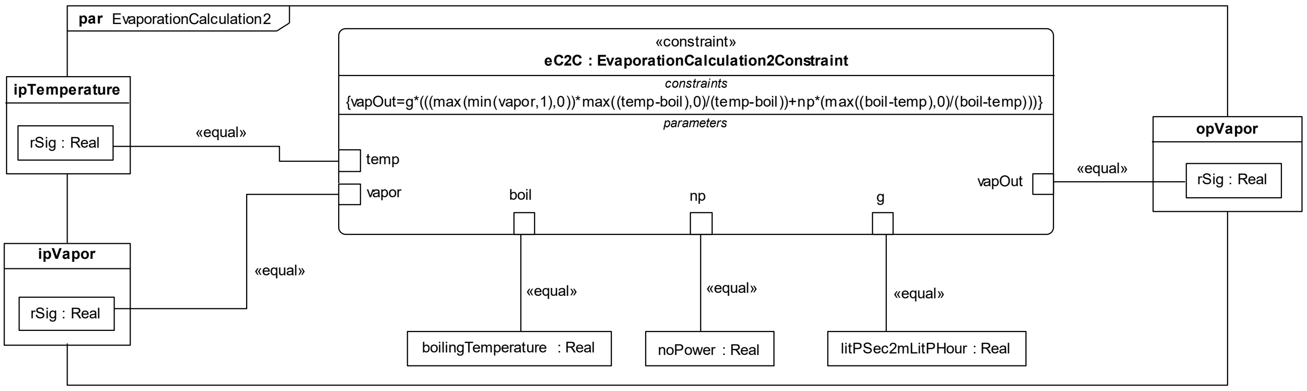

par EvaporationCalculation2
«constraint»
eC2C : EvaporationCalculation2Constraint
constraints
{vapOut=g*(((max(min(vapor,1),0))*max((temp-boil),0)/(temp-boil))+np*(max((boil-temp),0)/(boil-temp)))}
parameters
temp
vapor
boil
np
g
vapOut
ipTemperature
rSig : Real
«equal»
ipVapor
rSig : Real
«equal»
opVapor
rSig : Real
«equal»
«equal»
«equal»
«equal»
boilingTemperature : Real
noPower : Real
litPSec2mLitPHour : Real

Figure 91: Parametric diagram applying the second evaporation calculation constraint

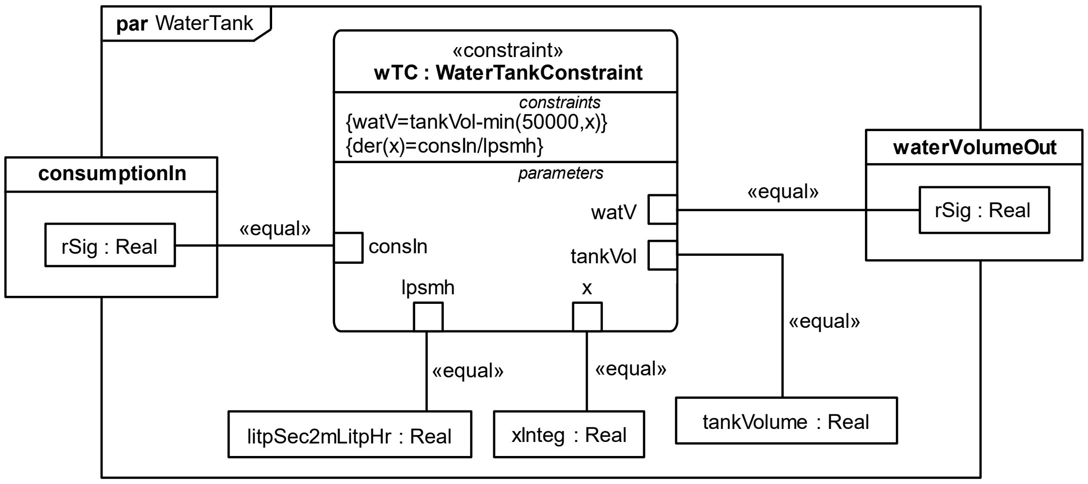


Figure 92: Parametric diagram applying the water tank constraint

A.5.8 State Machines

The state machine diagrams in this example specify how components react to changes by showing states of each component and the transition between these states. StateFlow only extends Simulink (see Subclause 10.12.4), which affects modeling of initial values (see Subannex A.5.9).

Figure 93 depicts the state machine of the block *HeaterControl*, the type of the *heatercontrol* property in the *Humidifier* internal block diagram (see Figure 68). The machine uses information from the block's ports to decide whether to operate the heater controller: the humidified room's current humidity from the input *humidityIn*, the target humidity from the property *targetHumidity*, and the control signal from the input *modeIn*. Its decision is sent to the vapor generation plant along the connection from the pin *heaterPowerRatioOut*.

Figure 94 depicts the state machine of the block *Control*, the type of the *control* property in the *Humidifier* internal structure (Figure 68). The machine determines the operation of the heater controller *heatercontrol* and the vapor generation plant *vaporgenerationplant* based on information received from the *Control* block's ports: a water volume signal *water Volum eIn* from the property *watertank,* a water temperature signal *waterTempIn* from *vaporgenerationplant*, and a switch decision signal *swtch* from *usage*.

Figure 95 depicts the state machine of the block *UsageScenario*, the type of the *usage* property in the *Humidifier* internal structure (Figure 68). The part property *usage* connects to the *control* part property with a signal from port *swtch* for the state machine *UsageScenario* to determine the time and duration for which the humidifier should humidify the room.

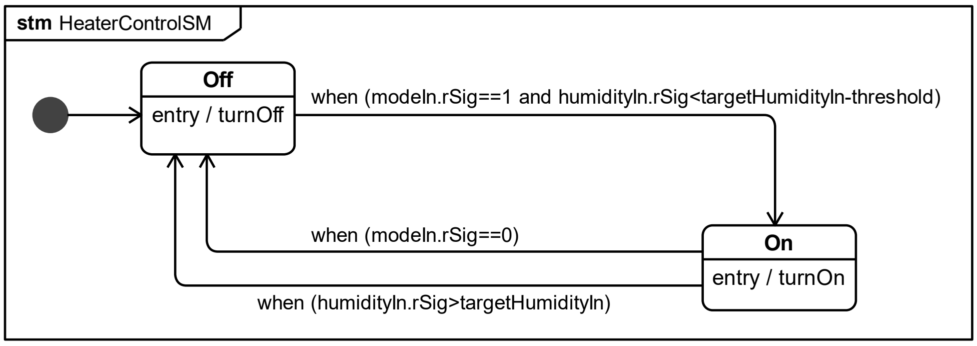

stm HeaterControlSM
Off
entry / turnOff
when (modeIn.rSig==1 and humidityIn.rSig<targetHumidityIn-threshold)
when (modeIn.rSig==0)
On
entry / turnOn
when (humidityIn.rSig>targetHumidityIn)

Figure 93: Heater Control State Machine Diagram

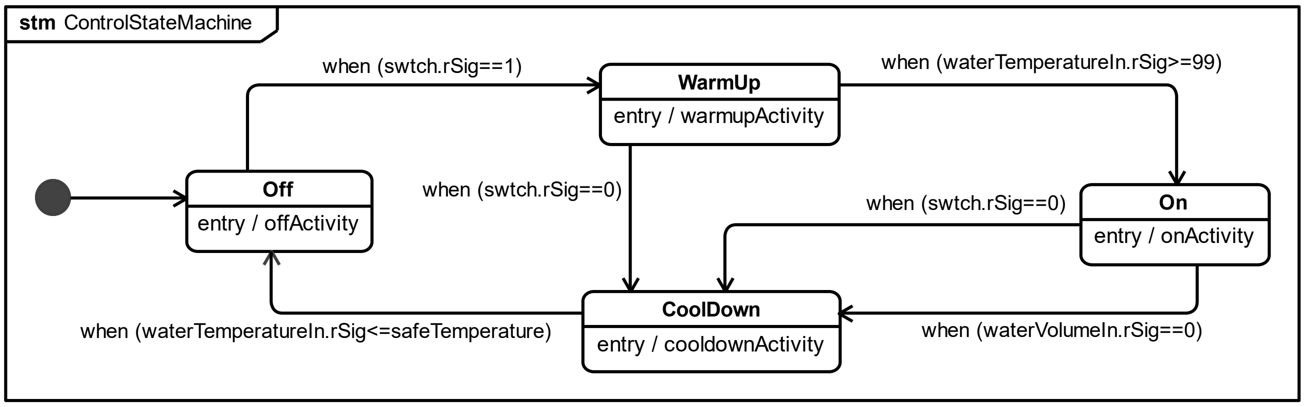


Figure 94: Humidifier Control State Machine Diagram

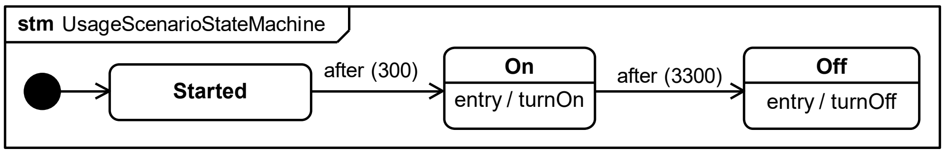


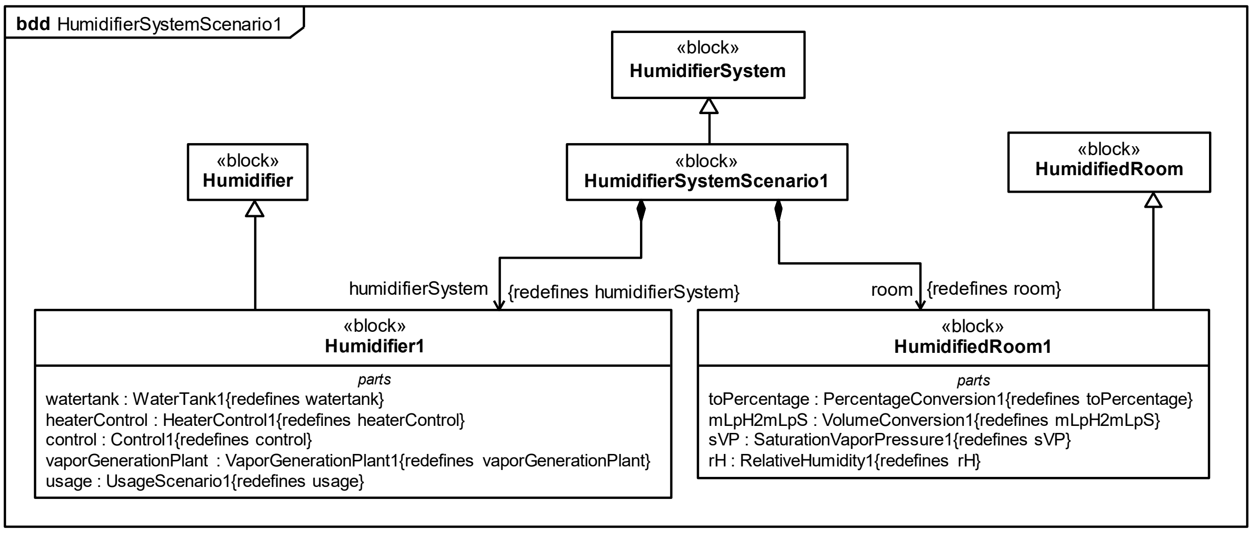
Figure 95: Humidifier Usage Scenario State Machine Diagram

A.5.9 Initial Values

Initial values are specified by block property redefinitions with default values in this example. This is necessary because StateFlow only extends Simulink (see Subclause 10.12.4), one of the desired platforms, and Simulink without Simscape does not have elements corresponding to SysML initial values on parts below the top level system (see

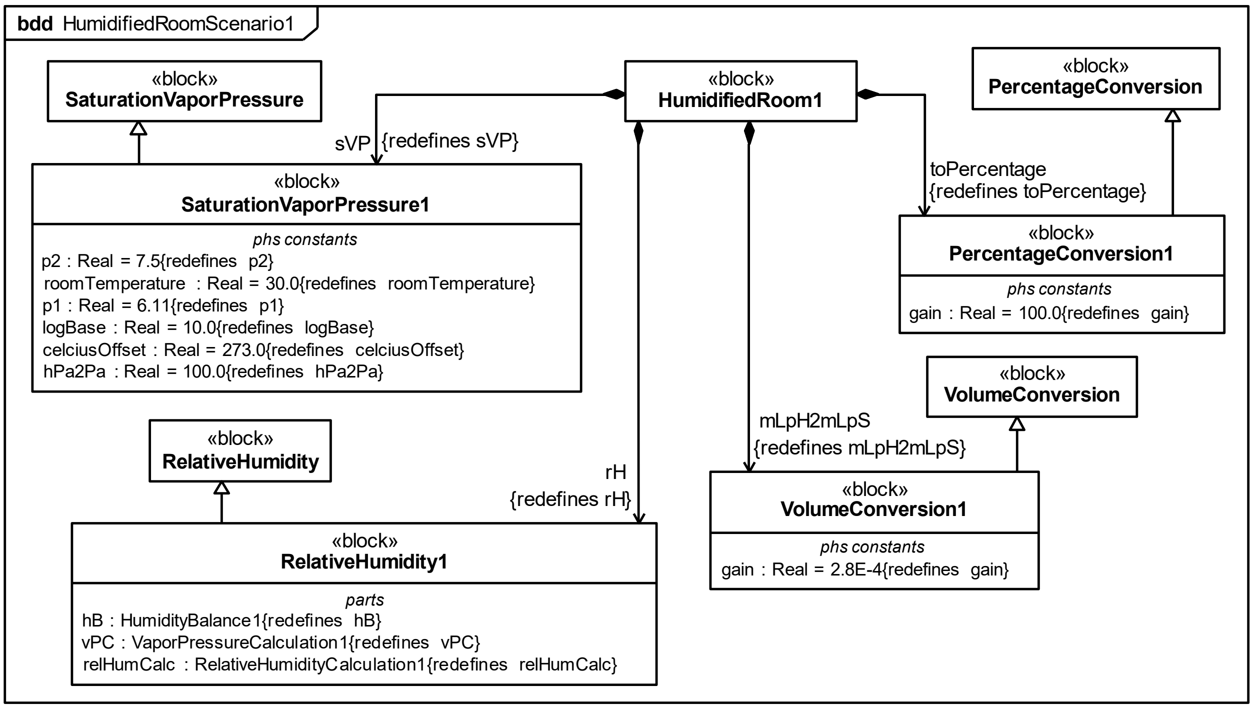
Subclause 10.10.4). SysML models must specialize component blocks to redefine properties and give default values, rather than use initial values, if they are to have corresponding elements in Simulink.

Each configuration (scenario) of values requires its own specializations and redefinitions, starting with a specialization the total system block. Blocks typing part properties of the specialized total system block (and any of their parts, recursively) are also specialized when they have values to be specified. The additional blocks in Figure 96 through Figure 102 are specialized from component blocks in Figure 72 through Figure 78, respectively (for parts of the total humidifier system, humidified room, relative humidity, humidifier, vapor generation plant, heating, and environment components). For example, Figure 96 shows *HumidifierSystemScenario1* specialized from the total system block. Specialized blocks have the name of their general components with the suffix *"-1 "*, indicating that this specialization is for the first scenario. Part property redefinitions with default values are indicated on each specialized block.



bdd HumidifierSystemScenario1
«block» HumidifierSystem
«block» Humidifier
«block» HumidifierSystemScenario1
«block» HumidifiedRoom
humidifierSystem
{redefines humidifierSystem}
room
{redefines room}
«block» Humidifier1
parts
watertank : WaterTank1{redefines watertank}
heaterControl : HeaterControl1{redefines heaterControl}
control : Control1{redefines control}
vaporGenerationPlant : VaporGenerationPlant1{redefines vaporGenerationPlant}
usage : UsageScenario1{redefines usage}
«block» HumidifiedRoom1
parts
toPercentage : PercentageConversion1{redefines toPercentage}
mLpH2mLpS : VolumeConversion1{redefines mLpH2mLpS}
sVP : SaturationVaporPressure1{redefines sVP}
rH : RelativeHumidity1{redefines rH}

Figure 96: Humidifier System Scenario Initial Values



bdd HumidifiedRoomScenario1
«block» SaturationVaporPressure
«block» HumidifiedRoom1
«block» PercentageConversion
sVP
{redefines sVP}
toPercentage
{redefines toPercentage}
«block» SaturationVaporPressure1
phs constants
p2 : Real = 7.5{redefines p2}
roomTemperature : Real = 30.0{redefines roomTemperature}
p1 : Real = 6.11{redefines p1}
logBase : Real = 10.0{redefines logBase}
celciusOffset : Real = 273.0{redefines celciusOffset}
hPa2Pa : Real = 100.0{redefines hPa2Pa}
«block» PercentageConversion1
phs constants
gain : Real = 100.0{redefines gain}
«block» VolumeConversion
«block» RelativeHumidity
mLpH2mLpS
{redefines mLpH2mLpS}
rH
{redefines rH}
«block» VolumeConversion1
phs constants
gain : Real = 2.8E-4{redefines gain}
«block» RelativeHumidity1
parts
hB : HumidityBalance1{redefines hB}
vPC : VaporPressureCalculation1{redefines vPC}
relHumCalc : RelativeHumidityCalculation1{redefines relHumCalc}

Figure 97: Humidified Room Scenario Initial Values

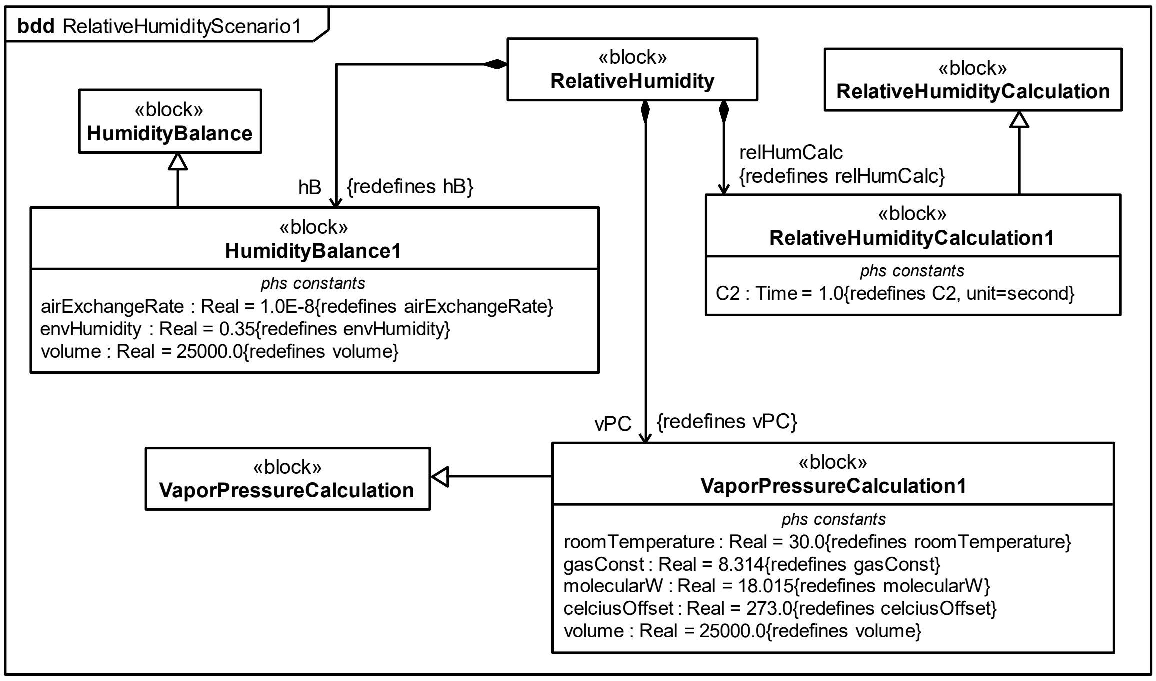

bdd RelativeHumidityScenario1
«block» RelativeHumidity
«block» HumidityBalance
«block» RelativeHumidityCalculation
hB {redefines hB}
relHumCalc {redefines relHumCalc}
«block» HumidityBalance1
phs constants
airExchangeRate : Real = 1.0E-8{redefines airExchangeRate}
envHumidity : Real = 0.35{redefines envHumidity}
volume : Real = 25000.0{redefines volume}
«block» RelativeHumidityCalculation1
phs constants
C2 : Time = 1.0{redefines C2, unit=second}
vPC {redefines vPC}
«block» VaporPressureCalculation
«block» VaporPressureCalculation1
phs constants
roomTemperature : Real = 30.0{redefines roomTemperature}
gasConst : Real = 8.314{redefines gasConst}
molecularW : Real = 18.015{redefines molecularW}
celciusOffset : Real = 273.0{redefines celciusOffset}
volume : Real = 25000.0{redefines volume}

Figure 98: Relative Humidity Scenario Initial Values

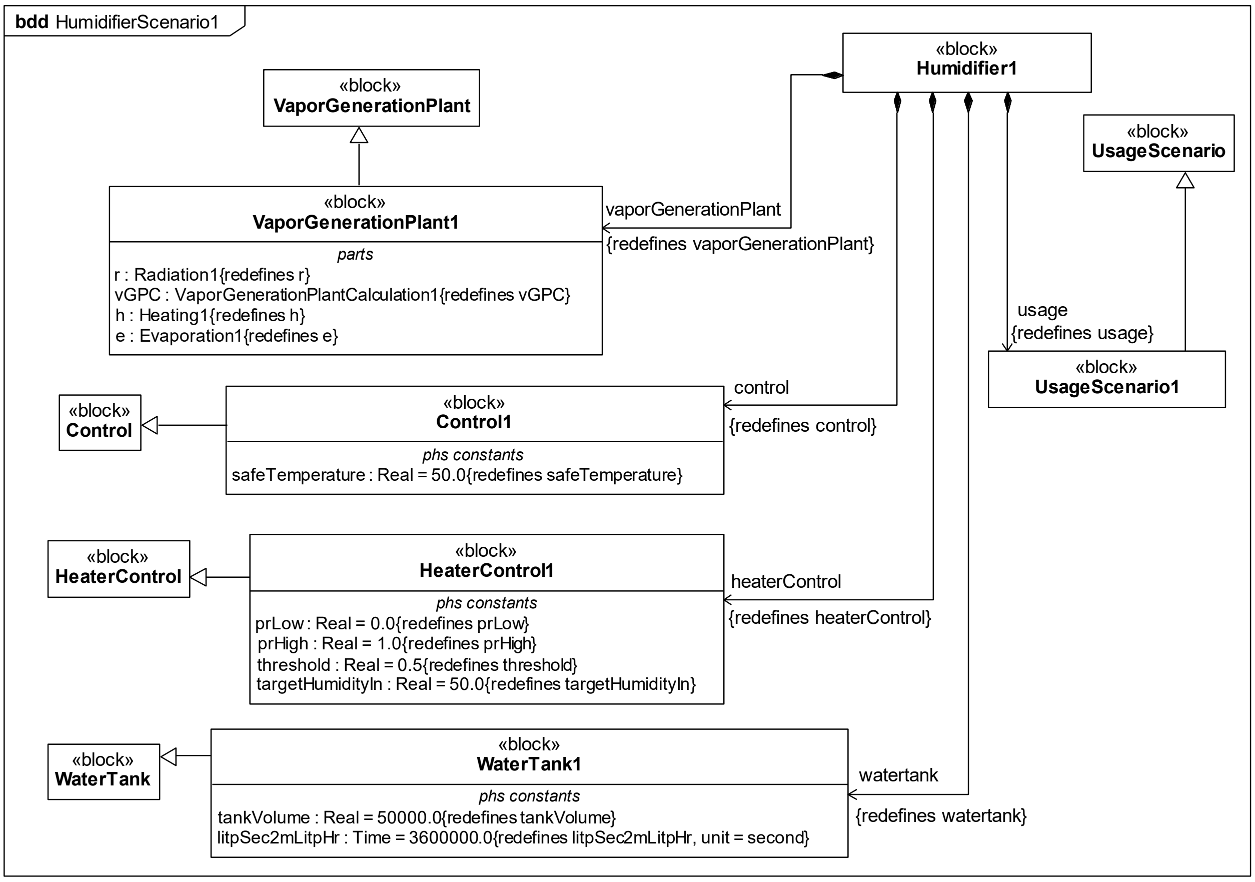

bdd HumidifierScenario1
«block» Humidifier1
«block» VaporGenerationPlant
«block» UsageScenario
«block» VaporGenerationPlant1
parts
r : Radiation1{redefines r}
vGPC : VaporGenerationPlantCalculation1{redefines vGPC}
h : Heating1{redefines h}
e : Evaporation1{redefines e}
vaporGenerationPlant
{redefines vaporGenerationPlant}
usage
{redefines usage}
«block» UsageScenario1
«block» Control
«block» Control1
phs constants
safeTemperature : Real = 50.0{redefines safeTemperature}
control
{redefines control}
«block» HeaterControl
«block» HeaterControl1
phs constants
prLow : Real = 0.0{redefines prLow}
prHigh : Real = 1.0{redefines prHigh}
threshold : Real = 0.5{redefines threshold}
targetHumidityIn : Real = 50.0{redefines targetHumidityIn}
heaterControl
{redefines heaterControl}
«block» WaterTank
«block» WaterTank1
phs constants
tankVolume : Real = 50000.0{redefines tankVolume}
litpSec2mLitpHr : Time = 3600000.0{redefines litpSec2mLitpHr, unit = second}
watertank
{redefines watertank}

Figure 99: Humidifier Scenario Initial Values

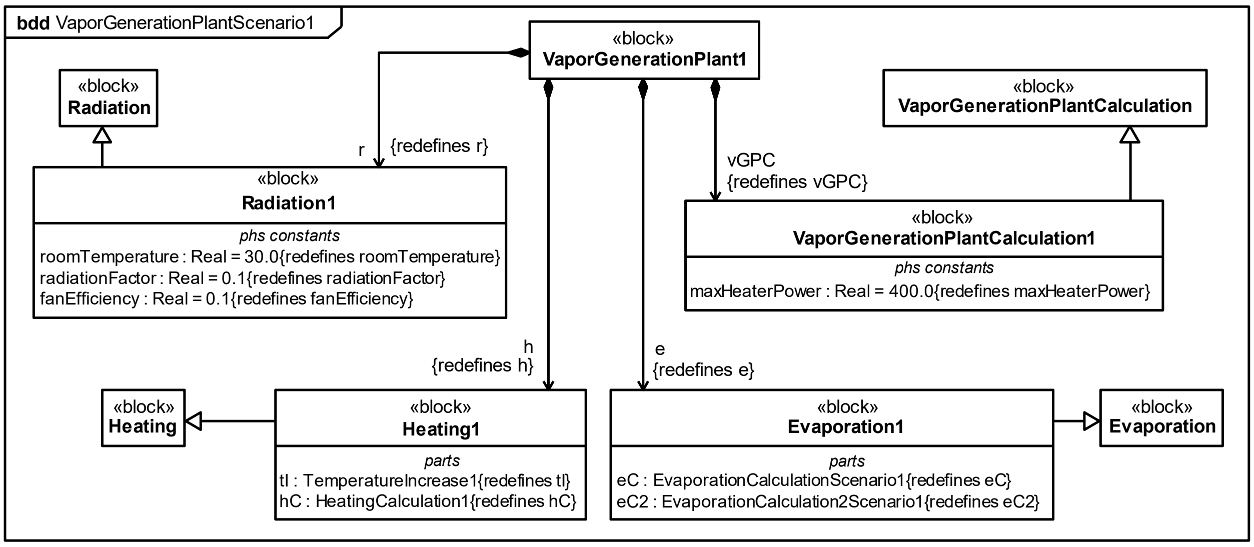

bdd VaporGenerationPlantScenario1
«block»
VaporGenerationPlant1
«block»
Radiation
r
{redefines r}
«block»
Radiation1
phs constants
roomTemperature : Real = 30.0{redefines roomTemperature}
radiationFactor : Real = 0.1{redefines radiationFactor}
fanEfficiency : Real = 0.1{redefines fanEfficiency}
«block»
VaporGenerationPlantCalculation
vGPC
{redefines vGPC}
«block»
VaporGenerationPlantCalculation1
phs constants
maxHeaterPower : Real = 400.0{redefines maxHeaterPower}
h
{redefines h}
e
{redefines e}
«block»
Heating
«block»
Heating1
parts
tI : TemperatureIncrease1{redefines tI}
hC : HeatingCalculation1{redefines hC}
«block»
Evaporation1
parts
eC : EvaporationCalculationScenario1{redefines eC}
eC2 : EvaporationCalculation2Scenario1{redefines eC2}
«block»
Evaporation

Figure 100: Vapor Generation Plant Scenario Initial Values

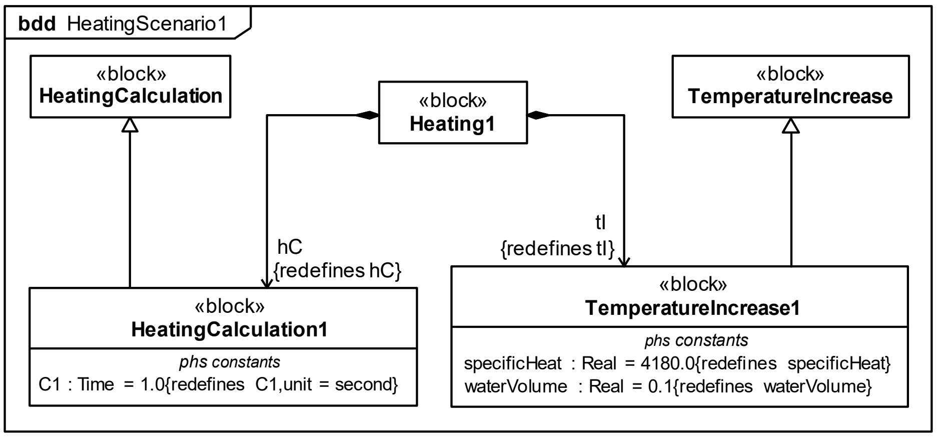


Figure 101: Heating Scenario Initial Values

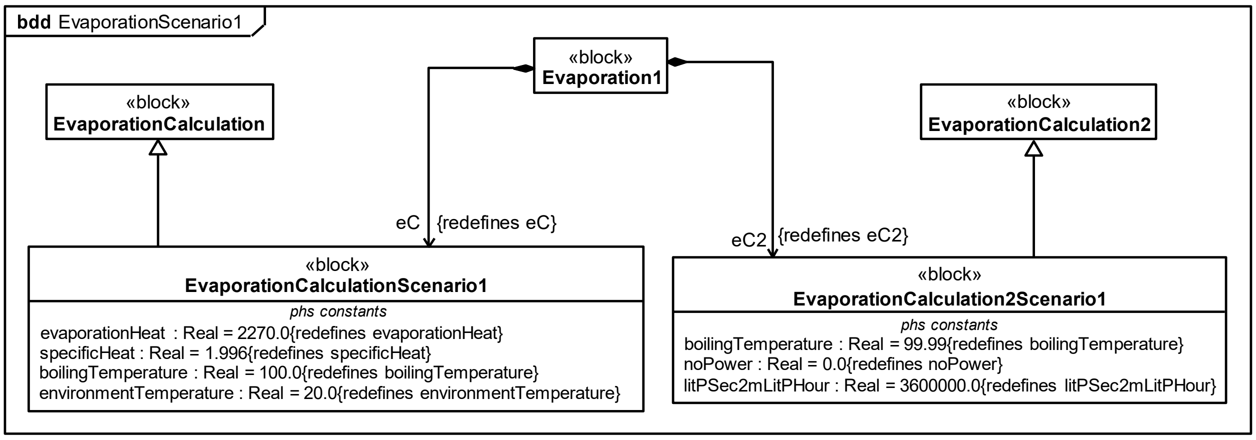

bdd EvaporationScenario1
«block»
Evaporation1
«block»
EvaporationCalculation
«block»
EvaporationCalculation2
eC
{redefines eC}
eC2
{redefines eC2}
«block»
EvaporationCalculationScenario1
phs constants
evaporationHeat : Real = 2270.0{redefines evaporationHeat}
specificHeat : Real = 1.996{redefines specificHeat}
boilingTemperature : Real = 100.0{redefines boilingTemperature}
environmentTemperature : Real = 20.0{redefines environmentTemperature}
«block»
EvaporationCalculation2Scenario1
phs constants
boilingTemperature : Real = 99.99{redefines boilingTemperature}
noPower : Real = 0.0{redefines noPower}
litPSec2mLitPHour : Real = 3600000.0{redefines litPSec2mLitPHour}

Figure 102: Evaporation Scenario Initial Values

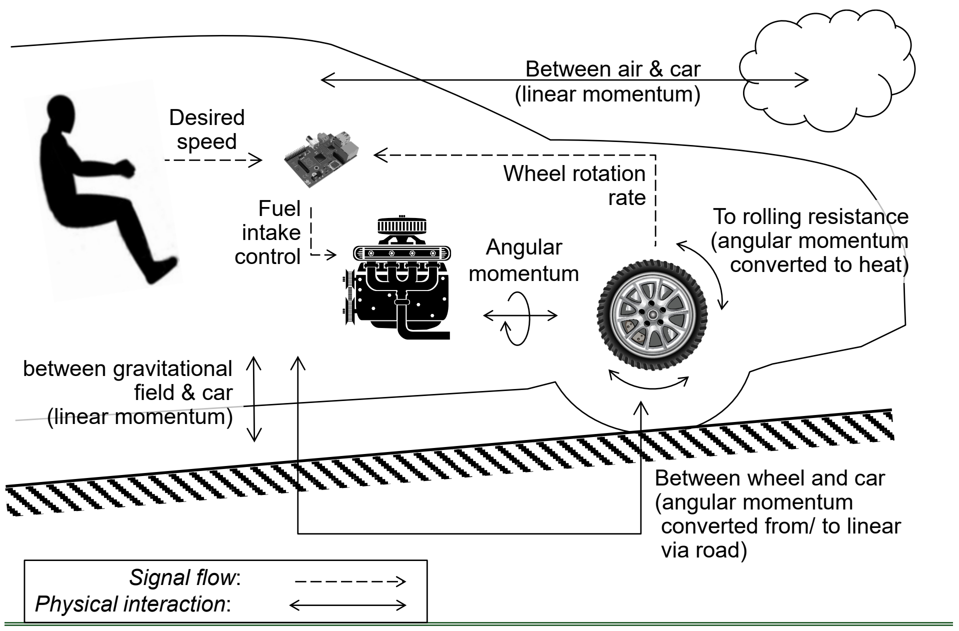
~~A1.6A1.1 Cruise Control System~~

~~A.7.0A.1.1 Introduction~~

~~This subannex gives a model of a cruise control system as an example that includes both physical interaction (linear and angular momentum) and signal flow (control and sensory signals).~~

~~A.9.0A.1.1 System Being Modeled~~

~~The automobile cruise control total system includes the vehicle, its operating environment, and the physical and informational processes involved, see Figure 103: Cruise control system example (physical interactions are shown with solid, bidirectional arrows between system components, and signal flows with dashed, unidirectional arrows).~~



~~**Figure 103: Cruise control system example**~~

~~A.13.0A.1.1 Internal Structure~~

~~Figure 104 shows the internal structure of a *CruiseControlTotalSystem* block. Part properties, typed by blocks defined in Subannex A.6.4, represent components of the system. They are connected to each other directly or through ports, representing physical interaction or signal flow between them. Item flows on connectors indicate the type of signal or conserved physical characteristic that passes along them. Signals control production of angular momentum by the engine. The cruise controller (*speedController*) receives speed signals from the driver (*driver*) and the wheels (*impeller*), where the former is the goal speed and the latter is the current speed. The cruise controller determines speed adjustments by sending the engine (*powerSource*) a signal containing the amount of fuel needed to inject into the engine. Angular momentum typically flows out from the engine to the wheels and is transformed to linear momentum back into the car through interaction with the road. This appears in Figure 105 as a connector between wheel and automobile supported by an association block specifying the transformation, as well as another connector between the wheel and road to depict the contact between the two. The car's linear momentum is also affected by gravitation (*gravVehicleLink*) and surrounding air (*atmosphere*), appearing in Figure 104 as additional connectors between the car and these components.~~

~~SysML initial values specify property values for components used in internal block diagrams. Figure 104 shows initial values and units for each of the system components (properties defined in Subannex A.6.4). The car gives its cross-sectional area, drag coefficient, and mass. The driver specifies values for decisions about the car's speed. The cruise controller gives its proportional-integrator and throttle-acceleration coefficients that determine the amount of fuel injected into the engine. The engine specifies its torque coefficient, related to the gears and crankshaft of the car. The wheel has a radius and a coefficient for dissipation of angular momentum into heat due to rolling resistance. Earth specifies its gravity and density of its atmosphere. The road gives values determining its slope. Initial values are specified directly on *CruiseControlTotalSystem* for brevity, but could be on specializations instead, defining multiple test cases without modifying the original system model. An alternative to initial values is to use default values on blocks typing system properties (see Subannex A.5.9).~~

A1.6 Cruise Control System

A.6.1 Introduction

This subannex gives a model of a cruise control system as an example that includes both physical interaction (linear and angular momentum) and signal flow (control and sensory signals).

A.6.2 System Being Modeled

The automobile cruise control total system includes the vehicle, its operating environment, and the physical and informational processes involved, see Figure 103: Cruise control system example (physical interactions are shown with solid, bidirectional arrows between system components, and signal flows with dashed, unidirectional arrows).

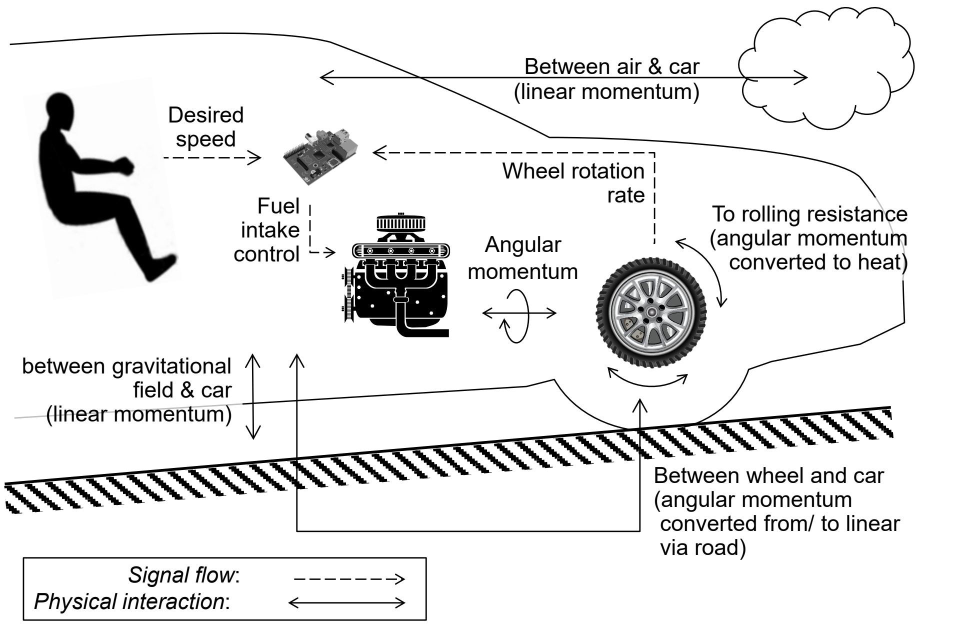
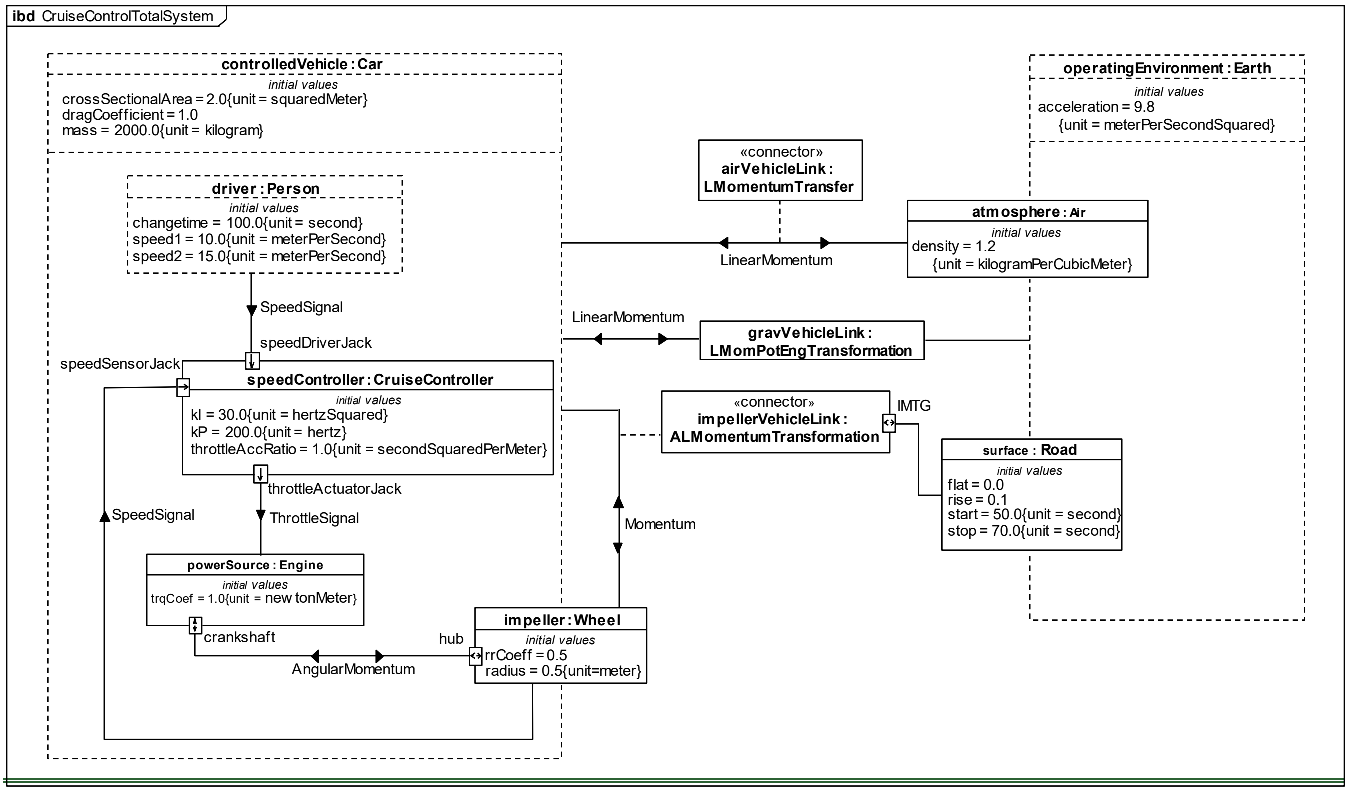


Figure 103: Cruise control system example

A.6.3 Internal Structure

Figure 104 shows the internal structure of a *CruiseControlTotalSystem* block. Part properties, typed by blocks defined in Subannex A.6.4, represent components of the system. They are connected to each other directly or through ports, representing physical interaction or signal flow between them. Item flows on connectors indicate the type of signal or conserved physical characteristic that passes along them. Signals control production of angular momentum by the engine. The cruise controller (*speedController)* receives speed signals from the driver (*driver*) and the wheels (*impeller*), where the former is the goal speed and the latter is the current speed. The cruise controller determines speed adjustments by sending the engine (*powerSource*) a signal containing the amount of fuel needed to inject into the engine. Angular momentum typically flows out from the engine to the wheels and is transformed to linear momentum back into the car through interaction with the road. This appears in Figure 105 as a connector between wheel and automobile supported by an association block specifying the transformation, as well as another connector between the wheel and road to depict the contact between the two. The car's linear momentum is also affected by gravitation (*gravVehicleLink*) and surrounding air (*atmosphere*), appearing in Figure 104 as additional connectors between the car and these components.

SysML initial values specify property values for components used in internal block diagrams. Figure 104 shows initial values and units for each of the system components (properties defined in Subannex A.6.4). The car gives its cross-sectional area, drag coefficient, and mass. The driver specifies values for decisions about the car's speed. The cruise controller gives its proportional-integrator and throttle-acceleration coefficients that determine the amount of fuel injected into the engine. The engine specifies its torque coefficient, related to the gears and crankshaft of the car. The wheel has a radius and a coefficient for dissipation of angular momentum into heat due to rolling resistance. Earth specifies its gravity and density of its atmosphere. The road gives values determining its slope. Initial values are specified directly on *CruiseControlTotalSystem* for brevity, but could be on specializations instead, defining multiple test cases without modifying the original system model. An alternative to initial values is to use default values on blocks typing system properties (see Subannex A.5.9).


ibd CruiseControlTotalSystem
controlledVehicle : Car
initial values
crossSectionalArea = 2.0{unit = squaredMeter}
dragCoefficient = 1.0
mass = 2000.0{unit = kilogram}
driver : Person
initial values
changetime = 100.0{unit = second}
speed1 = 10.0{unit = meterPerSecond}
speed2 = 15.0{unit = meterPerSecond}
SpeedSignal
speedDriverJack
speedSensorJack
speedController : CruiseController
initial values
kI = 30.0{unit = hertzSquared}
kP = 200.0{unit = hertz}
throttleAccRatio = 1.0{unit = secondSquaredPerMeter}
throttleActuatorJack
SpeedSignal
ThrottleSignal
powerSource : Engine
initial values
trqCoef = 1.0{unit = new tonMeter}
crankshaft
hub
AngularMomentum
impeller : Wheel
initial values
rrCoeff = 0.5
radius = 0.5{unit=meter}
«connector»
airVehicleLink : LMomentumTransfer
LinearMomentum
LinearMomentum
gravVehicleLink : LMomPotEngTransformation
«connector»
impellerVehicleLink : ALMomentumTransformation
IMTG
Momentum
operatingEnvironment : Earth
initial values
acceleration = 9.8
{unit = meterPerSecondSquared}
atmosphere : Air
initial values
density = 1.2
{unit = kilogramPerCubicMeter}
surface : Road
initial values
flat = 0.0
rise = 0.1
start = 50.0{unit = second}
stop = 70.0{unit = second}

~~**Figure 104: Internal structure of the cruise control system**~~

~~A.15.1A.1.1 Blocks and Ports~~

~~Figure 105: Total system blocks, ports, & component properties shows block definitions for components of *CruiseControlTotalSystem* in Figure 104. Figure 106 and Figure 107 show more detailed definitions about physical interactions between the car and the surrounding air and gravity, while Figure 109 and Figure 108 show these between the wheels and car and engine. Figure 110 shows definitions for signal flows in the car. Many components have their own behaviors, defined as constraints in Subannex A.6.6.~~

~~Components involved in the interaction between the car and surrounding air are defined in Figure 106 (the car and Earth's port typed by *Air*). They are generalized by *LMomFlowElement* from the physical interaction library (see Subclause 11.2.2) and linked by an association that is also a block, *LMomentum Transfer*, indicated by a dashed line (the association end on the library side is owned by the association, to avoid modifying the library element). The association block represents linear momentum transfer between the vehicle and the air around it. The internal structure of *LMomentumTransfer* is defined in Subannex A.6.7 (see Subclause 9.2.2 about association blocks).~~

~~Components involved in the interaction between the car and Earth's gravity are defined in Figure 107 (the car and its potential energy in Earth's gravitational field, *LMomPotEngTransformation*). They are generalized by *LMomFlowElement* from the physical interaction library, and linked by an association. The transformation between linear momentum and potential energy is not modeled with an association between the car and Earth to highlight that momentum converted to potential energy can only be transferred back to the car, as compared to momentum transferred to the air, which can be transferred to other objects. The connector to Earth reflects its involvement in converting between linear momentum and potential energy, even though the Earth is too large to accept or provide momentum. The connector also provides access to properties needed by interaction equations, such as gravitation of the Earth and slope of the road, see Subannexes A.6.6 and A.6.7. The internal structure of *LMomPotEngTransformation* is defined in Subannex A.6.7.~~

~~Components involved in the transformation between angular momentum of the wheels and linear momentum of the car are shown in Figure 108 (the car, road, and wheel). The car is generalized by LMomFlowElement as before, while the wheel is generalized by interface block *AMomFlowComponent*, which in turn is generalized by *AMomFlowElement*, from the physical interaction library (see Subclause 11.2.2). The library's *LMomFlowElement* and *AMomFlowComponent* are linked by an association that is also a block *ALMomentum Transformation*, indicated by a dashed line (the association ends are owned by the association, to avoid modifying the library elements). The association block represents transformation between the wheels' angular momentum and the car's linear momentum. It has a port *lMTG* typed by a block *LMomentumGround* (generalized by *LMomFlowElement*), for connecting to physical objects that are too large to accept or provide linear momentum, such as the road (generalized by *LMomentumGround*).~~

~~This~~

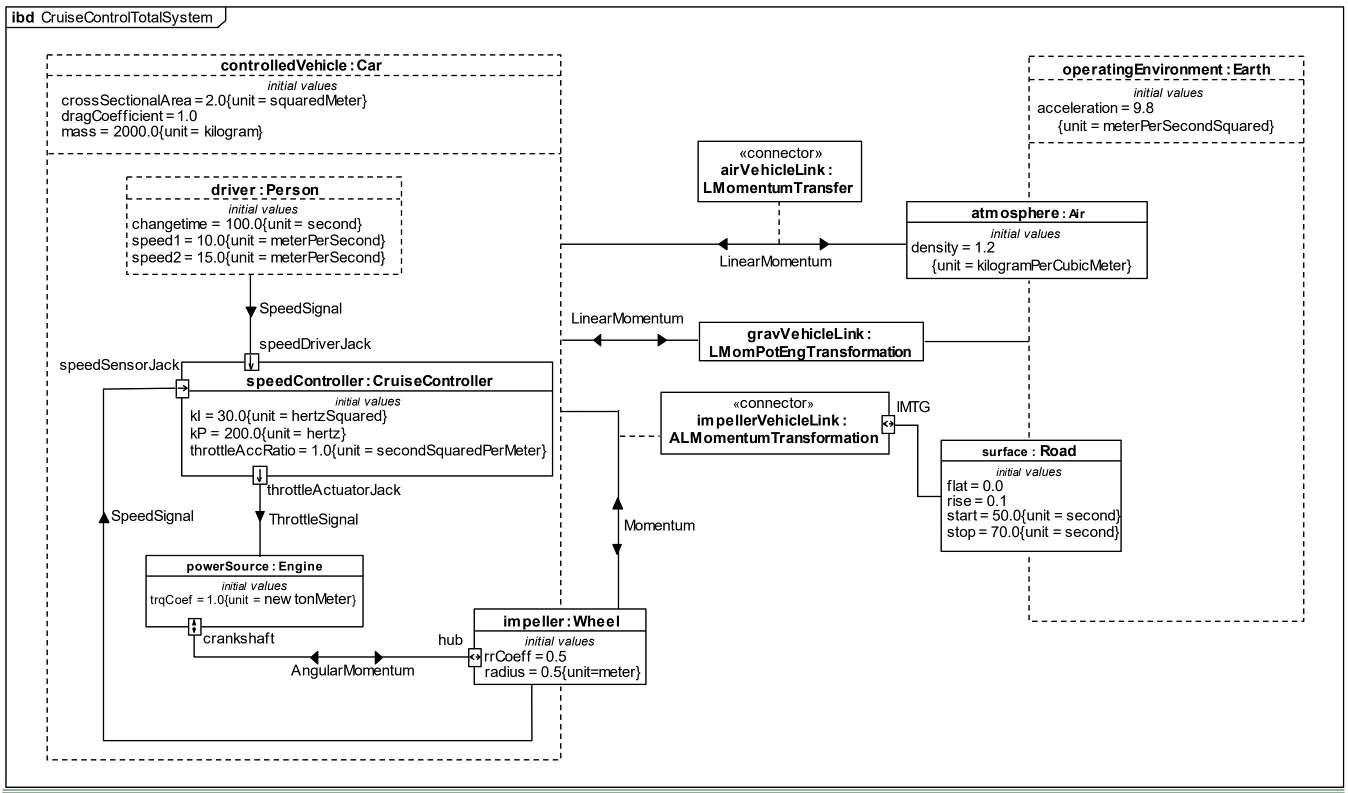


Figure 104: Internal structure of the cruise control system

A.6.4 Blocks and Ports

Figure 105: Total system blocks, ports, & component properties shows block definitions for components of *CruiseControlTotalSystem* in Figure 104. Figure 106 and Figure 107 show more detailed definitions about physical interactions between the car and the surrounding air and gravity, while Figure 109 and Figure 108 show these between the wheels and car and engine. Figure 110 shows definitions for signal flows in the car. Many components have their own behaviors, defined as constraints in Subannex A.6.6.

Components involved in the interaction between the car and surrounding air are defined in Figure 106 (the car and Earth's port typed by *Air*). They are generalized by *LMomFlowElement* from the physical interaction library (see Subclause 11.2.2) and linked by an association that is also a block, *LMomentum Transfer*, indicated by a dashed line (the association end on the library side is owned by the association, to avoid modifying the library element). The association block represents linear momentum transfer between the vehicle and the air around it. The internal structure of *LMomentumTransfer* is defined in Subannex A.6.7 (see Subclause 9.2.2 about association blocks).

Components involved in the interaction between the car and Earth's gravity are defined in Figure 107 (the car and its potential energy in Earth's gravitational field, *LMomPotEngTransformation*). They are generalized by *LMomFlowElement* from the physical interaction library, and linked by an association. The transformation between linear momentum and potential energy is not modeled with an association between the car and Earth to highlight that momentum converted to potential energy can only be transferred back to the car, as compared to momentum transferred to the air, which can be transferred to other objects. The connector to Earth reflects its involvement in converting between linear momentum and potential energy, even though the Earth is too large to accept or provide momentum. The connector also provides access to properties needed by interaction equations, such as gravitation of the Earth and slope of the road, see Subannexes A.6.6 and A.6.7. The internal structure of *LMomPotEngTransformation* is defined in Subannex A.6.7.

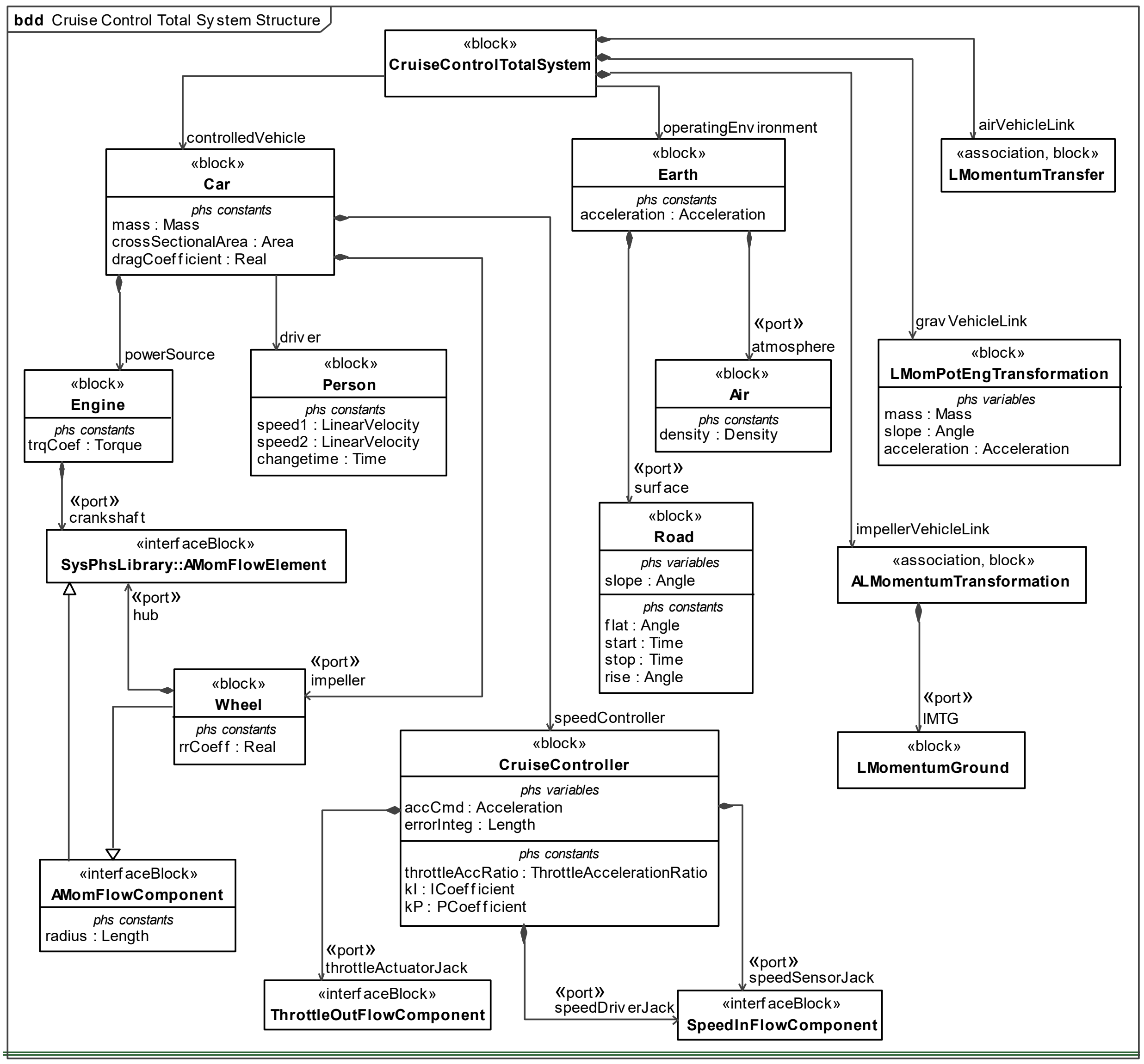
Components involved in the transformation between angular momentum of the wheels and linear momentum of the car are shown in Figure 108 (the car, road, and wheel). The car is generalized by LMomFlowElement as before, while the wheel is generalized by interface block *AMomFlowComponent*, which in turn is generalized by *AMomFlowElement*, from the physical interaction library (see Subclause 11.2.2). The library's *LMomFlowElement* and *AMomFlowComponent* are linked by an association that is also a block *ALMomentum Transformation*, indicated by a dashed line (the association ends are owned by the association, to avoid modifying the library elements). The association block represents transformation between the wheels' angular momentum and the car's linear momentum. It has a port *lMTG* typed by a block *LMomentumGround* (generalized by *LMomFlowElement*), for connecting to physical objects that are too large to accept or provide linear momentum, such as the road (generalized by *LMomentumGround*). This

~~connection appears in Figure 104, representing the road's involvement in the transformation between angular and linear momentum. The internal structure of *AMomFlowComponent* is defined in Subannex A.6.7.~~

~~Components involved in transferring angular momentum between the car's internal components are depicted in Figure 109 (the engine and the wheel, via their crankshaft and hub ports, respectively). The crankshaft and hub ports are typed by *AMomFlowElement* from the physical interaction library (crankshafts and hubs are modeled as interface blocks for brevity).~~

~~The library blocks *AMomFlowElement* and *LMomFlowElement* have flow properties *aMomF* and *lMomF*, respectively. They are typed by blocks *FlowingAMom* and *FlowingLMom* (also from the library) representing flow of conserved physical characteristics. These give flow rate and potential variables (*trq, aV* and *f, lV*). Models use the variables directly on library blocks or on specialized blocks that inherit them.~~

~~Components sending and receiving signals in the vehicle are shown in Figure 110 (the driver, wheels, engine, and the cruise controller via its ports). Two cruise controller ports receive signals giving the driver's desired speed and the vehicle's current speed, while a third sends signals to the engine setting the fuel injection rate. The speed ports on the cruise controller are typed by the interface block *SpeedInFlowComponent* to receive signals from the driver and wheels, which send them by specializing *SpeedOutFlowComponent*. The throttle actuator port on the cruise controller is typed by the interface block *ThrottleOutFlowComponent* to send fuel injection signals to the engine, which receives them by specializing *ThrottleInFlowComponent*.~~



~~**Figure 105: Total system blocks, ports, & component properties**~~ connection appears in Figure 104, representing the road's involvement in the transformation between angular and linear momentum. The internal structure of *AMomFlowComponent* is defined in Subannex A.6.7.

Components involved in transferring angular momentum between the car's internal components are depicted in Figure 109 (the engine and the wheel, via their crankshaft and hub ports, respectively). The crankshaft and hub ports are typed by *AMomFlowElement* from the physical interaction library (crankshafts and hubs are modeled as interface blocks for brevity).

The library blocks *AMomFlowElement* and *LMomFlowElement* have flow properties *aMomF* and *lMomF*, respectively. They are typed by blocks *FlowingAMom* and *FlowingLMom* (also from the library) representing flow of conserved physical characteristics. These give flow rate and potential variables (*trq, aV* and *f, lV*). Models use the variables directly on library blocks or on specialized blocks that inherit them.

Components sending and receiving signals in the vehicle are shown in Figure 110 (the driver, wheels, engine, and the cruise controller via its ports). Two cruise controller ports receive signals giving the driver's desired speed and the vehicle's current speed, while a third sends signals to the engine setting the fuel injection rate. The speed ports on the cruise controller are typed by the interface block *SpeedInFlowComponent* to receive signals from the driver and wheels, which send them by specializing *SpeedOutFlowComponent*. The throttle actuator port on the cruise controller is typed by the interface block *ThrottleOutFlowComponent* to send fuel injection signals to the engine, which receives them by specializing *ThrottleInFlowComponent*.

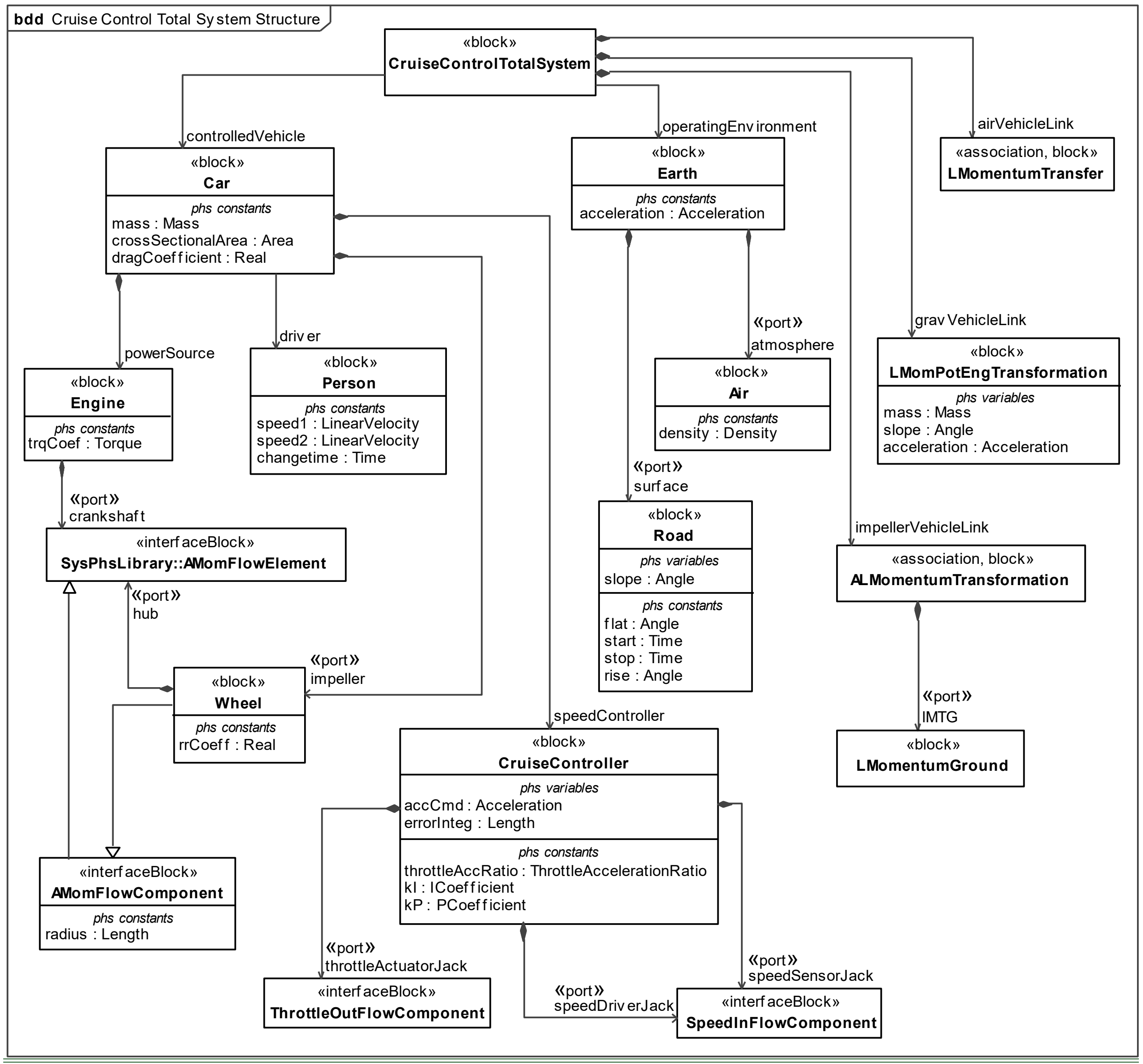
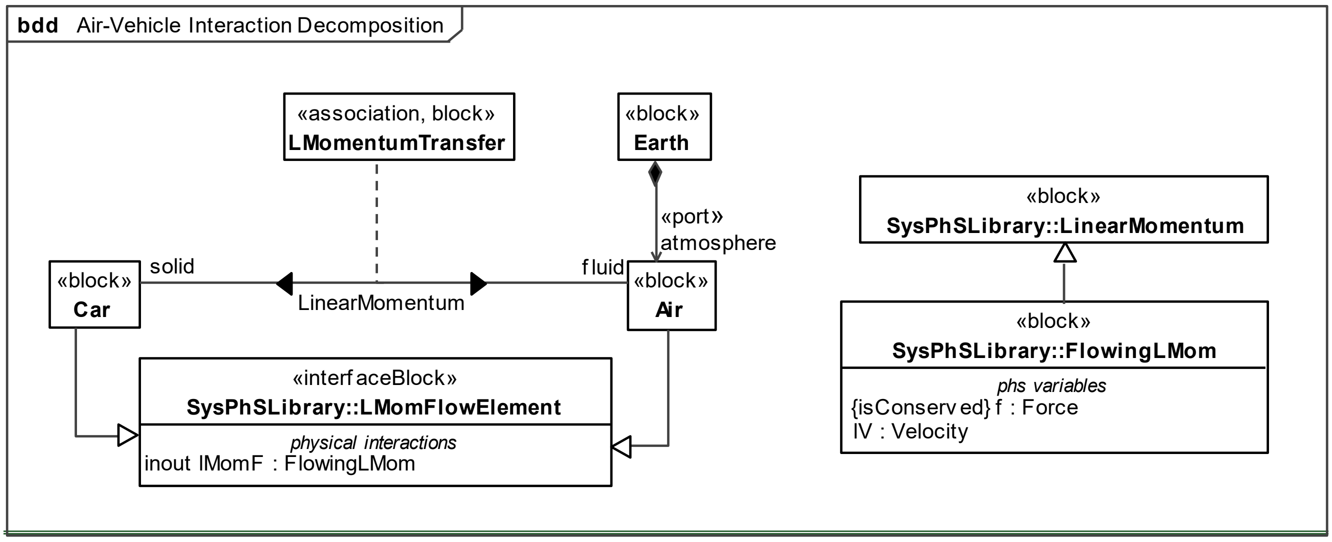
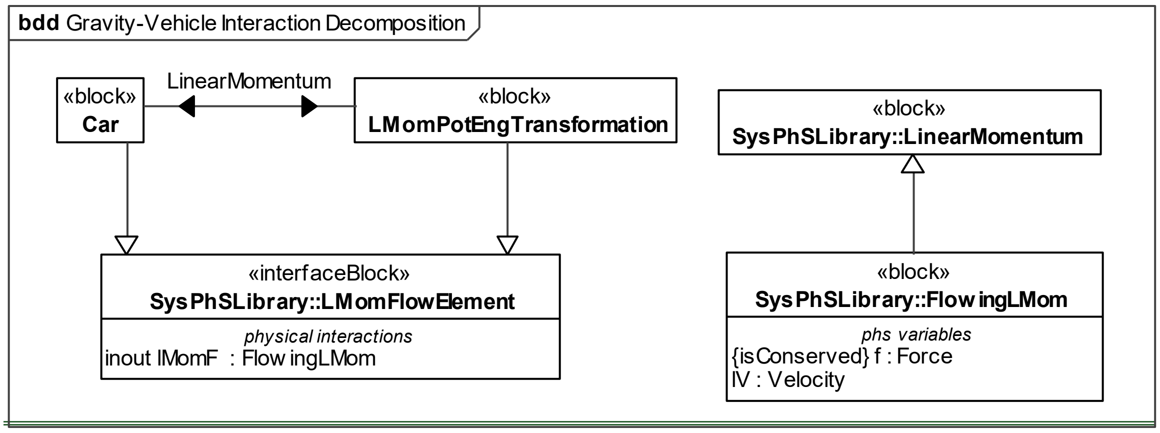


Figure 105: Total system blocks, ports, & component properties



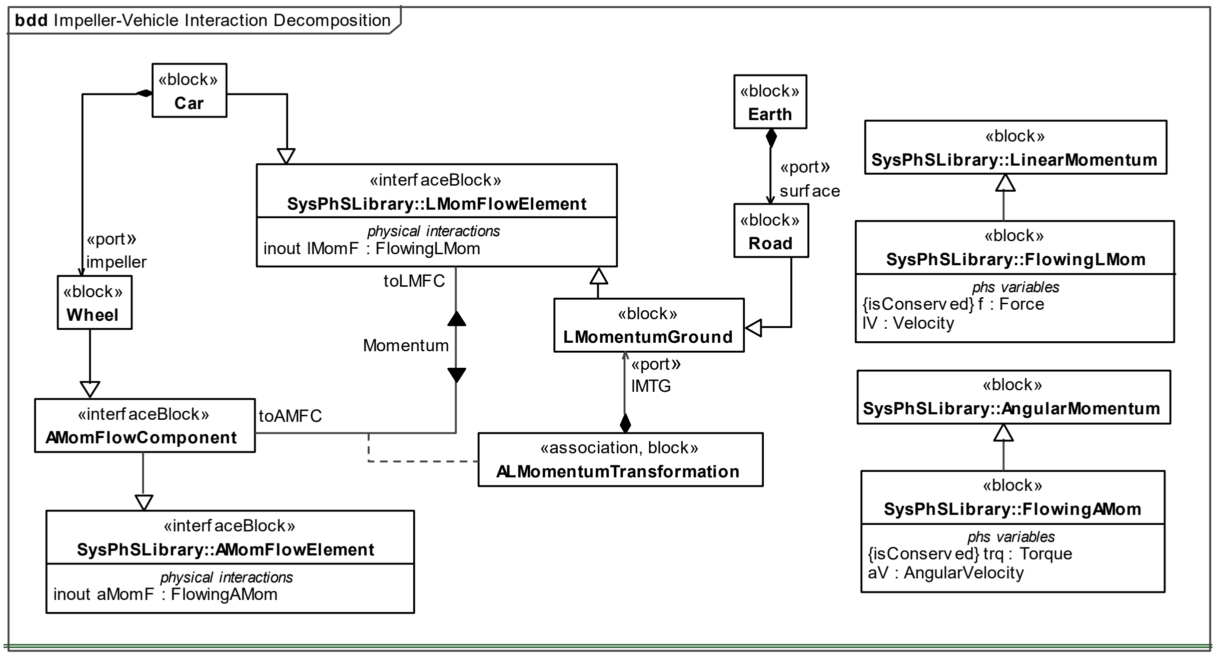
bdd Air-Vehicle Interaction Decomposition
«association, block» LMomentumTransfer
«block» Earth
«port» atmosphere
«block» SysPhSLibrary::LinearMomentum
«block» Car
solid
LinearMomentum
fluid
«block» Air
«interfaceBlock» SysPhSLibrary::LMomFlowElement
physical interactions
inout lMomF : FlowingLMom
«block» SysPhSLibrary::FlowingLMom
phs variables
{isConserved} f : Force
lV : Velocity

~~Figure 106: Air-vehicle interaction blocks, ports, & component properties~~



bdd Gravity-Vehicle Interaction Decomposition
«block» Car
LinearMomentum
«block» LMomPotEngTransformation
«block» SysPhSLibrary::LinearMomentum
«interfaceBlock» SysPhSLibrary::LMomFlowElement
physical interactions
inout lMomF : FlowingLMom
«block» SysPhSLibrary::FlowingLMom
phs variables
{isConserved} f : Force
lV : Velocity

~~Figure 107: Gravity-vehicle interaction blocks, ports, & component properties~~



bdd Impeller-Vehicle Interaction Decomposition
«block» Car
«block» Earth
«port» surface
«block» SysPhSLibrary::LinearMomentum
«interfaceBlock» SysPhSLibrary::LMomFlowElement
physical interactions
inout lMomF : FlowingLMom
«port» impeller
«block» Road
«block» SysPhSLibrary::FlowingLMom
phs variables
{isConserved} f : Force
lV : Velocity
«block» Wheel
toLMFC
Momentum
«block» LMomentumGround
«port» lMTG
«interfaceBlock» AMomFlowComponent
toAMFC
«association, block» ALMomentumTransformation
«block» SysPhSLibrary::AngularMomentum
«block» SysPhSLibrary::FlowingAMom
phs variables
{isConserved} trq : Torque
aV : AngularVelocity
«interfaceBlock» SysPhSLibrary::AMomFlowElement
physical interactions
inout aMomF : FlowingAMom

~~Figure 108: Impeller-vehicle interaction blocks, ports, & component properties~~

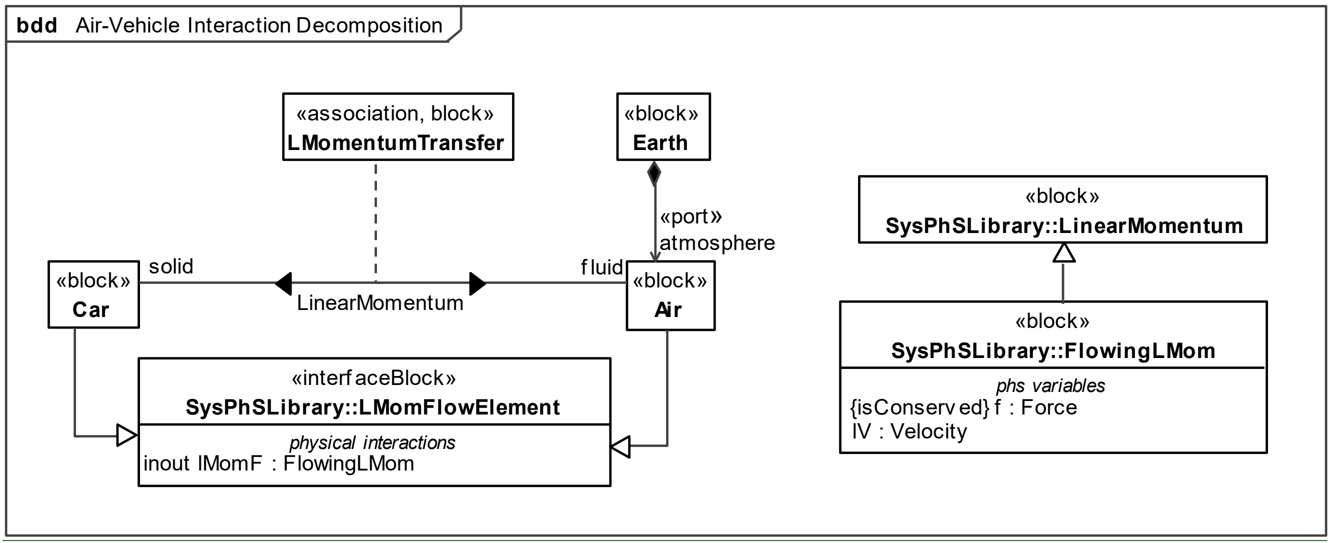

bdd Air-Vehicle Interaction Decomposition
«association, block» LMomentumTransfer
«block» Earth
«port» atmosphere
«block» Car
solid
LinearMomentum
fluid
«block» Air
«interfaceBlock» SysPhSLibrary::LMomFlowElement
physical interactions
inout lMomF : FlowingLMom
«block» SysPhSLibrary::LinearMomentum
«block» SysPhSLibrary::FlowingLMom
phs variables
{isConserved} f : Force
lV : Velocity

Figure 106: Air-vehicle interaction blocks, ports, & component properties

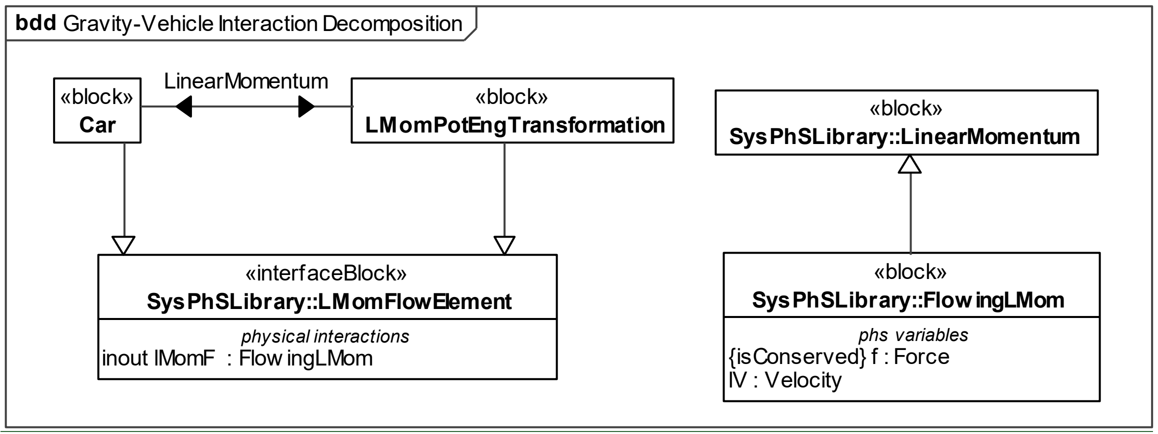

bdd Gravity-Vehicle Interaction Decomposition
«block» Car
LinearMomentum
«block» LMomPotEngTransformation
«block» SysPhSLibrary::LinearMomentum
«interfaceBlock» SysPhSLibrary::LMomFlowElement
physical interactions
inout lMomF : FlowingLMom
«block» SysPhSLibrary::FlowingLMom
phs variables
{isConserved} f : Force
lV : Velocity

Figure 107: Gravity-vehicle interaction blocks, ports, & component properties

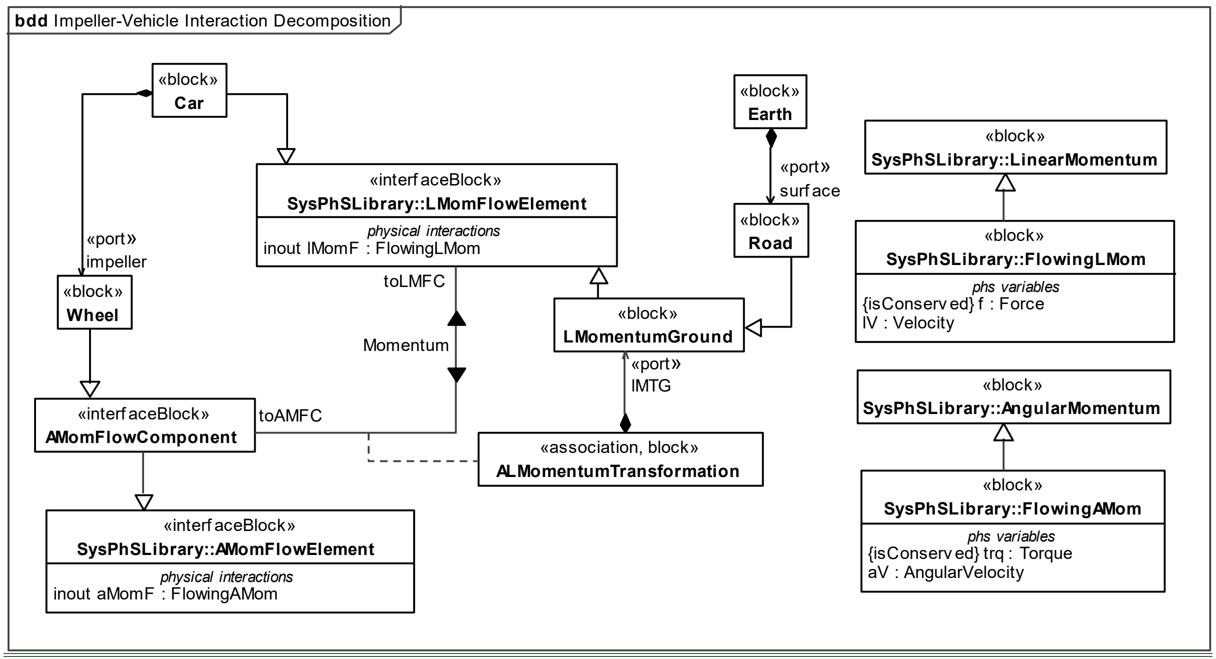
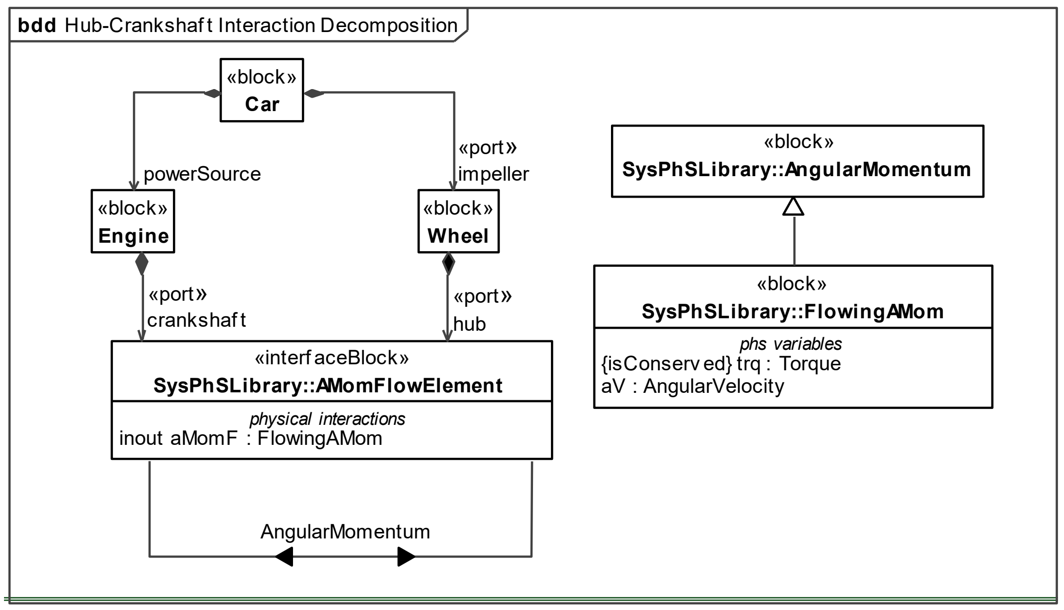
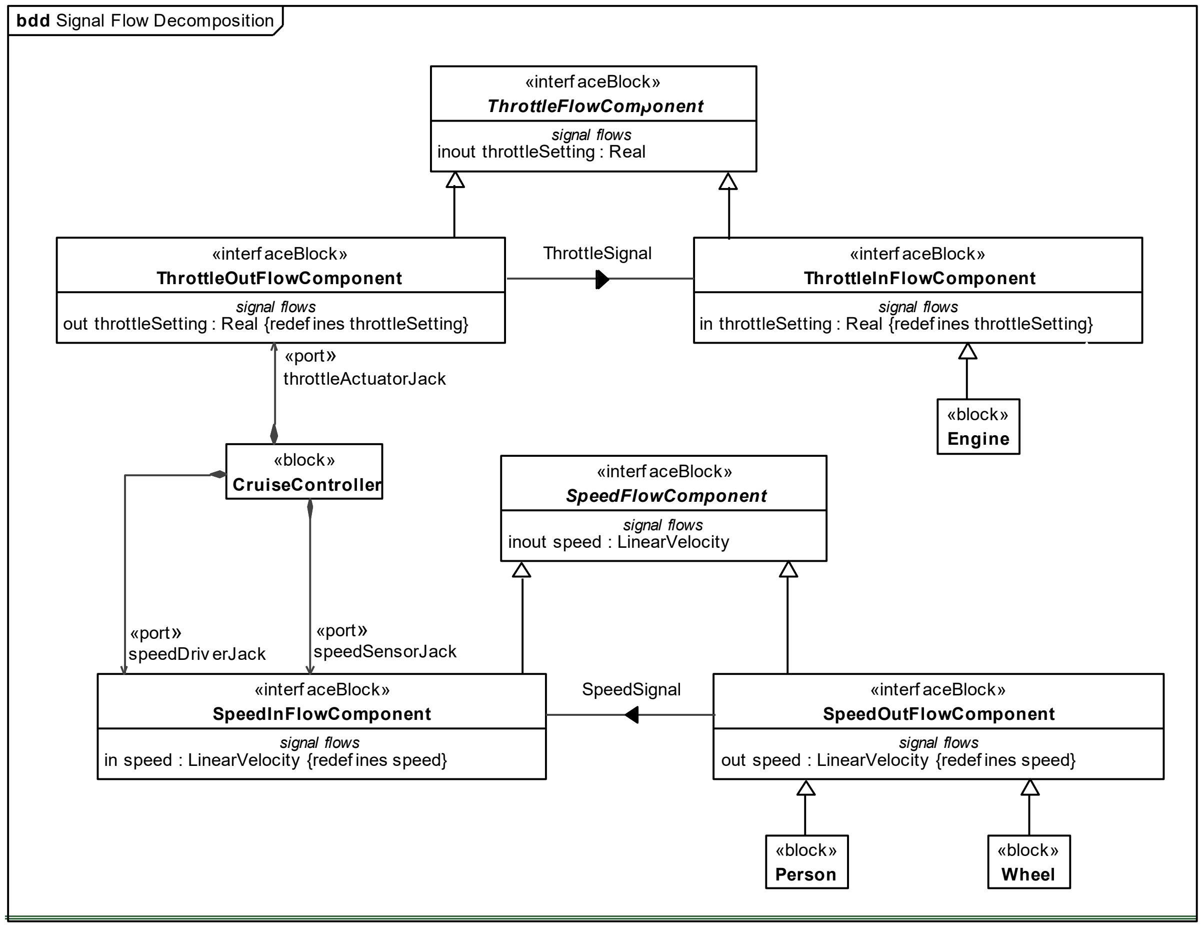

bdd Impeller-Vehicle Interaction Decomposition
«block» Car
«block» Earth
«port» surface
«block» Road
«interfaceBlock» SysPhSLibrary::LMomFlowElement
physical interactions
inout lMomF : FlowingLMom
«block» SysPhSLibrary::LinearMomentum
«block» SysPhSLibrary::FlowingLMom
phs variables
{isConserved} f : Force
lV : Velocity
«port» impeller
«block» Wheel
toLMFC
Momentum
«block» LMomentumGround
«port» lMTG
«interfaceBlock» AMomFlowComponent
toAMFC
«association, block» ALMomentumTransformation
«block» SysPhSLibrary::AngularMomentum
«block» SysPhSLibrary::FlowingAMom
phs variables
{isConserved} trq : Torque
aV : AngularVelocity
«interfaceBlock» SysPhSLibrary::AMomFlowElement
physical interactions
inout aMomF : FlowingAMom

Figure 108: Impeller-vehicle interaction blocks, ports, & component properties


bdd Hub-Crankshaft Interaction Decomposition
«block» Car
powerSource
«port» impeller
«block» Engine
«block» Wheel
«port» crankshaft
«port» hub
«interfaceBlock» SysPhSLibrary::AMomFlowElement
physical interactions
inout aMomF : FlowingAMom
AngularMomentum
«block» SysPhSLibrary::AngularMomentum
«block» SysPhSLibrary::FlowingAMom
phs variables
{isConserved} trq : Torque
aV : AngularVelocity

~~Figure 109: Hub-crankshaft interaction blocks, ports, & component properties~~


bdd Signal Flow Decomposition
«interfaceBlock» ThrottleFlowComponent
signal flows
inout throttleSetting : Real
«interfaceBlock» ThrottleOutFlowComponent
signal flows
out throttleSetting : Real {redefines throttleSetting}
ThrottleSignal
«interfaceBlock» ThrottleInFlowComponent
signal flows
in throttleSetting : Real {redefines throttleSetting}
«port» throttleActuatorJack
«block» Engine
«block» CruiseController
«interfaceBlock» SpeedFlowComponent
signal flows
inout speed : LinearVelocity
«port» speedDriverJack
«port» speedSensorJack
«interfaceBlock» SpeedInFlowComponent
signal flows
in speed : LinearVelocity {redefines speed}
SpeedSignal
«interfaceBlock» SpeedOutFlowComponent
signal flows
out speed : LinearVelocity {redefines speed}
«block» Person
«block» Wheel

~~Figure 110: Signal flow interactions blocks, ports, & component properties~~

~~A.15.2~~A.1.1 ~~Properties (variables)~~

~~Signal flow is the movement of numbers between system components. These numbers might reflect physical quantities or not. In this example, they do not (see Subannex A.5 for an example where they do). Signals flowing in and out of components are modeled by ports typed by interface blocks that have flow properties typed by numbers. In this example, signal flow ports are typed by *SpeedInFlowComponent, SpeedOutFlowComponent, ThrottleInFlowComponent,* or *ThrottleOutFlowComponent. SpeedInFlowComponent* and *SpeedOutFlowComponent* are generalized by the block *SpeedFlowComponent,* which has the flow property *speed* typed by *Linear Velocity*, as shown in Figure 110. *ThrottleInFlowComponent* and *ThrottleOutFlowComponent* are generalized by the block *ThrottleFlowComponent,*~~

~~which has the flow property *throttleSettling* typed by *Real*, from SysML, as shown in Figure 110~~

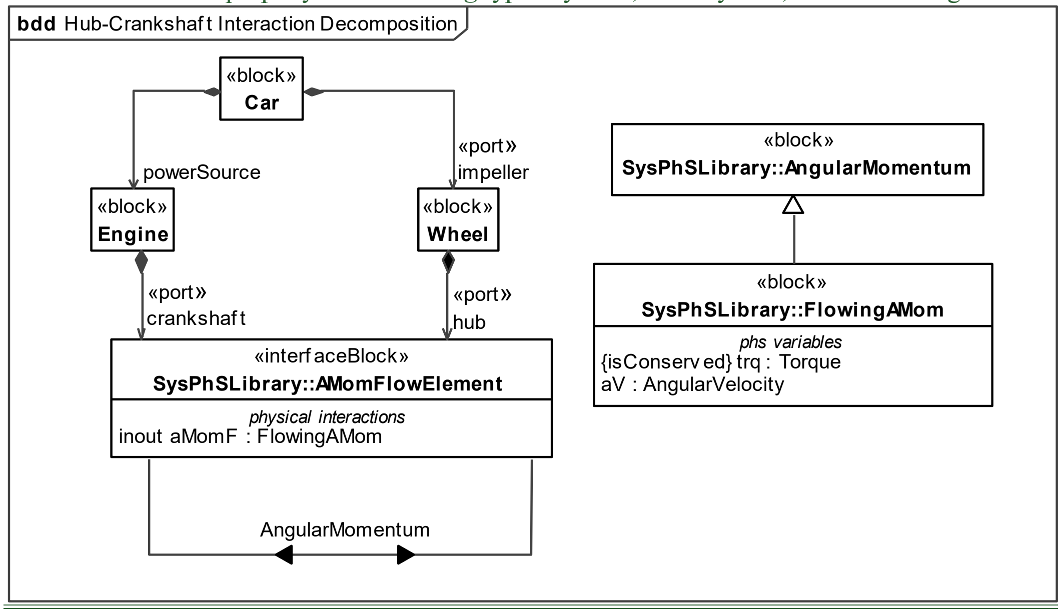

bdd Hub-Crankshaft Interaction Decomposition
«block» Car
powerSource
«port» impeller
«block» Engine
«block» Wheel
«port» crankshaft
«port» hub
«interfaceBlock» SysPhSLibrary::AMomFlowElement
physical interactions
inout aMomF : FlowingAMom
AngularMomentum
«block» SysPhSLibrary::AngularMomentum
«block» SysPhSLibrary::FlowingAMom
phs variables
{isConserved} trq : Torque
aV : AngularVelocity

Figure 109: Hub-crankshaft interaction blocks, ports, & component properties

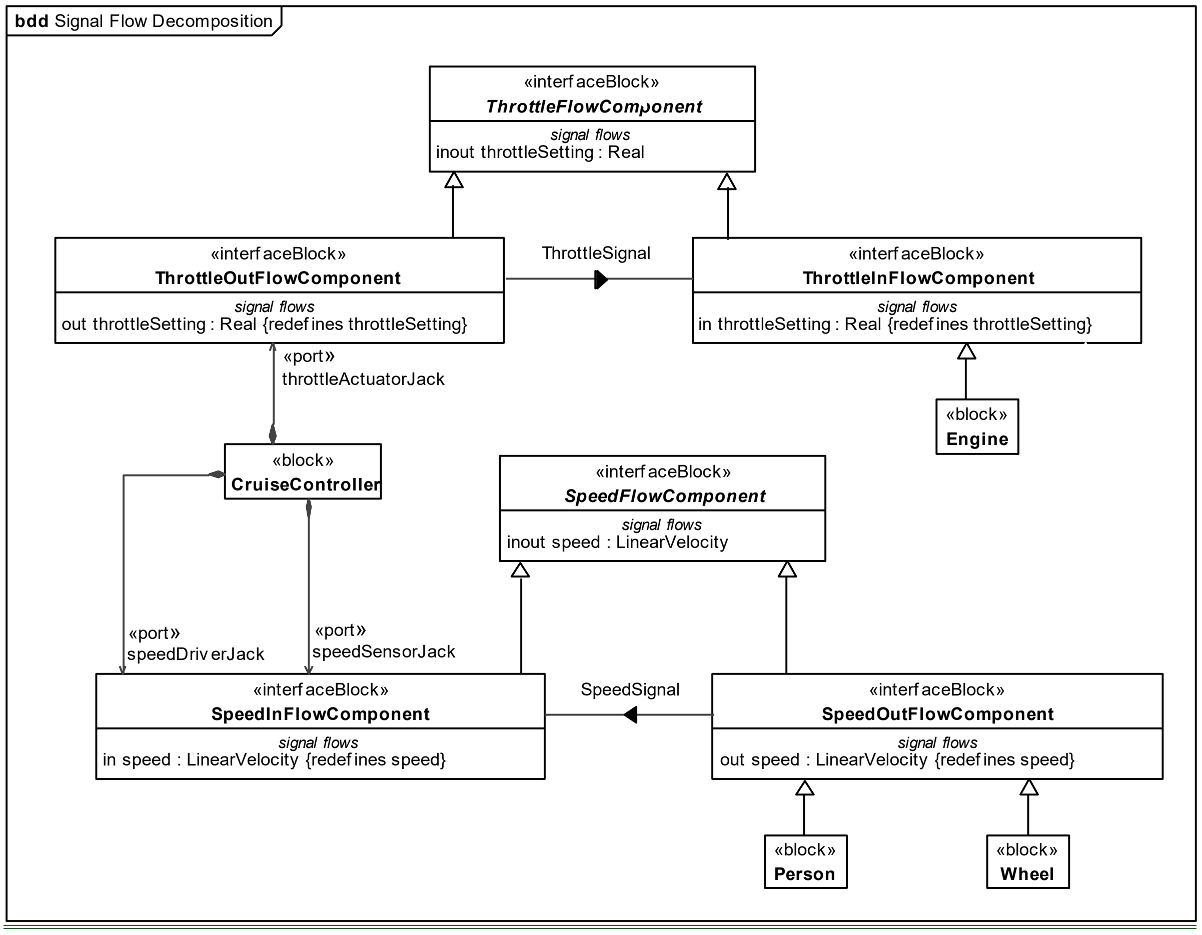

bdd Signal Flow Decomposition
«interfaceBlock» ThrottleFlowComponent
signal flows
inout throttleSetting : Real
«interfaceBlock» ThrottleOutFlowComponent
signal flows
out throttleSetting : Real {redefines throttleSetting}
ThrottleSignal
«interfaceBlock» ThrottleInFlowComponent
signal flows
in throttleSetting : Real {redefines throttleSetting}
«port» throttleActuatorJack
«block» Engine
«block» CruiseController
«interfaceBlock» SpeedFlowComponent
signal flows
inout speed : LinearVelocity
«port» speedDriverJack
«port» speedSensorJack
«interfaceBlock» SpeedInFlowComponent
signal flows
in speed : LinearVelocity {redefines speed}
SpeedSignal
«interfaceBlock» SpeedOutFlowComponent
signal flows
out speed : LinearVelocity {redefines speed}
«block» Person
«block» Wheel

Figure 110: Signal flow interactions blocks, ports, & component properties

A.6.5 Properties (variables)

Signal flow is the movement of numbers between system components. These numbers might reflect physical quantities or not. In this example, they do not (see Subannex A.5 for an example where they do). Signals flowing in and out of components are modeled by ports typed by interface blocks that have flow properties typed by numbers. In this example, signal flow ports are typed by *SpeedInFlowComponent, SpeedOutFlowComponent, ThrottleInFlowComponent,* or *ThrottleOutFlowComponent. SpeedInFlowComponent* and *SpeedOutFlowComponent* are generalized by the block *SpeedFlowComponent,* which has the flow property *speed* typed by *Linear Velocity*, as shown in Figure 110. *ThrottleInFlowComponent* and *ThrottleOutFlowComponent* are generalized by the block

ThrottleFlowComponent, which has the flow property *throttleSettling* typed by *Real*, from SysML, as shown in ~~Figure~~ 110~~. This value type has no unit, reflecting that the signals are not measurements of physical quantities and do not follow conservation laws.~~

~~Physical interaction is the movement of physical substances between system components, modeled in terms of conserved characteristics of the substances. In this example, linear and angular momentum are the conserved characteristics moving through the car (momentum moves without an associated physical substance) and between the car and the environment for the driving force and environmental disturbances from gravity or surrounding air. Movement is described by numeric variables for flow rate and potential to flow of conserved characteristics. In this example, movement of linear and angular momentum is characterized by force and torque variables for the flow rate as well as linear and angular velocity variables for potential to flow. The flow rate variable is conserved (values on ends of the interaction sum to zero) and the potential variable is not (values on ends of the interaction are the same). This is modeled in three parts:~~

- ~~Conserved physical characteristics are modeled as blocks directly specializedfrom *ConservedQuantityKind* in the physical interaction library (see Subclause 11.2.2), *LinearMomentum* and *AngularMomentum* in this example.~~

- ~~Flow variables are modeled as properties with the PhSVariable stereotype applied on specializations of conserved quantity kind blocks. In this example, the linear momentum flow rate and potential PhSVariables are *f* and *lV* on *FlowLMom* (*f* marked as *isConserved*), respectively, typed by *Force* and *Velocity*, respectively, all from the physical interaction library. Similarly, the angular momentum flow rate and potential PhSVariables are *trq* and *aV* on *FlowAMom* (*trq* marked as *is Conserved*), respectively, typed by *Torque* and *Angular Velocity*, respectively.~~

- ~~Flows in and out of components are modeled by ports typed by interface blocks that have flow properties typed by flowing conserved quantity kinds. In this example, ports are typed by *LMomFlowElement* or *AMomFlowElement* from the physical interaction library, which have flow property *lMomF* typed by *FlowingLMom* and flow property *aMomF* typed by *FlowingAMom*, respectively, shown in Figure 106 through Figure 109.~~

~~In Figure 105: Total system blocks, ports, & component propertiesthe blocks *LMomPotEngTransformation, Road,* and *CruiseController* have properties with PhSVariable stereotypes applied, specifying that the value of these properties may vary during simulation. The blocks *Car, Earth, Engine, Person, Air, Road, CruiseController,* and *AMomFlowComponent* have properties with PhSConstant stereotypes applied, specifying that the value of these properties do not change during each simulation run.~~

~~A.15.3 Constraints (equations)~~

~~Equations define mathematical relationships between the values of numeric variables. Equations in SysML, are constraints in constraint blocks that use properties of the blocks (parameters) as variables. In this example, the constraint blocks in Figure 111 each define parameters and constraints for component blocks in Figure 105 (*Car, Air, LMomPoteEngTransformation, Wheel, Road, Engine, CruiseController,* and *Person*). Constraint blocks for components are named according to the component they constrain. The constraint block *ALMomTransConstraint* defines parameters and constraints for the association block *ALMomentumTransformation*, and *FluidEffectConstraint* defines the parameters and constraints for the association block *LMomentumTransfer*. The constraints for *Air, Road,* and *Person* are not generally- applicable equations as they are for the other blocks. They are only for when the air is still (has no velocity), the road slope changes at two distinct times for a specified slope, and the driver changes the vehicle's speed at two separate time-stamps. The scenario has been defined with parameters in the constraint blocks for brevity, but their properties can also be defined with block property redefinitions (Subannex A.5.9) or by initial values in internal block diagrams (Subannex A.4.3).~~

~~The constraint blocks *PersonConstraint* and *CruiseControllerConstraint* specify manipulation of signals moving through their respective component block. The cruise controller constraint calculates the best fuel injection rate to reach the driver's desired vehicle speed from vehicle's current speed. All the other constraints specify physical interactions, either between components in the car (angular momentum between the engine and wheels) or between the car and its environment (angular momentum of the wheels to and from linear momentum of the car or air, to and from potential energy, or to heat due to wheel rolling resistance).~~Figure 110. This value type has no unit, reflecting that the signals are not measurements of physical quantities and do not follow conservation laws.

Physical interaction is the movement of physical substances between system components, modeled in terms of conserved characteristics of the substances. In this example, linear and angular momentum are the conserved characteristics moving through the car (momentum moves without an associated physical substance) and between the car and the environment for the driving force and environmental disturbances from gravity or surrounding air. Movement is described by numeric variables for flow rate and potential to flow of conserved characteristics. In this example, movement of linear and angular momentum is characterized by force and torque variables for the flow rate as well as linear and angular velocity variables for potential to flow. The flow rate variable is conserved (values on ends of the interaction sum to zero) and the potential variable is not (values on ends of the interaction are the same). This is modeled in three parts:

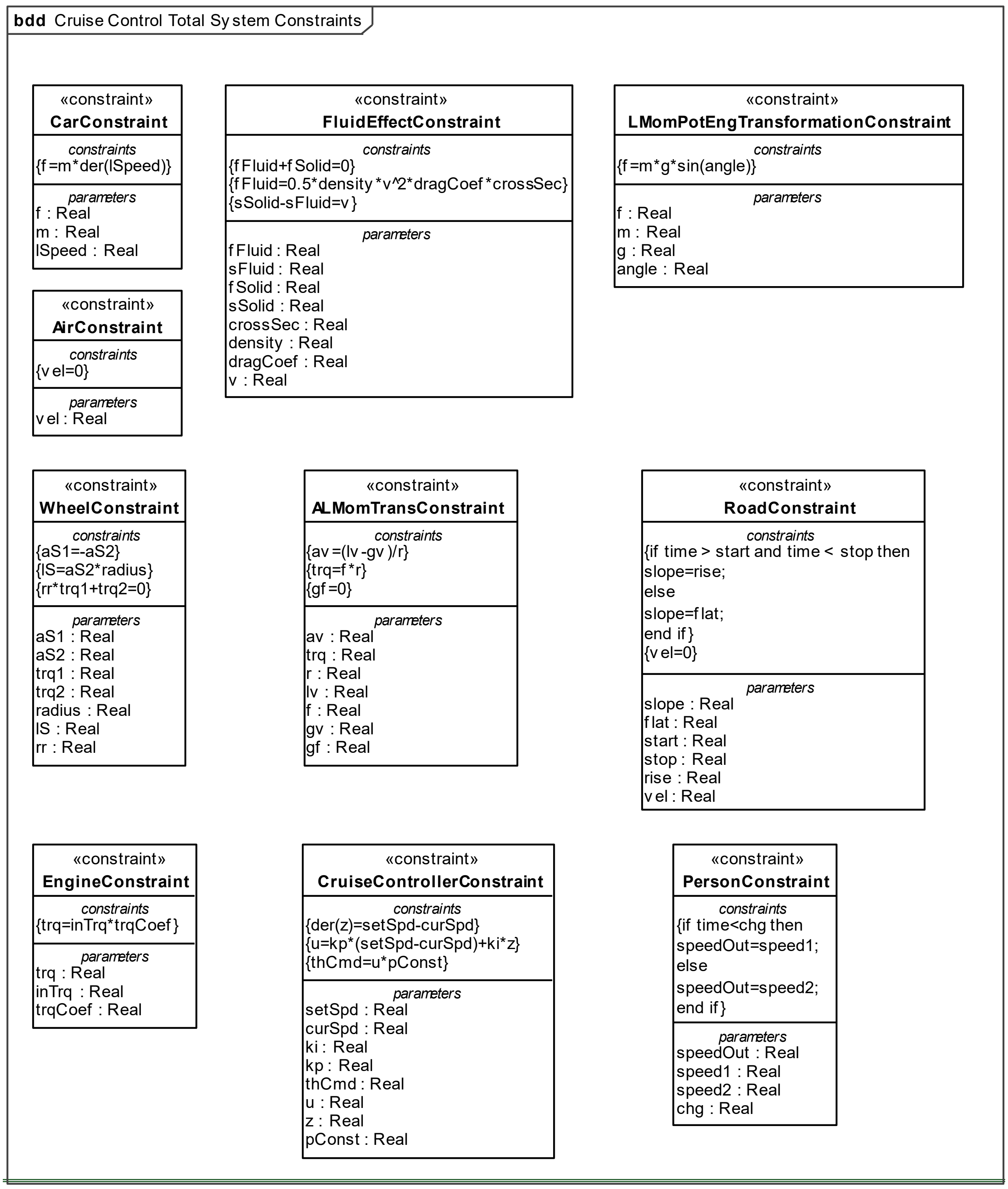
- Conserved physical characteristics are modeled as blocks directly specializedfrom*ConservedQuantityKind* in the physical interaction library (see Subclause 11.2.2), *LinearMomentum* and *AngularMomentum* in this example.
- Flow variables are modeled as properties with the PhSVariable stereotype applied on specializations of conserved quantity kind blocks. In this example, the linear momentum flow rate and potential PhSVariables are *f* and *lV* on *FlowLMom* (*f* marked as *isConserved*), respectively, typed by *Force* and *Velocity*, respectively, all from the physical interaction library. Similarly, the angular momentum flow rate and potential PhSVariables are *trq* and *aV* on *FlowAMom* (*trq* marked as *is Conserved*), respectively, typed by *Torque* and *Angular Velocity*, respectively.
- Flows in and out of components are modeled by ports typed by interface blocks that have flow properties typed by flowing conserved quantity kinds. In this example, ports are typed by *LMomFlowElement* or *AMomFlowElement* from the physical interaction library, which have flow property *lMomF* typed by *FlowingLMom* and flow property *aMomF* typed by *FlowingAMom*, respectively, shown in Figure 106 through Figure 109.

In Figure 105: Total system blocks, ports, & component propertiesthe blocks *LMomPotEngTransformation, Road,* and *CruiseController* have properties with PhSVariable stereotypes applied, specifying that the value of these properties may vary during simulation. The blocks *Car, Earth, Engine, Person, Air, Road, CruiseController,* and *AMomFlowComponent* have properties with PhSConstant stereotypes applied, specifying that the value of these properties do not change during each simulation run.

A.6.6 Constraints (equations)

Equations define mathematical relationships between the values of numeric variables. Equations in SysML, are constraints in constraint blocks that use properties of the blocks (parameters) as variables. In this example, the constraint blocks in Figure 111 each define parameters and constraints for component blocks in Figure 105 (*Car, Air, LMomPoteEngTransformation, Wheel, Road, Engine, CruiseController,* and *Person*). Constraint blocks for components are named according to the component they constrain. The constraint block *ALMomTransConstraint* defines parameters and constraints for the association block *ALMomentumTransformation*, and *FluidEffectConstraint* defines the parameters and constraints for the association block *LMomentumTransfer*. The constraints for *Air, Road,* and *Person* are not generally- applicable equations as they are for the other blocks. They are only for when the air is still (has no velocity), the road slope changes at two distinct times for a specified slope, and the driver changes the vehicle's speed at two separate time-stamps. The scenario has been defined with parameters in the constraint blocks for brevity, but their properties can also be defined with block property redefinitions (Subannex A.5.9) or by initial values in internal block diagrams (Subannex A.4.3).

The constraint blocks *PersonConstraint* and *CruiseControllerConstraint* specify manipulation of signals moving through their respective component block. The cruise controller constraint calculates the best fuel injection rate to reach the driver's desired vehicle speed from vehicle's current speed. All the other constraints specify physical interactions, either between components in the car (angular momentum between the engine and wheels) or between the car and its environment (angular momentum of the wheels to and from linear momentum of the car or air, to and from potential energy, or to heat due to wheel rolling resistance).



~~**Figure 111: Cruise control total system constraint blocks**~~

~~A.15.4A.1.1 Constraint properties and bindings~~

~~Equations in constraint blocks are applied to components using binding connectors in component parametric diagrams. Component parametric diagrams show properties typed by constraint blocks (constraint properties), as well as component and port simulation variables and constants. Binding connectors link constraint parameters to simulation variables and constants, indicating their values must be the same. Figure 112 through Figure 119 show the parametric diagrams for the car, air, transformation between linear momentum and gravitational potential energy, wheel, road, engine, cruise controller, and person, respectively.~~

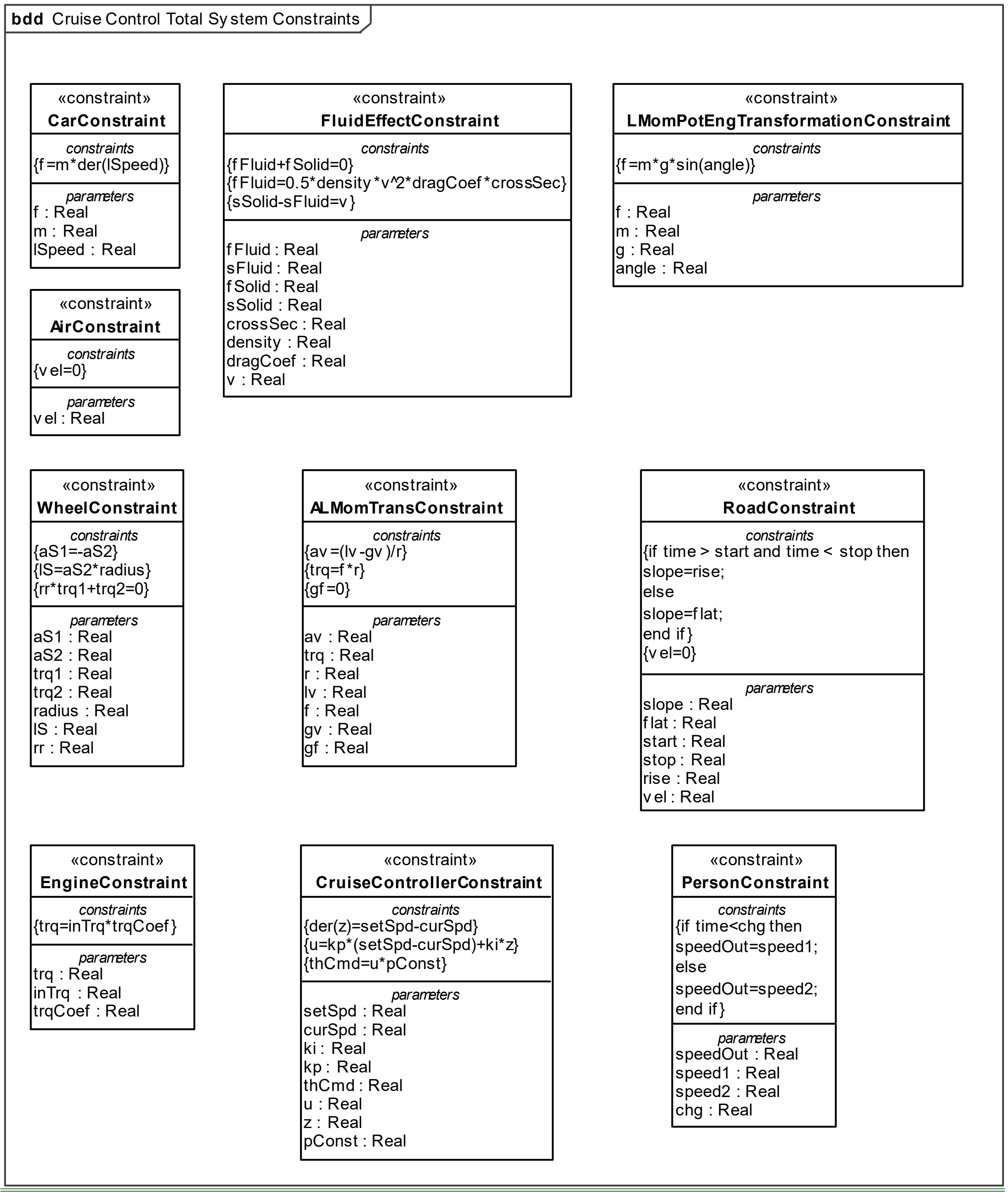
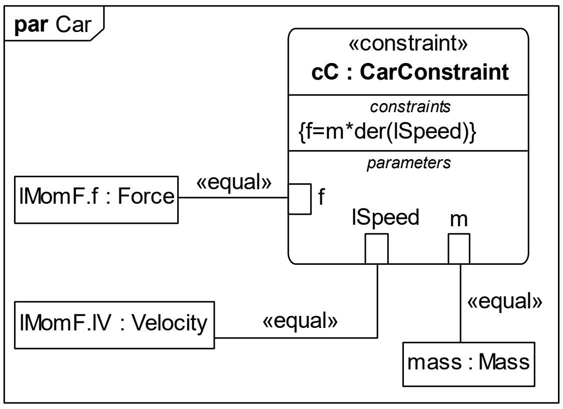


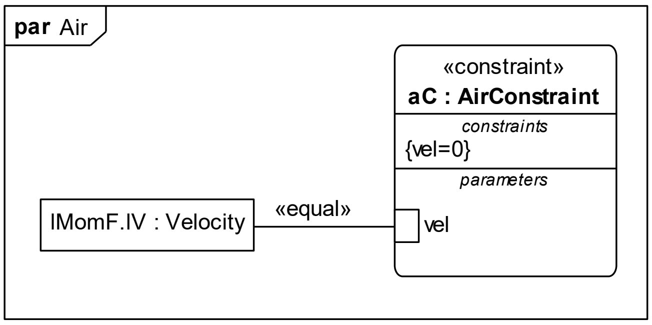
Figure 111: Cruise control total system constraint blocks

A.6.7 Constraint properties and bindings

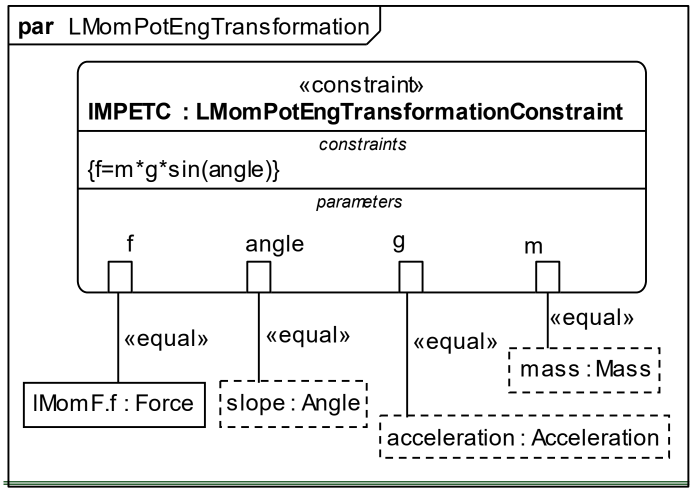
Equations in constraint blocks are applied to components using binding connectors in component parametric diagrams. Component parametric diagrams show properties typed by constraint blocks (constraint properties), as well as component and port simulation variables and constants. Binding connectors link constraint parameters to simulation variables and constants, indicating their values must be the same. Figure 112 through Figure 119 show the parametric diagrams for the car, air, transformation between linear momentum and gravitational potential energy, wheel, road, engine, cruise controller, and person, respectively.



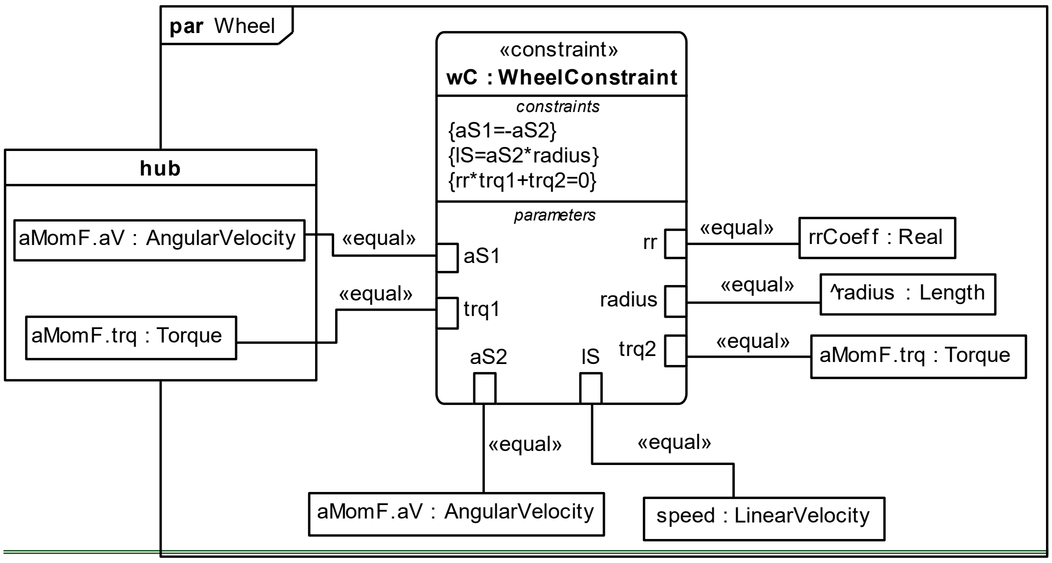
~~Figure 112: Parametric diagram applying the car constraint~~



~~Figure 113: Parametric diagram applying the air constraint~~



~~Figure 114: Parametric diagram applying the linear momentum-potential energy transformation constraint~~



~~Figure 115: Parametric diagram applying the wheel constraint~~

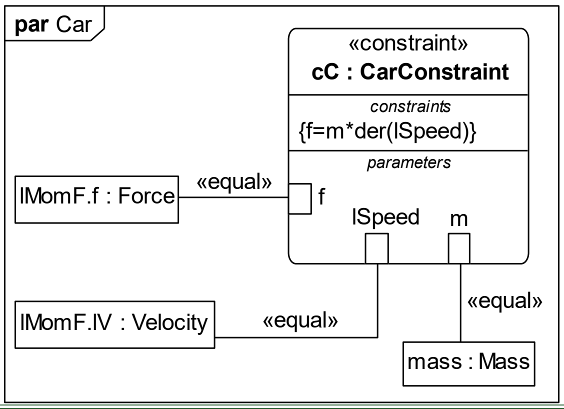


Figure 112: Parametric diagram applying the car constraint

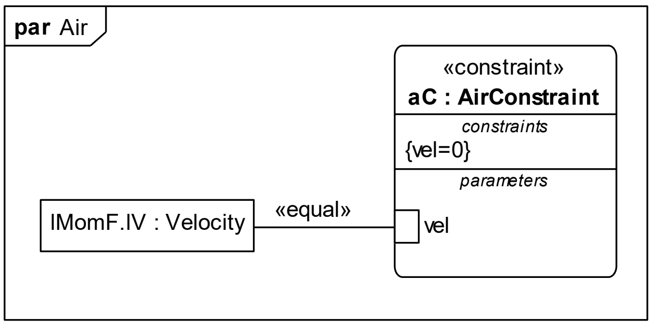


Figure 113: Parametric diagram applying the air constraint

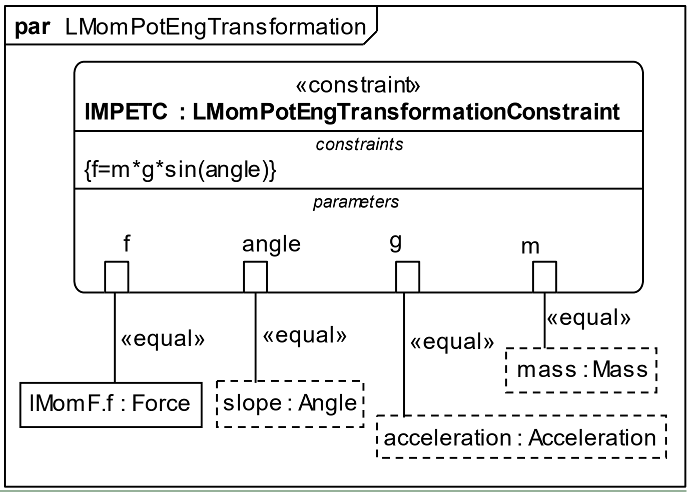


Figure 114: Parametric diagram applying the linear momentum-potential energy transformation constraint

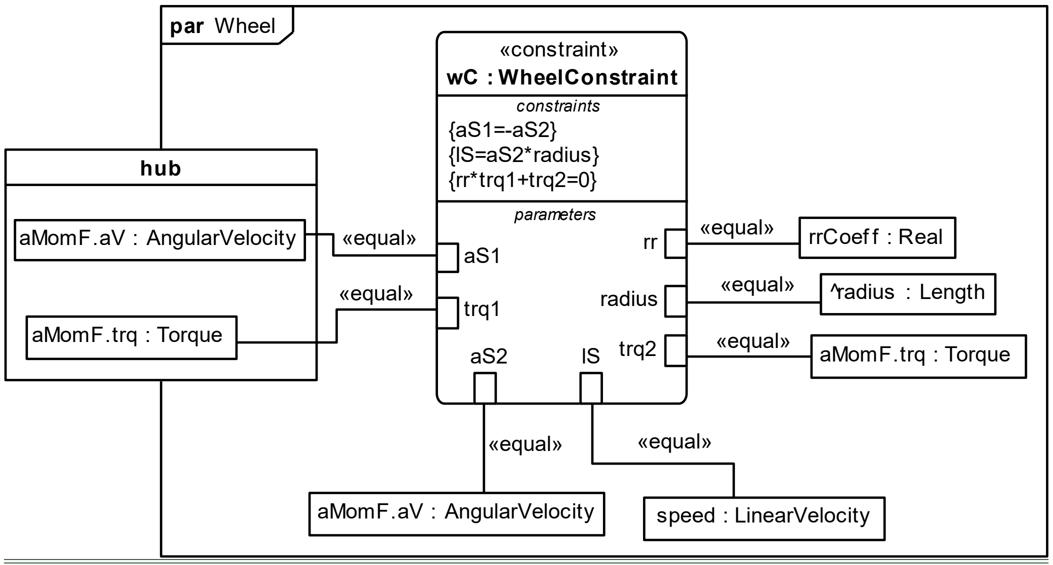
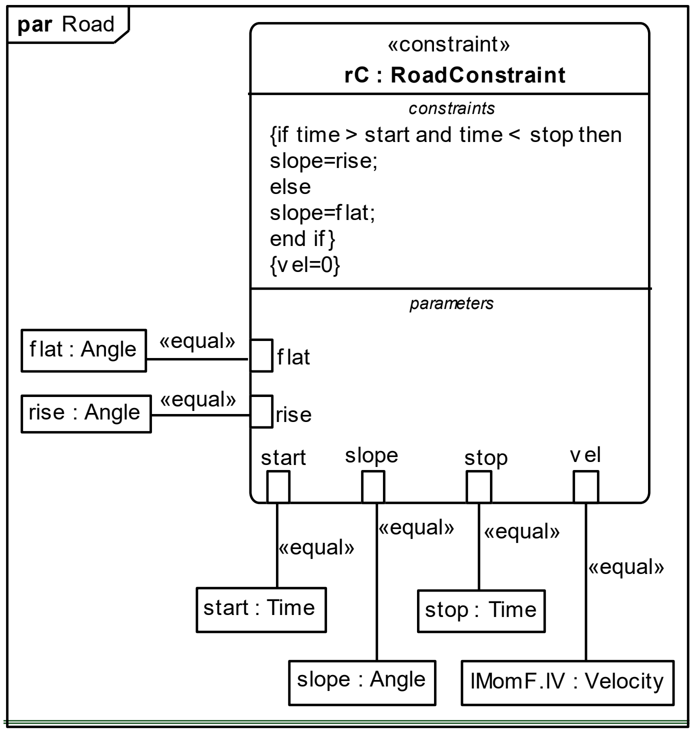
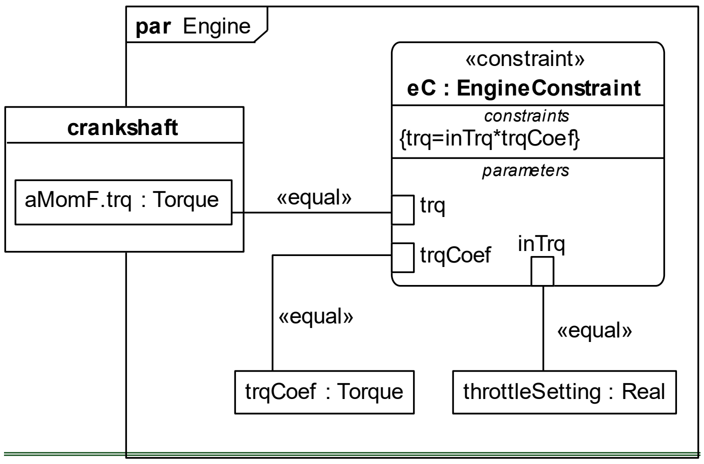


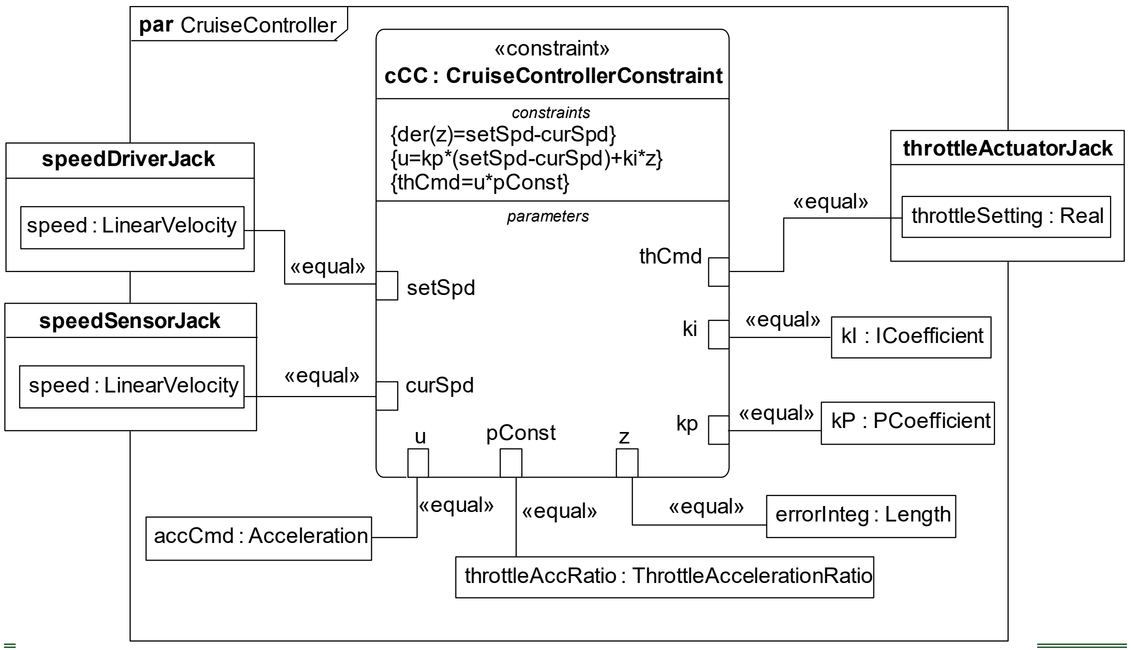
Figure 115: Parametric diagram applying the wheel constraint



~~Figure 116: Parametric diagram applying the road constraint~~



~~Figure 117: Parametric diagram applying the engine constraint~~



~~Figure 118: Parametric diagram applying the cruise controller constraint~~

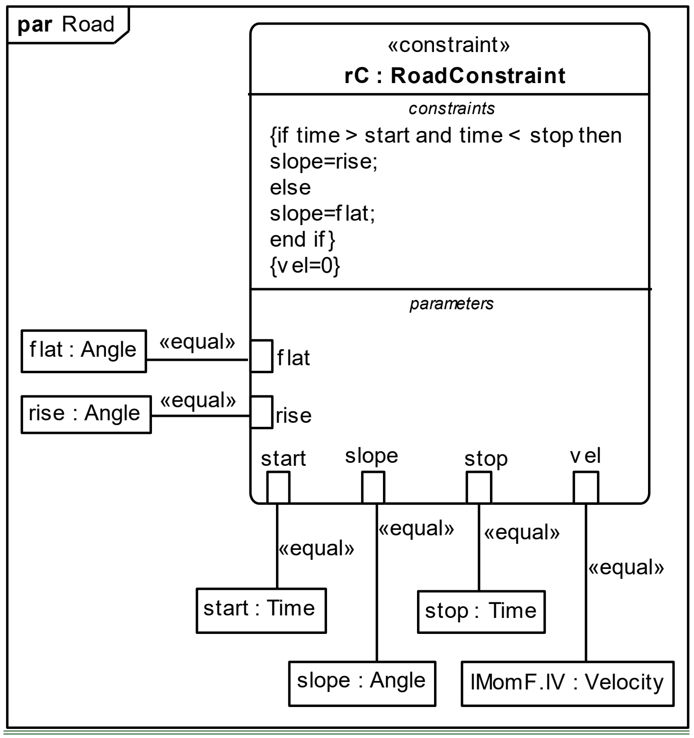


Figure 116: Parametric diagram applying the road constraint

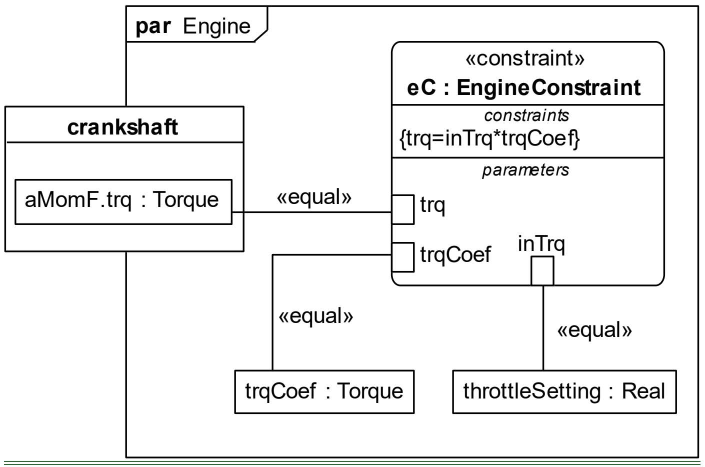


Figure 117: Parametric diagram applying the engine constraint

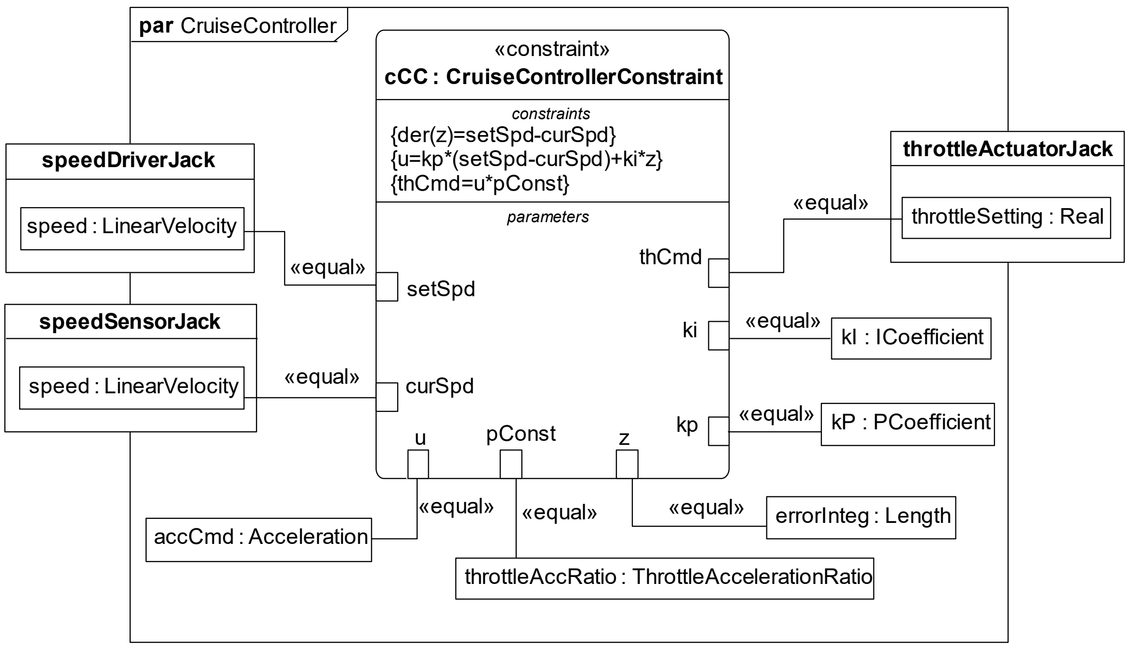
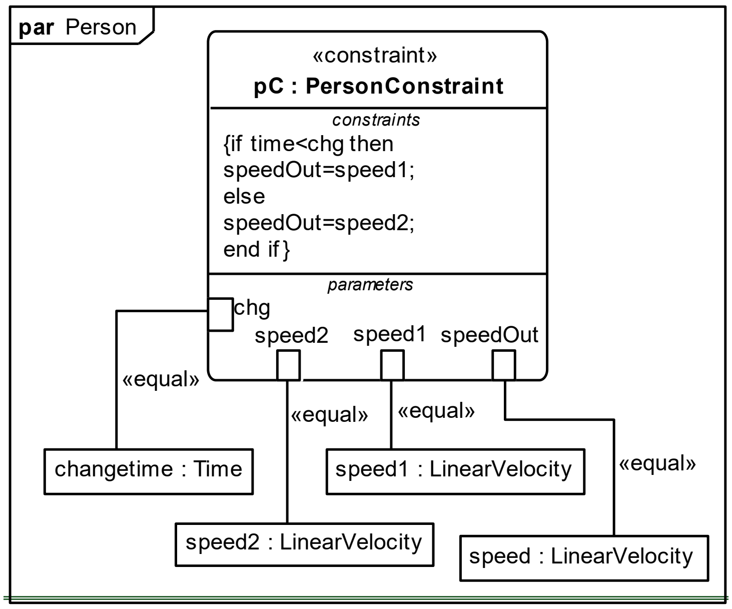
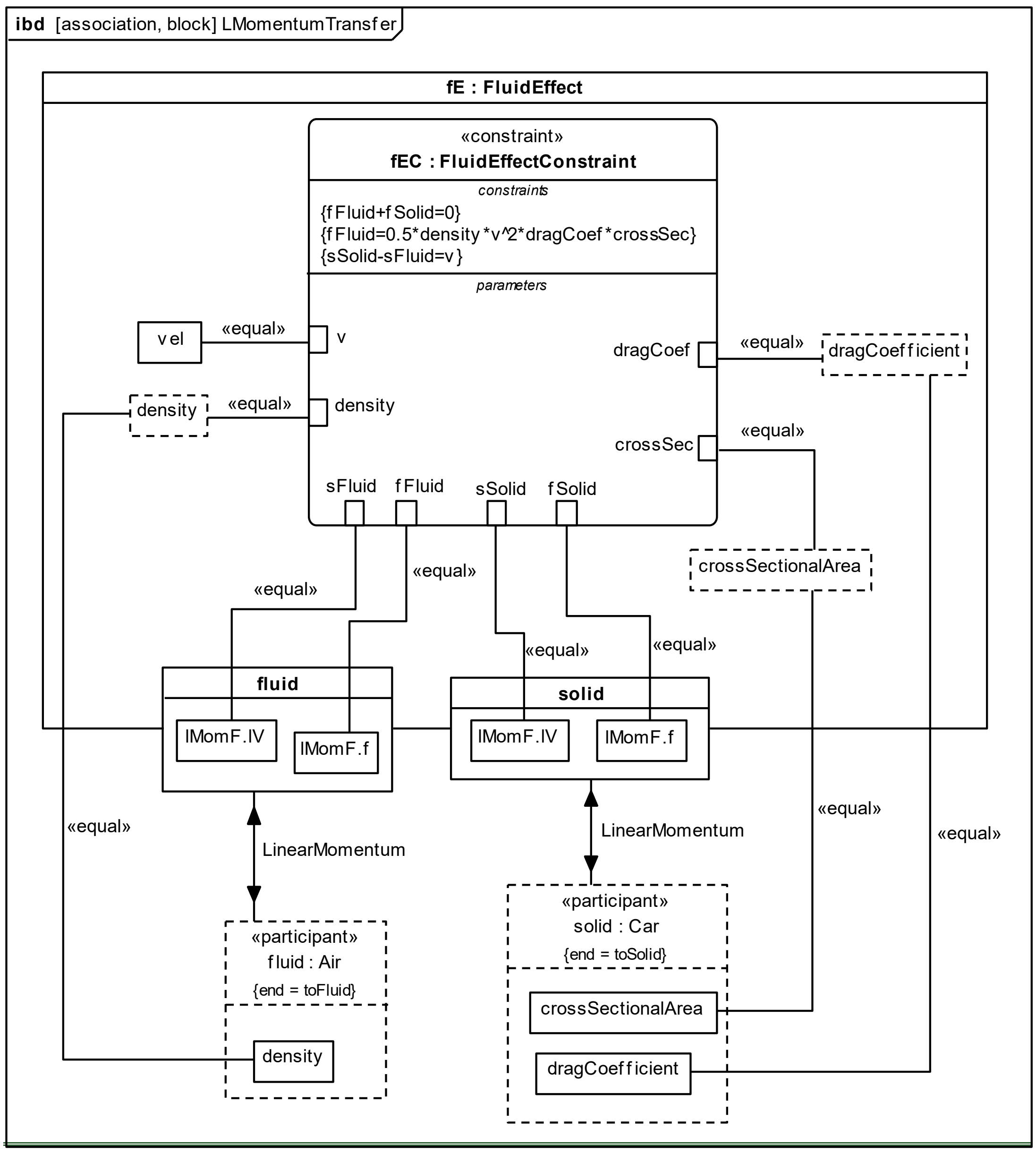


Figure 118: Parametric diagram applying the cruise controller constraint



~~Figure 119: Parametric diagram applying the person constraint~~

~~Figure 120 and Figure 121 are association block internal block diagrams rather than component parametric diagrams, to include connectors other than binding. These diagrams bind properties of the blocks linked by the association (participants) to variables and constants of a block inside the association.~~



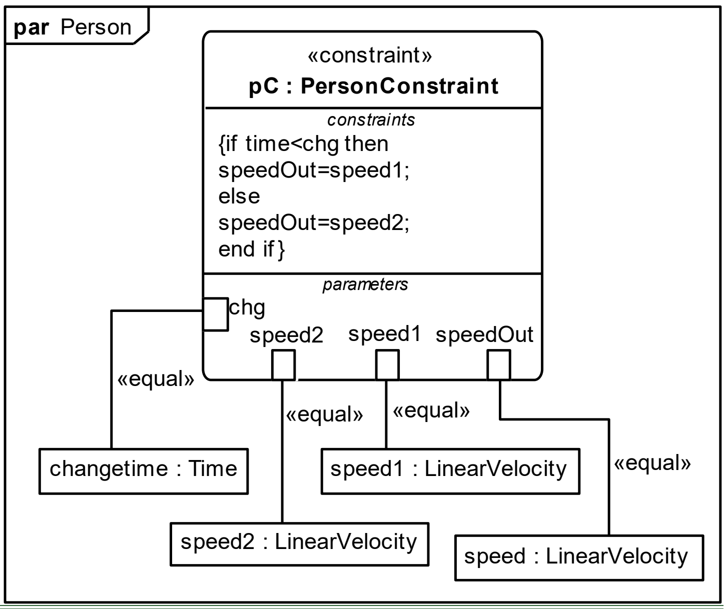
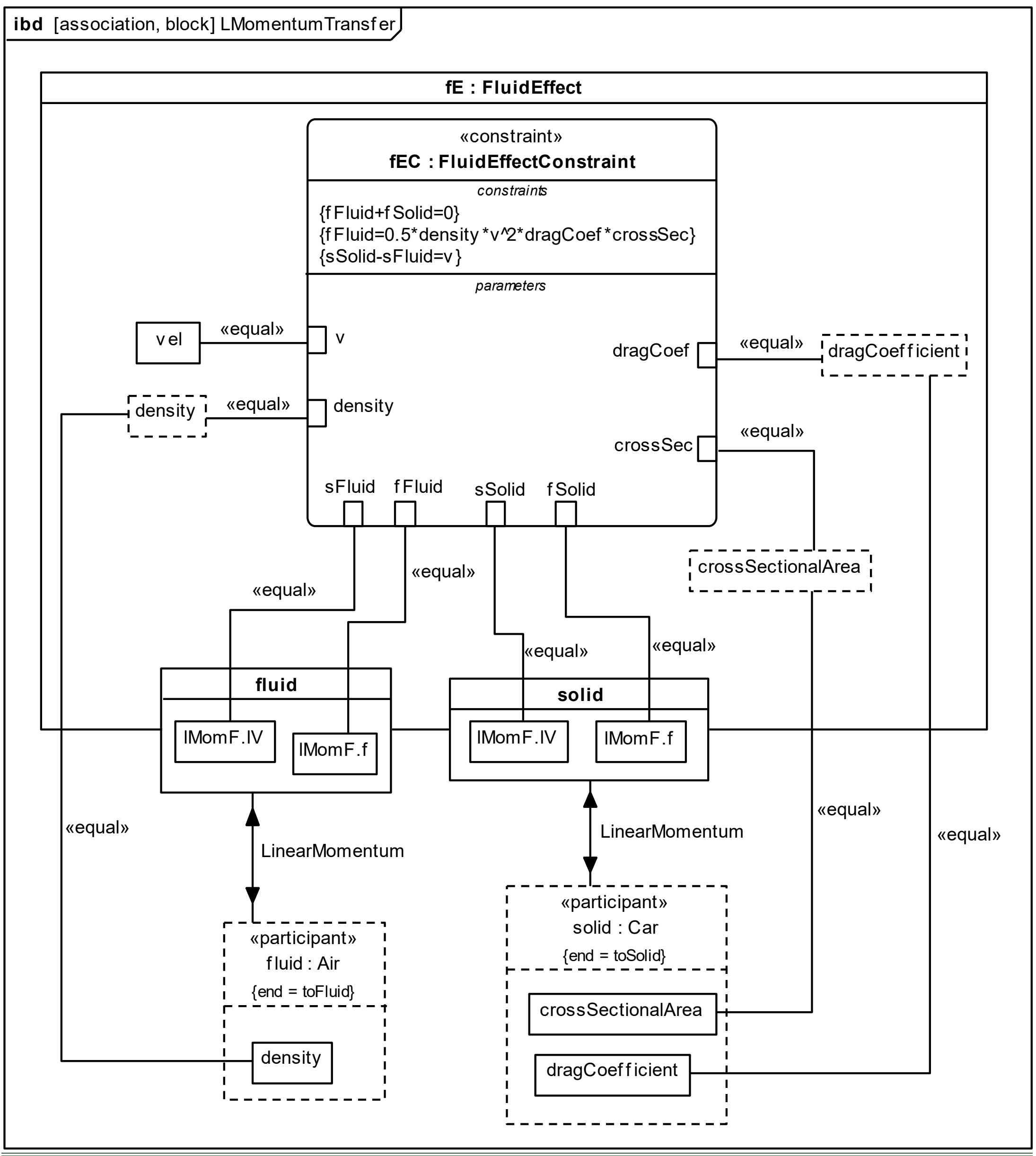
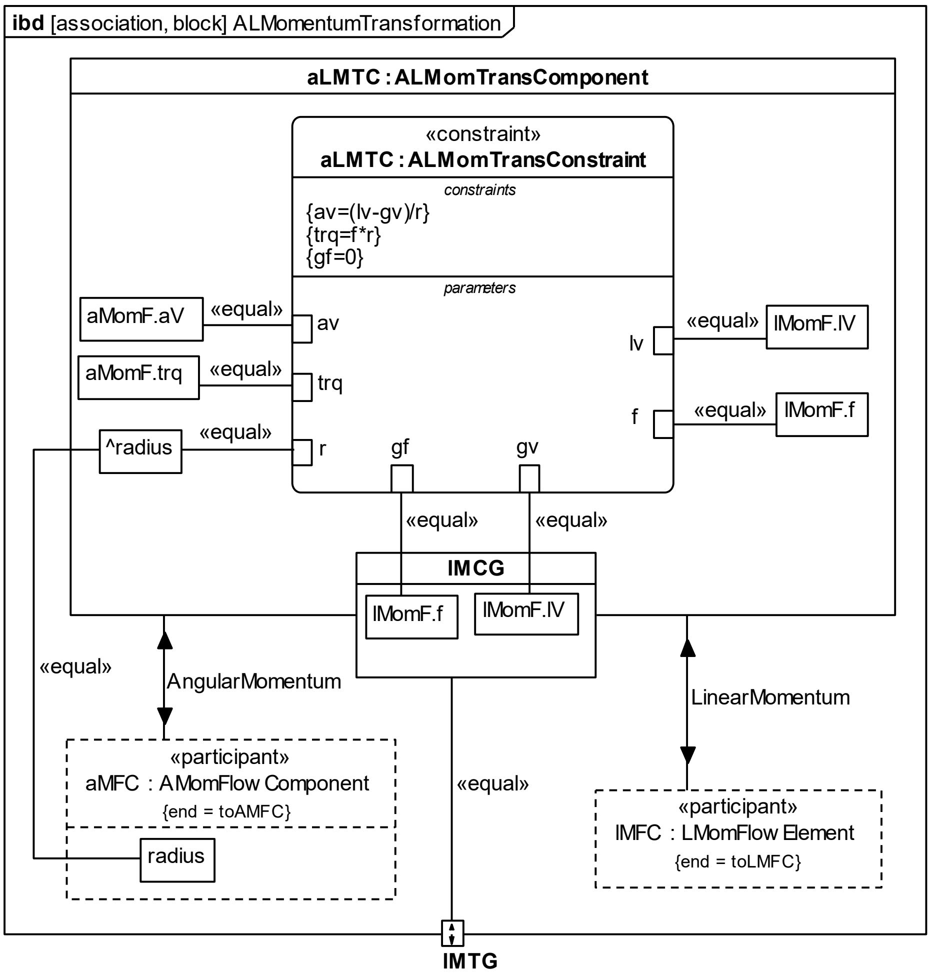


Figure 119: Parametric diagram applying the person constraint

Figure 120 and Figure 121 are association block internal block diagrams rather than component parametric diagrams, to include connectors other than binding. These diagrams bind properties of the blocks linked by the association (participants) to variables and constants of a block inside the association.

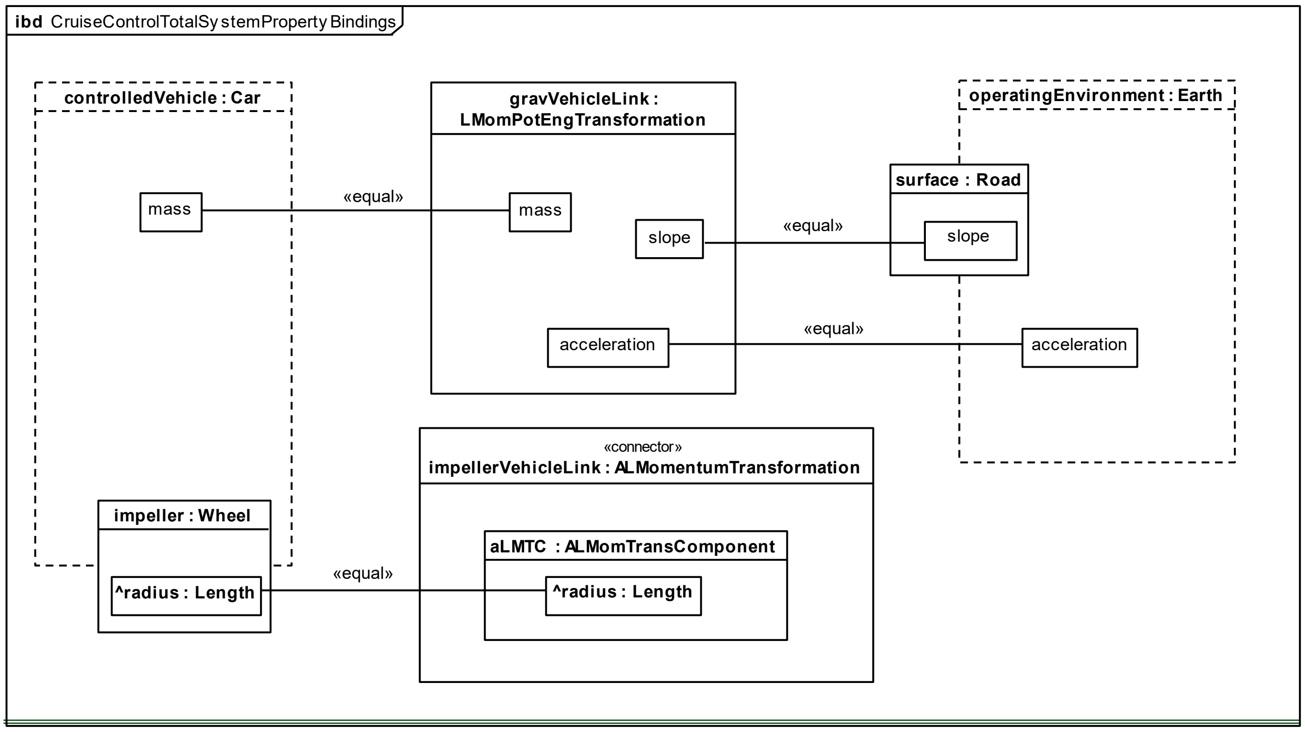


~~Figure 120: Internal block diagram applying the fluid effect constraint in the association block~~



~~**Figure 121: Internal block diagram applying a transformation constraint in the association block ALMomentumTransformation**~~

~~Figure 122 shows bindings between some value properties on separate components in Figure 104. For example, the values of some properties of the car and Earth parts are used in the gravitational potential energy block.~~


ibd CruiseControlTotalSystemPropertyBindings
controlledVehicle : Car
mass
«equal»
gravVehicleLink : LMomPotEngTransformation
mass
slope
acceleration
«equal»
«equal»
operatingEnvironment : Earth
surface : Road
slope
acceleration
impeller : Wheel
^radius : Length
«equal»
«connector»
impellerVehicleLink : ALMomentumTransformation
aLMTC : ALMomTransComponent
^radius : Length

~~**Figure 122: Internal block diagram applying property bindings across system components**~~

Figure 120: Internal block diagram applying the fluid effect constraint in the association block

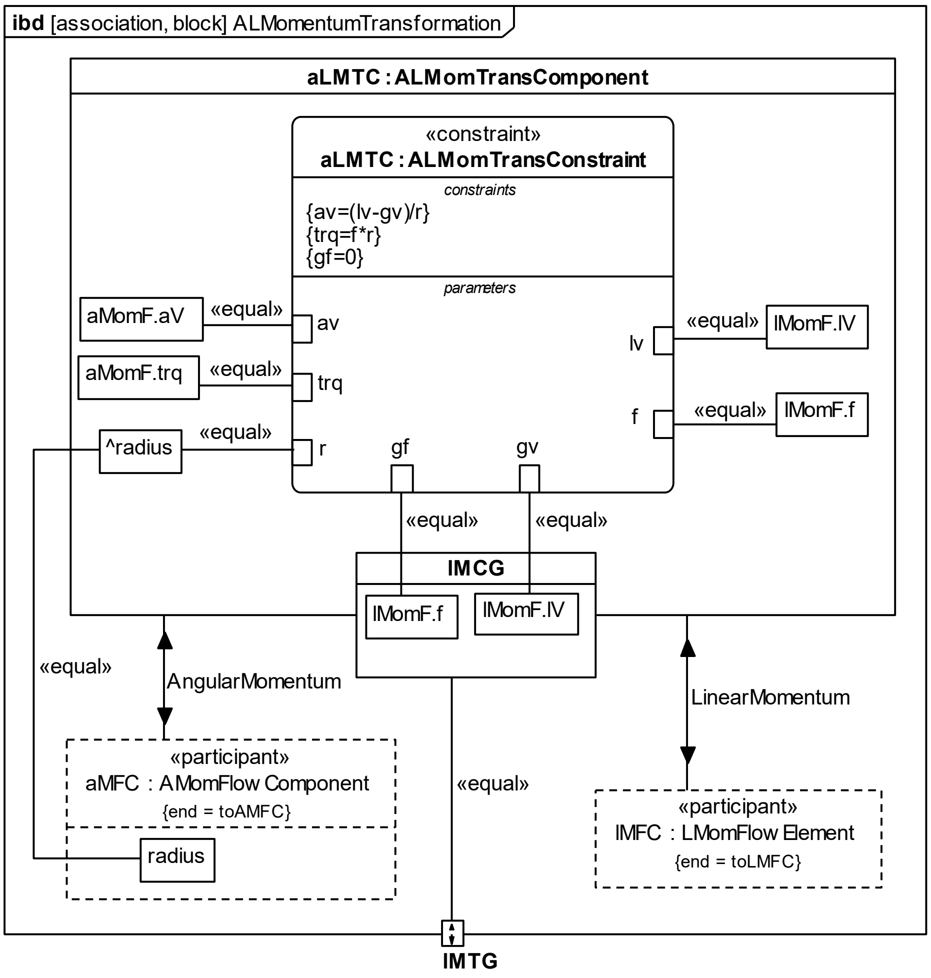


Figure 121: Internal block diagram applying a transformation constraint in the association block ALMomentumTransformation

Figure 122 shows bindings between some value properties on separate components in Figure 104. For example, the values of some properties of the car and Earth parts are used in the gravitational potential energy block.

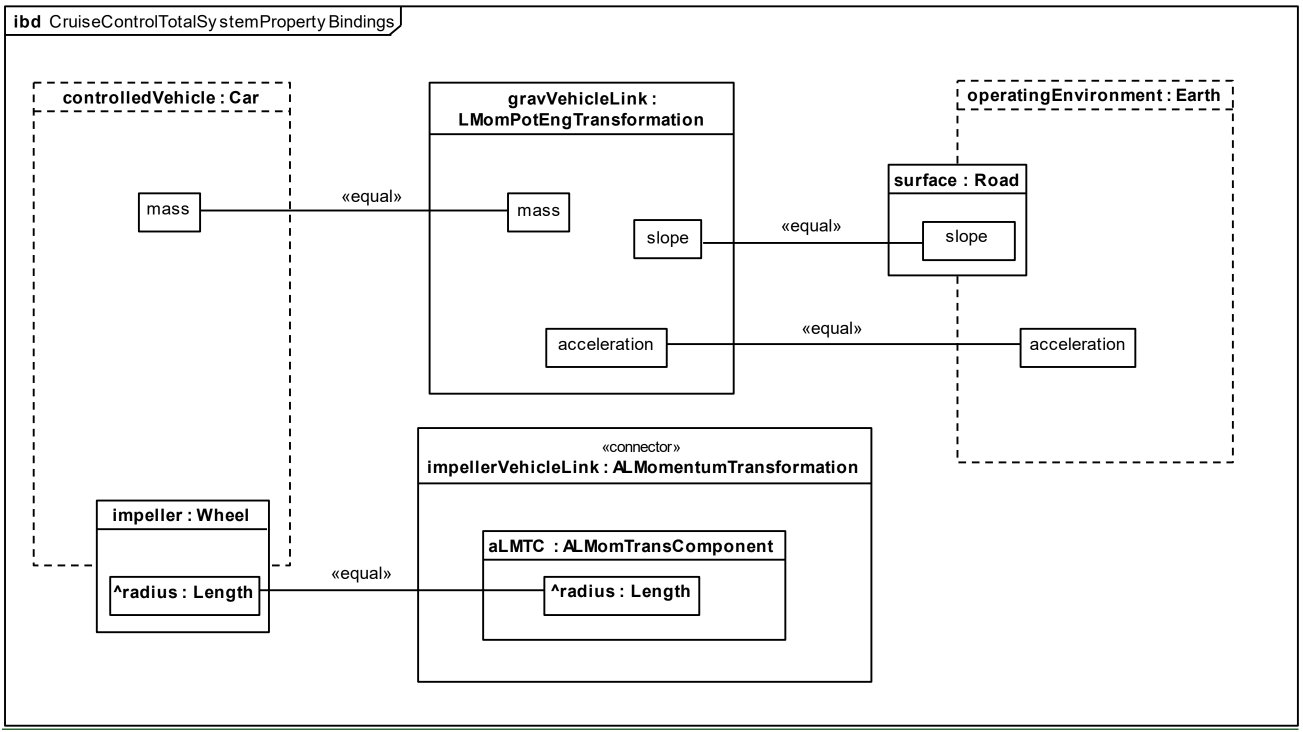

ibd CruiseControlTotalSystemPropertyBindings
controlledVehicle : Car
mass
«equal»
gravVehicleLink : LMomPotEngTransformation
mass
slope
acceleration
«equal»
«equal»
operatingEnvironment : Earth
surface : Road
slope
acceleration
impeller : Wheel
^radius : Length
«equal»
«connector»
impellerVehicleLink : ALMomentumTransformation
aLMTC : ALMomTransComponent
^radius : Length

Figure 122: Internal block diagram applying property bindings across system components

This page intentionally left blank.

Annex B - Platform-Independent Debugging (non-normative)

~~B.1 Introduction~~

~~It is helpful to identify causes of errors in earlier stages of system model development before they propagate to (potentially multiple) simulation models. It can also verify and increase understanding of the relationships captured in system models before discipline-specific experts focus on parts of the system in their own models and tools. Any errors not due to usage of SysML or its extensions, translators, or simulator execution engines will be in the source SysML models.~~

~~This annex gives an overview of platform-independent debugging procedures for physical interaction and signal flow in SysML models extended with SysPhS, before translation to simulation platforms. They are intended to complement existing debugging techniques on those platforms.~~

~~The type of failure influences the debugging procedures required to identify and fix errors. This annex is concerned with fixing system model errors that cause failure to:~~

- ~~Compile or execute simulation models translated from system models,~~
- ~~Produce expected results from simulation execution.~~

~~Failures of translation from extended SysML models to simulation due to incorrect usage of SysPhS or translator construction are not addressed.~~

~~Errors that cause failure to simulate arise from system model structure. These show the modeler's design does not properly support simulation. The underlying equations might be inconsistent, including being overconstrained (more equations than variables) or underconstrained (fewer equations than variables). The model might have equations that would divide by zero, functions being called outside of their real domain (such as the square root of a negative number), or other erroneous symbolic transformations.~~

~~Errors that cause simulations to produce unintended results arise from the meaning of the system model. These reflect discrepancies between desired behavior and simulation execution. Although some errors can be identified automatically depending on the simulation tool being used (such as variable values outside bounds), these errors can also be found manually after trying to validate the simulation results. These errors can come from incorrect equations, incorrect parameter or initialization values, and incorrect function calls from equations. Errors can also be due to integration errors with the solvers being used, which are not considered in this annex.~~

~~Debugging errors in physical interactions is more complicated than in signal flows, because following ordered execution of command sequences or operations does not work for bidirectional relationships (see Clause 6.1 on the bidirectionality of physical interactions). Debugging errors in physical interactions must examine chains of variable transformations in the model (mathematical operations on variables to give values to other variables).~~

~~This annex describes two debugging techniques for SysML system models of physical interactions and signal flows that are intended to be translated into simulation platforms:~~

- ~~Static debugging identifies errors that cause failure to compile simulation models to executable code. These techniques trace variable (symbolic) transformations through the model to identify erroneous sections.~~
- ~~Dynamic debugging identifies errors that cause simulation to produce unexpected results. These techniques involve interactive inspection of models during execution to bookkeep changing variable values over simulated time. They must be used after static debugging techniques to ensure models can be compiled to executable code.~~

~~The rest of this annex gives an overview of these debugging procedures applied to the vehicular cruise control system example from Subannex A.6.~~

~~B.2 Preprocessing: Simplifying Models~~

~~If a simulation model fails to compile or execute correctly, the cause can be identified by tracing through chains of connectors between components. This is the basis for static debugging techniques and facilitates dynamic debugging. It simplifies debugging to move physical interaction and signal flow connectors into separate models. The two simpler models can be debugged separately before replicating the resulting fixes in the complete model, a simpler task than debugging the entire model all at once.~~

~~First, create a model of physical interactions only by removing all connectors in the original model's internal block diagram (IBD) that do not represent physical interactions (saving a separate copy of the~~

~~original model first). Any remaining parts or ports that are not at the end of the remaining connectors or do not possess a port that is at the end of a~~ B.1 Introduction

It is helpful to identify causes of errors in earlier stages of system model development before they propagate to (potentially multiple) simulation models. It can also verify and increase understanding of the relationships captured in system models before discipline-specific experts focus on parts of the system in their own models and tools. Any errors not due to usage of SysML or its extensions, translators, or simulator execution engines will be in the source SysML models.

This annex gives an overview of platform-independent debugging procedures for physical interaction and signal flow in SysML models extended with SysPhS, before translation to simulation platforms. They are intended to complement existing debugging techniques on those platforms.

The type of failure influences the debugging procedures required to identify and fix errors. This annex is concerned with fixing system model errors that cause failure to:

- Compile or execute simulation models translated from system models,
- Produce expected results from simulation execution.

Failures of translation from extended SysML models to simulation due to incorrect usage of SysPhS or translator construction are not addressed.

Errors that cause failure to simulate arise from system model structure. These show the modeler's design does not properly support simulation. The underlying equations might be inconsistent, including being overconstrained (more equations than variables) or underconstrained (fewer equations than variables). The model might have equations that would divide by zero, functions being called outside of their real domain (such as the square root of a negative number), or other erroneous symbolic transformations.

Errors that cause simulations to produce unintended results arise from the meaning of the system model. These reflect discrepancies between desired behavior and simulation execution. Although some errors can be identified automatically depending on the simulation tool being used (such as variable values outside bounds), these errors can also be found manually after trying to validate the simulation results. These errors can come from incorrect equations, incorrect parameter or initialization values, and incorrect function calls from equations. Errors can also be due to integration errors with the solvers being used, which are not considered in this annex.

Debugging errors in physical interactions is more complicated than in signal flows, because following ordered execution of command sequences or operations does not work for bidirectional relationships (see Clause 6.1 on the bidirectionality of physical interactions). Debugging errors in physical interactions must examine chains of variable transformations in the model (mathematical operations on variables to give values to other variables).

This annex describes two debugging techniques for SysML system models of physical interactions and signal flows that are intended to be translated into simulation platforms:

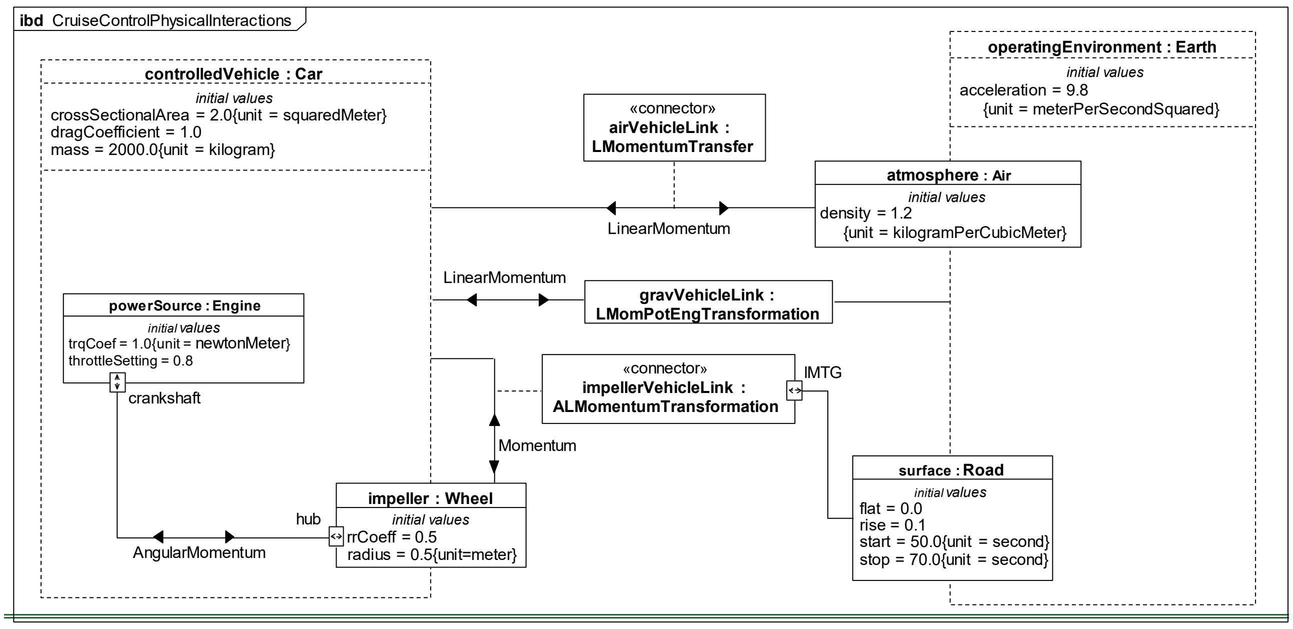
- Static debugging identifies errors that cause failure to compile simulation models to executable code. These techniques trace variable (symbolic) transformations through the model to identify erroneous sections.
- Dynamic debugging identifies errors that cause simulation to produce unexpected results. These techniques involve interactive inspection of models during execution to bookkeep changing variable values over simulated time. They must be used after static debugging techniques to ensure models can be compiled to executable code.

The rest of this annex gives an overview of these debugging procedures applied to the vehicular cruise control system example from Subannex A.6.

B.2 Preprocessing: Simplifying Models

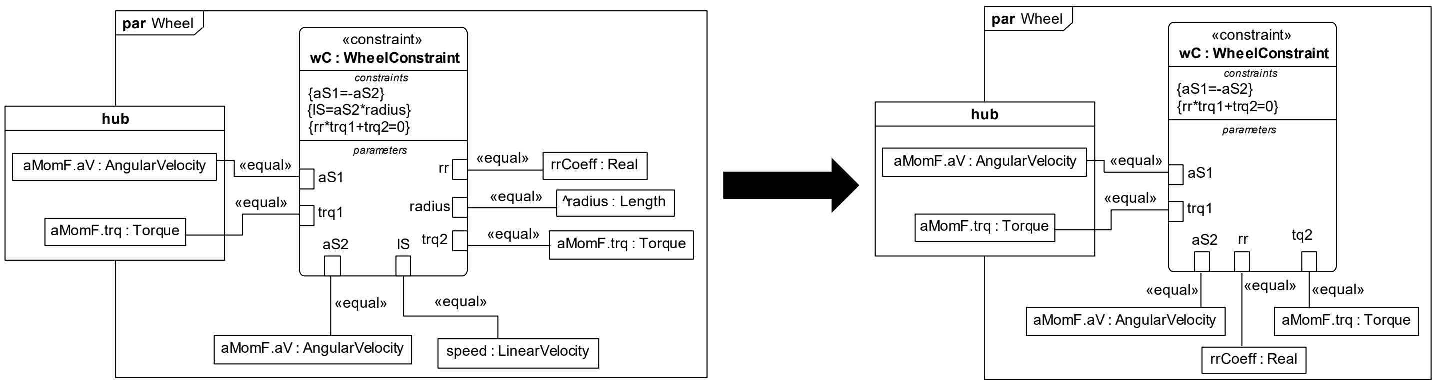
If a simulation model fails to compile or execute correctly, the cause can be identified by tracing through chains of connectors between components. This is the basis for static debugging techniques and facilitates dynamic debugging. It simplifies debugging to move physical interaction and signal flow connectors into separate models. The two simpler models can be debugged separately before replicating the resulting fixes in the complete model, a simpler task than debugging the entire model all at once.

First, create a model of physical interactions only by removing all connectors in the original model's internal block diagram (IBD) that do not represent physical interactions (saving a separate copy of the original model first). Any remaining parts or ports that are not at the end of the remaining connectors or do not possess a port that is at the end of a ~~remaining connector are also removed. Figure 123 shows internal structure from the cruise control system example (Figure 104: Internal structure of the cruise control system in Subannex A.6) with only its physical interactions.~~


ibd CruiseControlPhysicalInteractions
controlledVehicle : Car
initial values
crossSectionalArea = 2.0{unit = squaredMeter}
dragCoefficient = 1.0
mass = 2000.0{unit = kilogram}
«connector»
airVehicleLink : LMomentumTransfer
operatingEnvironment : Earth
initial values
acceleration = 9.8 {unit = meterPerSecondSquared}
atmosphere : Air
initial values
density = 1.2 {unit = kilogramPerCubicMeter}
LinearMomentum
LinearMomentum
gravVehicleLink : LMomPotEngTransformation
powerSource : Engine
initial values
trqCoef = 1.0{unit = newtonMeter}
throttleSetting = 0.8
crankshaft
«connector»
impellerVehicleLink : ALMomentumTransformation
lMTG
Momentum
surface : Road
initial values
flat = 0.0
rise = 0.1
start = 50.0{unit = second}
stop = 70.0{unit = second}
impeller : Wheel
initial values
rrCoeff = 0.5
radius = 0.5{unit=meter}
hub
AngularMomentum

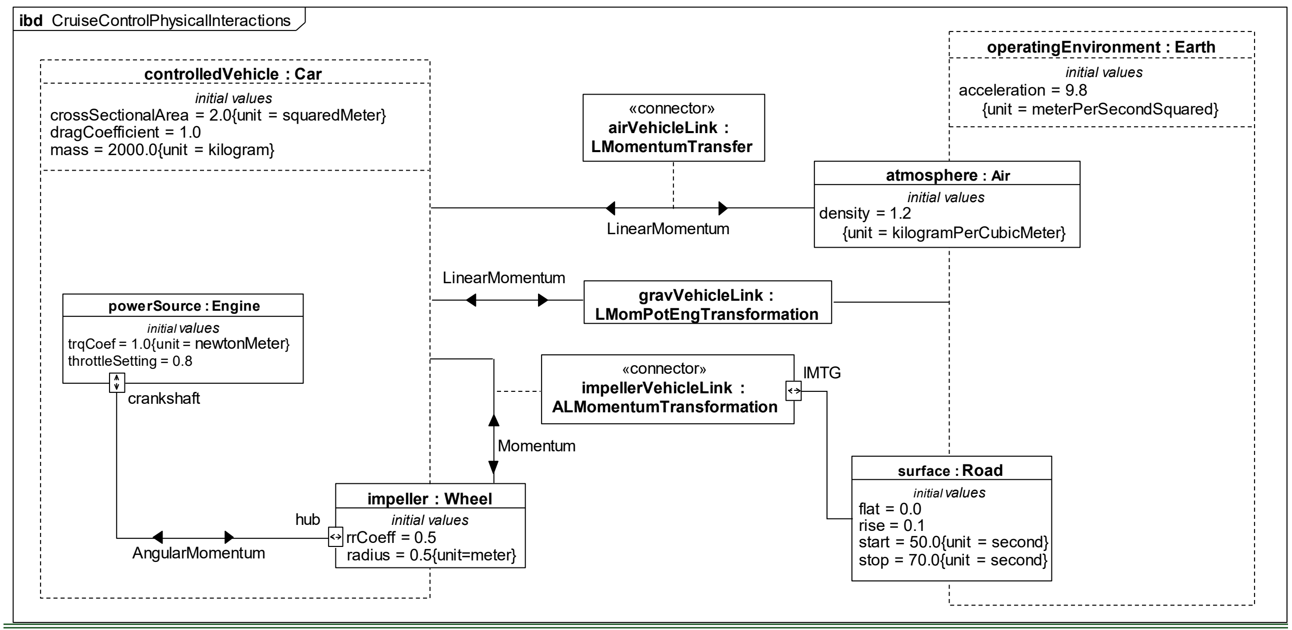
~~**Figure 123: Cruise control total system model with only physical interactions**~~

~~Next, in the parametric diagrams for the remaining parts or ports, remove equations (constraints) determining values of variables (constraint parameters) that are bound to (signal flow) *out*-flow properties (see Clause 7 for discussion on flow properties for signal flows and physical interactions). Remove part or port properties that are bound to variables on these *out*-flow equations as well. Replace any remaining equation variables bound to *in*-flow properties on the parts or ports by constant values, either by directly replacing the parameter with a constant value in constraints or by introducing a binding to a *PhSConstant*-stereotyped property that has a constant default value or instance value (see Subclause 10.10.2 for value assignment examples). Figure 124 depicts a parametric diagram for a component in Figure 123, before and after these changes were made for a physical interactions-only model.~~



~~***Figure 124: Show two (2) parametric diagrams of the same component (before and after changes for the physical interactions-only model)***~~

~~A separate system model for signal flows is created by first removing all connectors in the original model's IBD that do not represent signal flows (while saving a separate copy of the original model). Also remove any remaining parts or ports that are not at the end of the remaining connectors or does not possess a port that is at the end of a connector. Figure 125 shows an IBD with only the signal flows in the original cruise control total system model.~~ remaining connector are also removed. Figure 123 shows internal structure from the cruise control system example (Figure 104: Internal structure of the cruise control system in Subannex A.6) with only its physical interactions.



ibd CruiseControlPhysicalInteractions
controlledVehicle : Car
initial values
crossSectionalArea = 2.0{unit = squaredMeter}
dragCoefficient = 1.0
mass = 2000.0{unit = kilogram}
«connector»
airVehicleLink : LMomentumTransfer
operatingEnvironment : Earth
initial values
acceleration = 9.8 {unit = meterPerSecondSquared}
atmosphere : Air
initial values
density = 1.2 {unit = kilogramPerCubicMeter}
LinearMomentum
LinearMomentum
gravVehicleLink : LMomPotEngTransformation
powerSource : Engine
initial values
trqCoef = 1.0{unit = newtonMeter}
throttleSetting = 0.8
crankshaft
«connector»
impellerVehicleLink : ALMomentumTransformation
lMTG
Momentum
surface : Road
initial values
flat = 0.0
rise = 0.1
start = 50.0{unit = second}
stop = 70.0{unit = second}
impeller : Wheel
initial values
rrCoeff = 0.5
radius = 0.5{unit=meter}
hub
AngularMomentum

Figure 123: Cruise control total system model with only physical interactions

Next, in the parametric diagrams for the remaining parts or ports, remove equations (constraints) determining values of variables (constraint parameters) that are bound to (signal flow) *out*-flow properties (see Clause 7 for discussion on flow properties for signal flows and physical interactions). Remove part or port properties that are bound to variables on these *out*-flow equations as well. Replace any remaining equation variables bound to *in*-flow properties on the parts or ports by constant values, either by directly replacing the parameter with a constant value in constraints or by introducing a binding to a *PhSConstant*-stereotyped property that has a constant default value or instance value (see Subclause 10.10.2 for value assignment examples). Figure 124 depicts a parametric diagram for a component in Figure 123, before and after these changes were made for a physical interactions-only model.

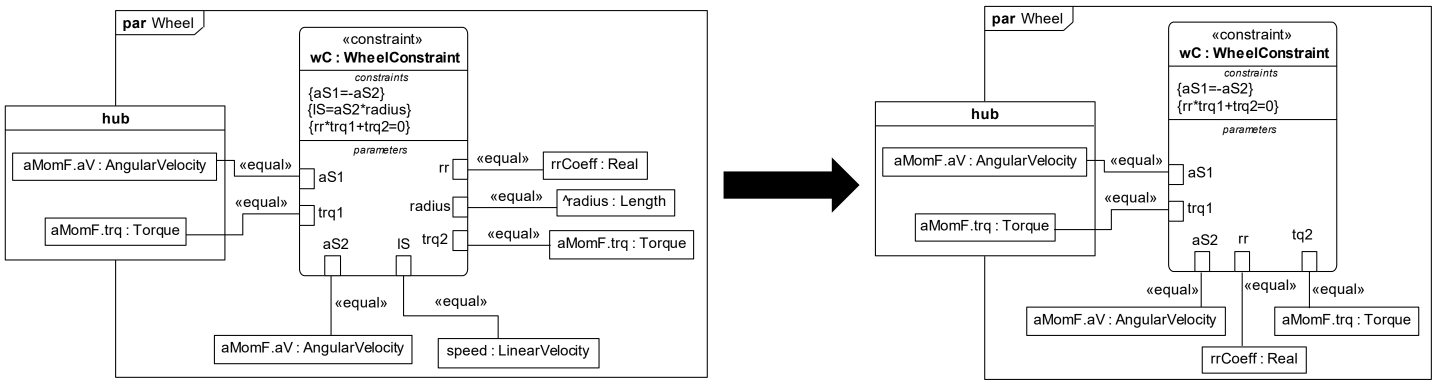
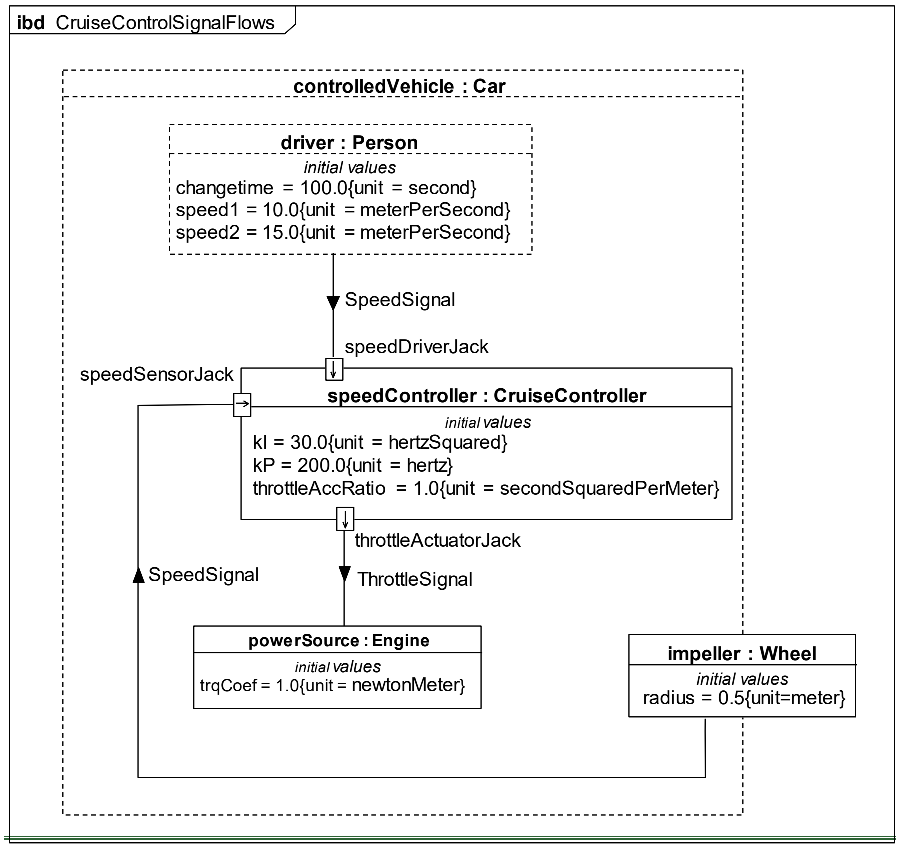


Figure 124: Show two (2) parametric diagrams of the same component (before and after changes for the physical interactions-only model)

A separate system model for signal flows is created by first removing all connectors in the original model's IBD that do not represent signal flows (while saving a separate copy of the original model). Also remove any remaining parts or ports that are not at the end of the remaining connectors or does not possess a port that is at the end of a connector. Figure 125 shows an IBD with only the signal flows in the original cruise control total system model.

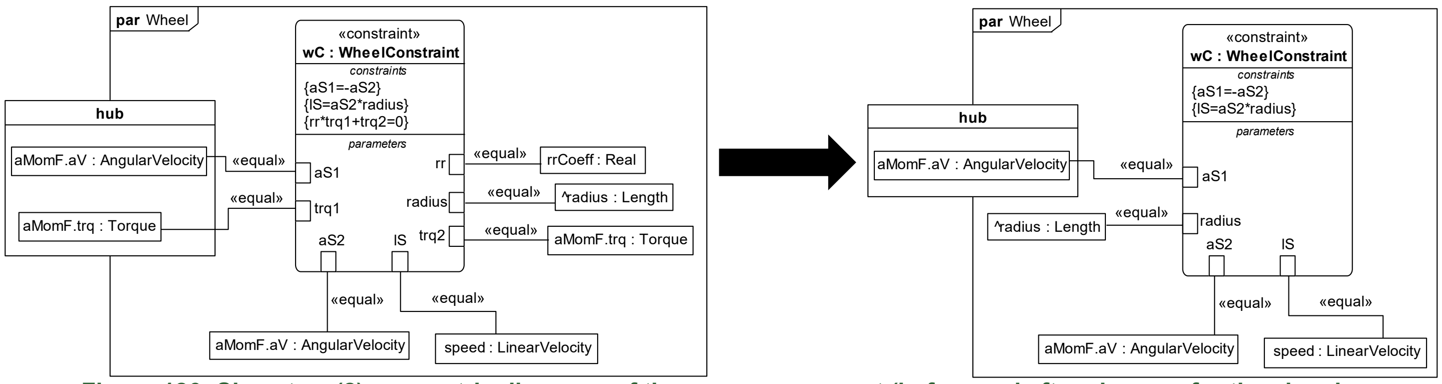


ibd CruiseControlSignalFlows
controlledVehicle : Car
driver : Person
initial values
changetime = 100.0{unit = second}
speed1 = 10.0{unit = meterPerSecond}
speed2 = 15.0{unit = meterPerSecond}
SpeedSignal
speedDriverJack
speedSensorJack
speedController : CruiseController
initial values
kI = 30.0{unit = hertzSquared}
kP = 200.0{unit = hertz}
throttleAccRatio = 1.0{unit = secondSquaredPerMeter}
throttleActuatorJack
SpeedSignal
ThrottleSignal
powerSource : Engine
initial values
trqCoef = 1.0{unit = newtonMeter}
impeller : Wheel
initial values
radius = 0.5{unit=meter}

~~**Figure 125: Cruise control total system model with only signal flows**~~

~~Next, in each parametric diagram for the remaining parts or ports, remove equations that play no role in determining values of variables bound to *out*-flow properties (see Clause 7 for discussion on flow properties for signal flows and physical interactions) or equations that do not have any bindings to *in*-flow properties. Remove part or port properties not bound to variables on the remaining equations. Of the remaining equations, some variables might be bound to physical interaction *inout*-flow properties on the parts or ports. These flow properties are replaced during simplification. If any equation variable bound to these flow properties determine the value of a variable bound to an *out*-flow property, then remove the *inout*-flow property and give a new constant value to its variable by binding to a *PhSConstant*-stereotyped property that has a constant default value or instance value (see Subclause 10.10.2 for value assignment examples). If any equation variable bound to these flow properties is determined by a variable in the same equation that is bound to an *in*-flow property, then remove the *inout*-flow property and give its variable a new binding to a new property with a *PhSVariable* (see Subclause 10.6.2 on applying variable- and constant-value stereotypes to properties).~~

~~Figure 126 depicts a parametric diagram for a component in Figure 123, before and after these changes were made for a signal flows-only model.~~



~~**Figure 126: Show two (2) parametric diagrams of the same component (before and after changes for the signal flows-only model)**~~

~~The remaining sections present the debugging techniques. Static techniques find causes of failure to compile and simulate translated models. This type of failure prevents generating a simulation run-time from the translated model. Once compilation succeeds, dynamic debugging techniques identify causes of failure to produce intended simulation behavior. The underlying theme for static debugging is tracing symbolic transformations in the model to find errors. Transformation tracing is also useful for dynamic debugging to better understand the model and sources of potential simulation-related errors.~~

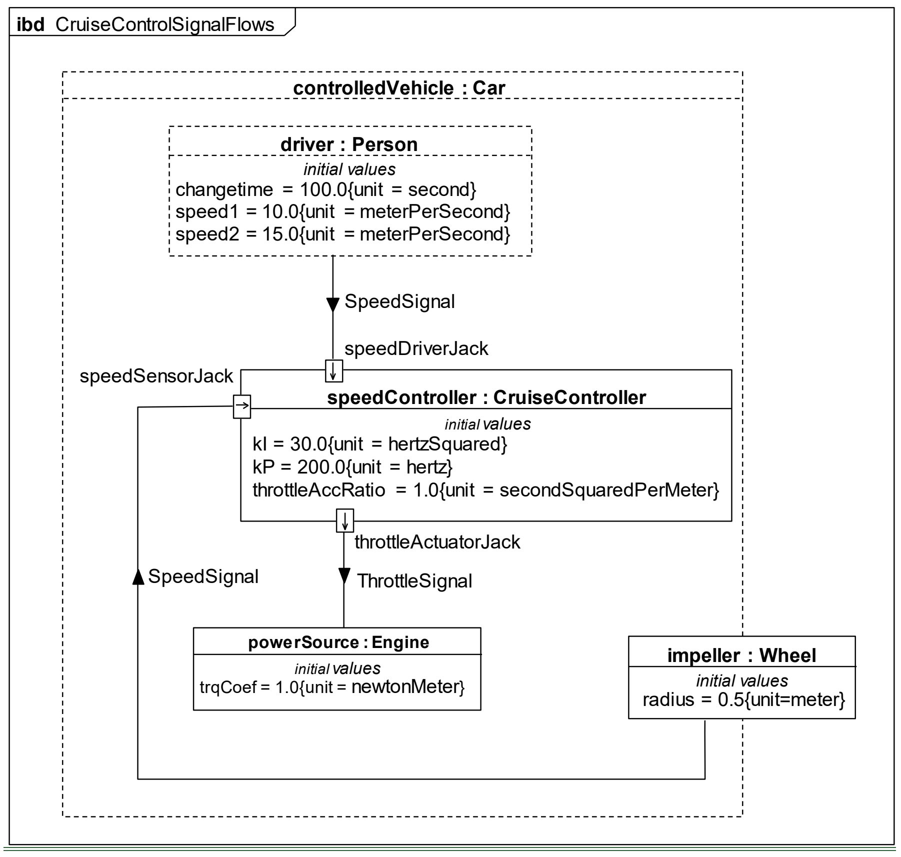

ibd CruiseControlSignalFlows
controlledVehicle : Car
driver : Person
initial values
changetime = 100.0{unit = second}
speed1 = 10.0{unit = meterPerSecond}
speed2 = 15.0{unit = meterPerSecond}
SpeedSignal
speedDriverJack
speedSensorJack
speedController : CruiseController
initial values
kI = 30.0{unit = hertzSquared}
kP = 200.0{unit = hertz}
throttleAccRatio = 1.0{unit = secondSquaredPerMeter}
throttleActuatorJack
SpeedSignal
ThrottleSignal
powerSource : Engine
initial values
trqCoef = 1.0{unit = newtonMeter}
impeller : Wheel
initial values
radius = 0.5{unit=meter}

Figure 125: Cruise control total system model with only signal flows

Next, in each parametric diagram for the remaining parts or ports, remove equations that play no role in determining values of variables bound to *out*-flow properties (see Clause 7 for discussion on flow properties for signal flows and physical interactions) or equations that do not have any bindings to *in*-flow properties. Remove part or port properties not bound to variables on the remaining equations. Of the remaining equations, some variables might be bound to physical interaction *inout*-flow properties on the parts or ports. These flow properties are replaced during simplification. If any equation variable bound to these flow properties determine the value of a variable bound to an *out*-flow property, then remove the *inout*-flow property and give a new constant value to its variable by binding to a *PhSConstant*-stereotyped property that has a constant default value or instance value (see Subclause 10.10.2 for value assignment examples). If any equation variable bound to these flow properties is determined by a variable in the same equation that is bound to an *in*-flow property, then remove the *inout*-flow property and give its variable a new binding to a new property with a *PhSVariable* (see Subclause 10.6.2 on applying variable- and constant-value stereotypes to properties).

Figure 126 depicts a parametric diagram for a component in Figure 123, before and after these changes were made for a signal flows-only model.

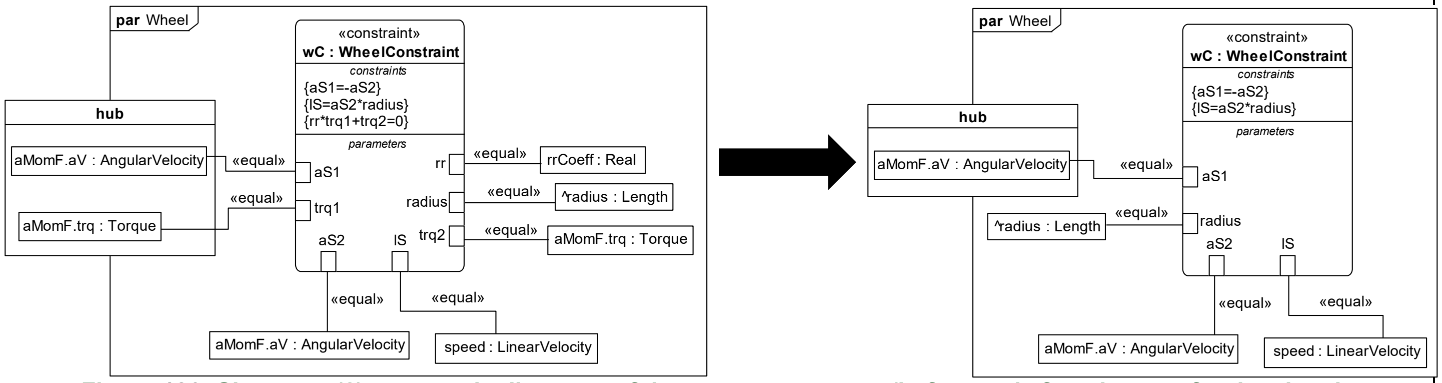


Figure 126: Show two (2) parametric diagrams of the same component (before and after changes for the signal flows-only model)

The remaining sections present the debugging techniques. Static techniques find causes of failure to compile and simulate translated models. This type of failure prevents generating a simulation run-time from the translated model. Once compilation succeeds, dynamic debugging techniques identify causes of failure to produce intended simulation behavior. The underlying theme for static debugging is tracing symbolic transformations in the model to find errors.

Transformation tracing is also useful for dynamic debugging to better understand the model and sources of potential simulation-related errors.

~~B.3 Static Debugging for Failure-to-Execute Simulation~~

~~The failure of a simulation model (translated from a system model) to compile and execute on a simulation platform indicates a static error. These errors can be identified with debugging techniques applied to the system model without translating and simulating it (statically). These techniques trace chains of symbolic transformations in the model, which appear in SysML as mathematical relationships in constraint equations (in parametrics diagrams) or implied by connectors (in IBDs). Specifically, tracing refers to tracking transformations of known and unknown variables through a model. Known variables are properties whose values are assigned a constant value or determined through mathematical relationships. Tracing is complemented by bookkeeping, which records the known or unknown status of these variables when operations apply to them in the model.~~

~~Static debugging can be performed on complete system models, but is described here on simplified, complementary models of a system's physical interactions and signal flows. For models with physical interactions, the first task is to identify the part, port, or connector property in IBDs where physical interaction will first occur or initiate other physical interactions in the system. Multiple parts and ports where physical interactions simultaneously occur can initiate further interactions, but any one can be arbitrarily picked to begin tracing. Tracing and bookkeeping of mathematical transformations start with properties associated to this selected part or port. Deciding which system component commences the physical interactions is easy in many cases. For example, the initiators of flow of electric charge in an electric circuit are the voltage sources or current sources. In the cruise control system represented in IBDs in Figure 104 and Figure 123, the throttle in the engine physically initiates the car's interaction with the road and air (this happens on command from the driver, but the command is signal flow, not physical interaction).~~

~~When the initiator of physical interaction is not obvious, it can help to inspect the parametric diagrams of parts or ports in IBDs. Parametric diagrams contain bindings between properties of the parts (or ports) to variables in the part's constraint equations. Look for parametric diagrams of parts that have a higher number of *PhSConstant*-stereotyped properties (with values given explicitly in the model) than *PhSVariable*-stereotyped properties (with values determined by mathematical relationships in the model), except for *PhSVariables* that give simulation time. The equation variables (constraint parameters) bound to *PhSConstant* or time properties are used in the part's equations (constraints) to determine values of other variables, which are bound to other properties used in the part's equations. To find an initiator, search for a part or port where most of its properties or properties of its ports are bound to constants or time values in its parametric diagram. The only properties without constant or time values should be flow properties, which can only have their values determined through connectors. Parts or ports initiating physical interactions have the fewest of these flow properties.~~

~~Tracing bindings and constraints in parametric diagrams helps understand and keep track of (bookkeep) which variables in the equations are known and unknown. Constraint equations show mathematical transformations between known variables, bound to properties with known values, and unknown variables, bound to properties with unknown values. Before simulation, the only known variables are the ones bound to *PhSConstant* properties, the variables bound to properties given (initial) values at the start of simulation, and properties that give simulation time values. These should lead to values assigned to all variables in the parametrics diagram of physical interaction-initiating parts. The status of these variables will change as tracing shows their values being assigned through constraints or connectors, which is recorded by bookkeeping.~~

~~Physical interaction flow properties on the current part in the debugging process link to flow properties on parts or ports at the other end of the linking connectors. Trace along these connectors to find out whether values are assigned to these flow properties leads to parametric diagrams of other parts, ports, and connector properties linked to the current part. Repeat the same methods of tracing and bookkeeping in these other parametric diagrams to determine whether values are assigned to unknown variables and to find flow properties that lead to new connectors and parametric diagrams. The trace must go through all connectors and parametric diagrams of the system's parts, ports, and connector properties. Figure 127 shows an example of tracing and bookkeeping value assignments between the vehicle's engine, the physical interaction-initiating part of the cruise control system, and the rest of the physical interactions-only system IBD from Figure 123. Bookkeeping of the total trace completes the tracking of value assignments. The bookkeepings of variables for Figure 127are depicted in the tables that follow the figure. The traces in the bookkeepings correspond to the marked points in the figure (A, B, C, D, E, & F).~~

~~A system model will compile and simulate when translated if it: a) uses all the constraint equations and connectors in the model for mathematical transformations between known and unknown variables and b) has all its property values determined by simulation of mathematical transformations. If tracing and bookkeeping identifies a constraint equation or connector that is not used, the system is overconstrained. In this scenario, the modeler must choose whether unused equations or connectors should be removed or a new property should be included and related to them. If an unknown property is not defined by any mathematical constraint or connector, then the system is underconstrained. In this scenario, the modeler must choose between using this property in a new equation or removing the property. Tracing and~~

~~bookkeeping of equations also helps spot constraint equations that involve a division by zero and functions called outside their domains. Once corrections to the model are made, they are replicated in the original system model.~~

B.3 Static Debugging for Failure-to-Execute Simulation

The failure of a simulation model (translated from a system model) to compile and execute on a simulation platform indicates a static error. These errors can be identified with debugging techniques applied to the system model without translating and simulating it (statically). These techniques trace chains of symbolic transformations in the model, which appear in SysML as mathematical relationships in constraint equations (in parametrics diagrams) or implied by connectors (in IBDs). Specifically, tracing refers to tracking transformations of known and unknown variables through a model. Known variables are properties whose values are assigned a constant value or determined through mathematical relationships. Tracing is complemented by bookkeeping, which records the known or unknown status of these variables when operations apply to them in the model.

Static debugging can be performed on complete system models, but is described here on simplified, complementary models of a system's physical interactions and signal flows. For models with physical interactions, the first task is to identify the part, port, or connector property in IBDs where physical interaction will first occur or initiate other physical interactions in the system. Multiple parts and ports where physical interactions simultaneously occur can initiate further interactions, but any one can be arbitrarily picked to begin tracing. Tracing and bookkeeping of mathematical transformations start with properties associated to this selected part or port. Deciding which system component commences the physical interactions is easy in many cases. For example, the initiators of flow of electric charge in an electric circuit are the voltage sources or current sources. In the cruise control system represented in IBDs in Figure 104 and Figure 123, the throttle in the engine physically initiates the car's interaction with the road and air (this happens on command from the driver, but the command is signal flow, not physical interaction).

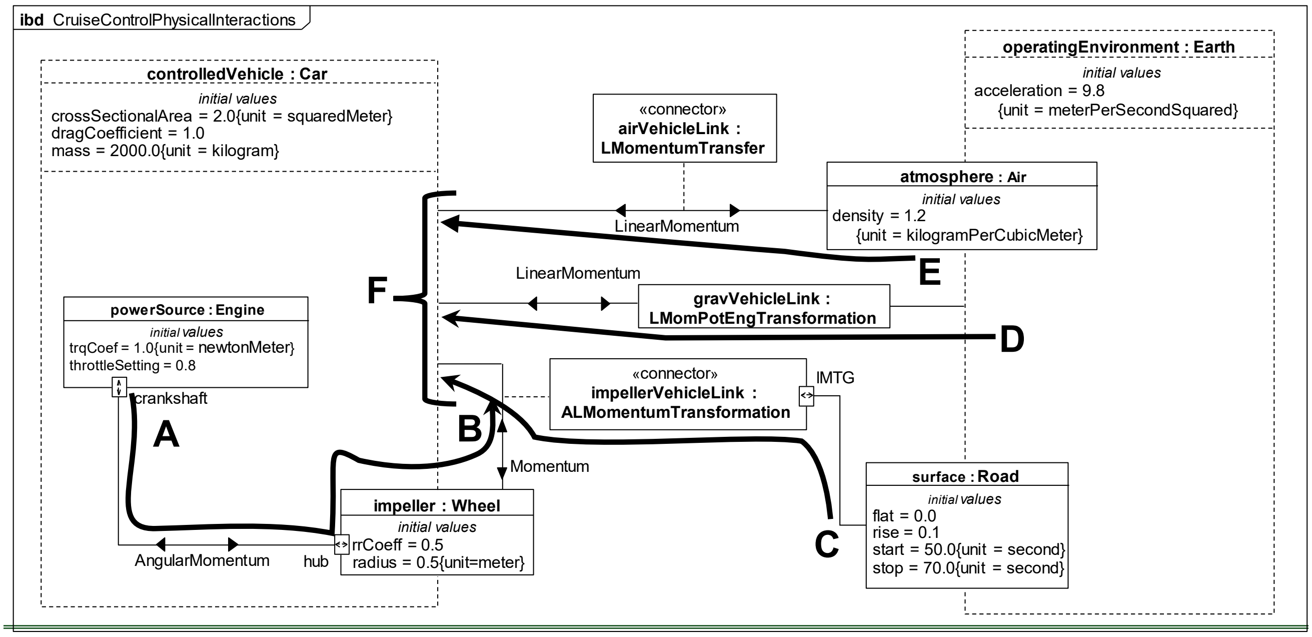
When the initiator of physical interaction is not obvious, it can help to inspect the parametric diagrams of parts or ports in IBDs. Parametric diagrams contain bindings between properties of the parts (or ports) to variables in the part's constraint equations. Look for parametric diagrams of parts that have a higher number of *PhSConstant*-stereotyped properties (with values given explicitly in the model) than *PhSVariable*-stereotyped properties (with values determined by mathematical relationships in the model), except for *PhSVariables* that give simulation time. The equation variables (constraint parameters) bound to *PhSConstant* or time properties are used in the part's equations (constraints) to determine values of other variables, which are bound to other properties used in the part's equations. To find an initiator, search for a part or port where most of its properties or properties of its ports are bound to constants or time values in its parametric diagram. The only properties without constant or time values should be flow properties, which can only have their values determined through connectors. Parts or ports initiating physical interactions have the fewest of these flow properties.

Tracing bindings and constraints in parametric diagrams helps understand and keep track of (bookkeep) which variables in the equations are known and unknown. Constraint equations show mathematical transformations between known variables, bound to properties with known values, and unknown variables, bound to properties with unknown values. Before simulation, the only known variables are the ones bound to *PhSConstant* properties, the variables bound to properties given (initial) values at the start of simulation, and properties that give simulation time values. These should lead to values assigned to all variables in the parametrics diagram of physical interaction-initiating parts. The status of these variables will change as tracing shows their values being assigned through constraints or connectors, which is recorded by bookkeeping.

Physical interaction flow properties on the current part in the debugging process link to flow properties on parts or ports at the other end of the linking connectors. Trace along these connectors to find out whether values are assigned to these flow properties leads to parametric diagrams of other parts, ports, and connector properties linked to the current part. Repeat the same methods of tracing and bookkeeping in these other parametric diagrams to determine whether values are assigned to unknown variables and to find flow properties that lead to new connectors and parametric diagrams. The trace must go through all connectors and parametric diagrams of the system's parts, ports, and connector properties. Figure 127 shows an example of tracing and bookkeeping value assignments between the vehicle's engine, the physical interaction-initiating part of the cruise control system, and the rest of the physical interactions-only system IBD from Figure 123. Bookkeeping of the total trace completes the tracking of value assignments. The bookkeepings of variables for Figure 127are depicted in the tables that follow the figure. The traces in the bookkeepings correspond to the marked points in the figure (A, B, C, D, E, & F).

A system model will compile and simulate when translated if it: a) uses all the constraint equations and connectors in the model for mathematical transformations between known and unknown variables and b) has all its property values determined by simulation of mathematical transformations. If tracing and bookkeeping identifies a constraint equation or connector that is not used, the system is overconstrained. In this scenario, the modeler must choose whether unused equations or connectors should be removed or a new property should be included and related to them. If an unknown property is not defined by any mathematical constraint or connector, then the system is underconstrained. In this scenario, the modeler must choose between using this property in a new equation or removing the property. Tracing and bookkeeping of equations also helps spot constraint equations that involve a division by zero and functions called outside their domains. Once corrections to the model are made, they are replicated in the original system model.

~~If there is a complementary model of signal flows, repeat the process of tracing and bookkeeping in a similar fashion, but start tracing from all parts that do not have *in*-flow properties or do not own ports that have *in*-flow properties. The *in*-flow property on these parts indicate that they receive unidirectional signals from another part in the model, so they cannot be the initiator of signal flows. Corrections in this model should likewise (the physical interactions model) be reproduced in the original, complete model of the system. Translate the corrected SysML model and test on simulation platforms to determine if more debugging is needed.~~



~~**Figure 127: Shows initiating physical interaction component (at point A), direction of traces, bookkeeping of variables, and value assignment that occur through the total trace (ending at F)**~~

Bookkeeping of variables through parts and ports from A to B									
power-Source	**Value known?**	**crankshaft**	**Value known?**	**hub**	**Value known?**	**impeller**	**Value known?**	**impeller-VehicleLink**	**Value known?**
trqCoeff	☒	torque	☒	torque	☒	torque	☒	torque	☒
throttle-Setting	☒	angular velocity	☒	angular velocity	☒	angular velocity	☒	angular velocity	☐
						rrCoeff	☒	force	☒
						radius	☒	linear velocity	☐
								radius	☒
								ground-force	☐
								ground-linear velocity	☐

If there is a complementary model of signal flows, repeat the process of tracing and bookkeeping in a similar fashion, but start tracing from all parts that do not have *in*-flow properties or do not own ports that have *in*-flow properties. The *in*-flow property on these parts indicate that they receive unidirectional signals from another part in the model, so they cannot be the initiator of signal flows. Corrections in this model should likewise (the physical interactions model) be reproduced in the original, complete model of the system. Translate the corrected SysML model and test on simulation platforms to determine if more debugging is needed.

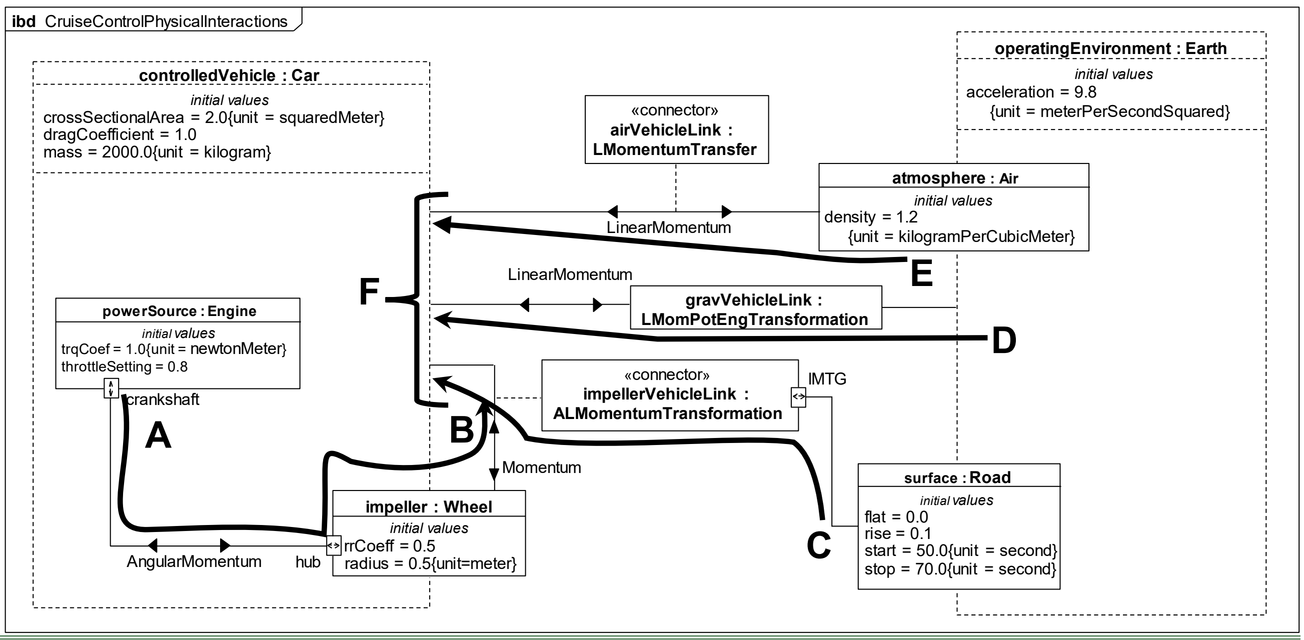


Figure 127: Shows initiating physical interaction component (at point A), direction of traces, bookkeeping of variables, and value assignment that occur through the total trace (ending at F)

Bookkeeping of variables through parts and ports from A to B

power-Source	Value known?	crankshaft	Value known?	hub	Value known?	impeller	Value known?	impeller-VehicleLink	Value known?
trqCoeff	☒	torque	☒	torque	☒	torque	☒	torque	☒
throttle-Setting	☒	angular velocity	☒	angular velocity	☒	angular velocity	☒	angular velocity	☐
						rrCoeff	☒	force	☒
						radius	☒	linear velocity	☐
								radius	☒
								ground-force	☐
								ground-linear velocity	☐

Bookkeeping of variables through parts and ports from C to B to F

surface	Value known?	lMTG (ground)	Value known?	impeller-VehicleLink	Value known?	controlled-Vehicle	Value known?
linear velocity	☒	force	☒	torque	☒	mass	☒
slope	☒	linear velocity	☒	angular velocity	☒	force	☒
				force	☒	linear velocity	☒
				linear velocity	☒		
				radius	☒		
				ground-force	☒		
				ground-linear velocity	☒		

Bookkeeping of variables through parts and ports from D to F

Operating-Environment	Value known?	grav-VehicleLink	Value known?	controlled-Vehicle	Value known?
acceleration	☒	slope	☒	mass	☒
		acceleration	☒	force	☒
		mass	☒	linear velocity	☒
		force	☒		

Bookkeeping of variables through parts and ports from E to F

air	Value known?	air-VehicleLink	Value known?	controlled-Vehicle	Value known?
linear velocity	☒	density	☒	mass	☒
		cross-sectional Area	☒	force	☒
		dragCoeff	☒	linear velocity	☒
		fluid-linear velocity	☒		
		fluid-force	☒		
		velocity	☒		
		solid-linear velocity	☒		
		solid-force	☒		

Bookkeeping of variables through parts and ports from C to B to F							
surface	**Value known?**	**lMTG (ground)**	**Value known?**	**impeller-VehicleLink**	**Value known?**	**controlled-Vehicle**	**Value known?**
linear velocity	☒	force	☒	torque	☒	mass	☒
slope	☒	linear velocity	☒	angular velocity	☒	force	☒
				force	☒	linear velocity	☒
				linear velocity	☒		
				radius	☒		
				ground-force	☒		
				ground-linear velocity	☒		

Bookkeeping of variables through parts and ports from D to F					
Operating-Environment	**Value known?**	**grav-VehicleLink**	**Value known?**	**controlled-Vehicle**	**Value known?**
acceleration	☒	slope	☒	mass	☒
		acceleration	☒	force	☒
		mass	☒	linear velocity	☒
		force	☒		

Bookkeeping of variables through parts and ports from E to F					
air	**Value known?**	**air-VehicleLink**	**Value known?**	**controlled-Vehicle**	**Value known?**
linear velocity	☒	density	☒	mass	☒
		cross-sectional Area	☒	force	☒
		dragCoeff	☒	linear velocity	☒
		fluid-linear velocity	☒		
		fluid-force	☒		
		velocity	☒		
		solid-linear velocity	☒		
		solid-force	☒		

B.4 Dynamic Debugging for Unexpected Simulation Results

~~Failure of a simulation model (translated from a system model) to produce expected results when executed indicates a dynamic error. The simulation model is able to compile and simulate but produces variable values that deviate from modeler expectations. These errors can be identified with dynamic debugging techniques applied to the system model. These techniques examine executed simulations to understand exactly when signals and conserved substances flow through the system and what their characteristics are.~~

~~They focus on simulation results for variables involved in the static traces of flow properties linked by connectors in the previous section. This showed how variables characterizing flow of physical substances and signals during simulation are related via transformations in the system model (mathematical operations via constraint equations and connectors). Though dynamic debugging can be performed without prior static debugging, fixing static errors first ensures the simulation model will compile and execute, and static tracing improves understanding of how variables change during simulation.~~

~~Dynamic debugging can be performed on complete system models, but is described here on simplified, complementary models of a system's physical interactions and signal flows. Behavior of conserved substances in physical interactions is characterized by their flow rate and potential to flow. Flow rate and potential to flow appear in simulation as variables translated from properties at the ends of connectors in the system model. This enables modelers to track simulation variables that correspond to properties in SysML system models. The SysPhS translator uses the names of association ends and constraint parameters in the resulting simulation models to facilitate this but tracking simulation variables might require some familiarity with the simulation language. Lastly, like static debugging, dynamic debugging starts by tracing simulation variable transformations at points in the model that initiate physical interactions in the rest of the model. These points must be identified before debugging.~~

~~Physical interaction variables simulate flow of conserved substances only at their corresponding connector endpoint (part or port) in the system model. A more complete picture of symbolic transformations of these variables is seen by observing their values over simulated time and comparing them to other physical interaction simulation variables in the model. Graphical displays in simulation tools show these values, enabling comparison of simulated values to their intended mathematical relationships. The relationships are defined, correctly or not, through transformations (mathematical relationships between variables derived from connectors and parametric diagrams in the system model) of corresponding flow properties in the system model. To visualize these transformations, observe variables when their corresponding flow properties have not undergone more than one set of transformations (operations that occur on flow properties in the constraints of one parametric diagram or in the mathematical relationship implied by one connector). Compare simulation values of these variables with those of other physical interaction variables related to the same part or port in the system model, as well as simulation variables related to the other end of the variables' associated connectors.~~

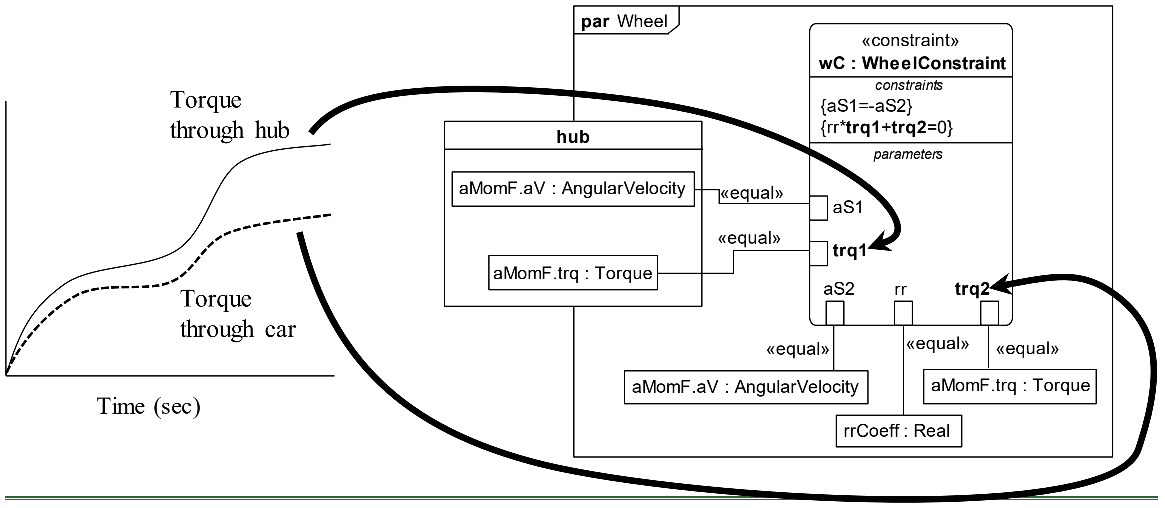
Failure of a simulation model (translated from a system model) to produce expected results when executed indicates a dynamic error. The simulation model is able to compile and simulate but produces variable values that deviate from modeler expectations. These errors can be identified with dynamic debugging techniques applied to the system model. These techniques examine executed simulations to understand exactly when signals and conserved substances flow through the system and what their characteristics are.

They focus on simulation results for variables involved in the static traces of flow properties linked by connectors in the previous section. This showed how variables characterizing flow of physical substances and signals during simulation are related via transformations in the system model (mathematical operations via constraint equations and connectors). Though dynamic debugging can be performed without prior static debugging, fixing static errors first ensures the simulation model will compile and execute, and static tracing improves understanding of how variables change during simulation.

Dynamic debugging can be performed on complete system models, but is described here on simplified, complementary models of a system's physical interactions and signal flows. Behavior of conserved substances in physical interactions is characterized by their flow rate and potential to flow. Flow rate and potential to flow appear in simulation as variables translated from properties at the ends of connectors in the system model. This enables modelers to track simulation variables that correspond to properties in SysML system models. The SysPhS translator uses the names of association ends and constraint parameters in the resulting simulation models to facilitate this but tracking simulation variables might require some familiarity with the simulation language. Lastly, like static debugging, dynamic debugging starts by tracing simulation variable transformations at points in the model that initiate physical interactions in the rest of the model. These points must be identified before debugging.

Physical interaction variables simulate flow of conserved substances only at their corresponding connector endpoint (part or port) in the system model. A more complete picture of symbolic transformations of these variables is seen by observing their values over simulated time and comparing them to other physical interaction simulation variables in the model. Graphical displays in simulation tools show these values, enabling comparison of simulated values to their intended mathematical relationships. The relationships are defined, correctly or not, through transformations (mathematical relationships between variables derived from connectors and parametric diagrams in the system model) of corresponding flow properties in the system model. To visualize these transformations, observe variables when their corresponding flow properties have not undergone more than one set of transformations (operations that occur on flow properties in the constraints of one parametric diagram or in the mathematical relationship implied by one connector). Compare simulation values of these variables with those of other physical interaction variables related to the same part or port in the system model, as well as simulation variables related to the other end of the variables' associated connectors.

~~Analysis of simulation variable results is performed in simulation runs that are sufficiently long for their values to reach a steady-state or a recognizable pattern of changes. Check that changes follow the mathematical transformations specified in corresponding constraint equations and connector links in the system model, which can be modified to produce better results. Figure 128 shows the relationship between simulated variable values over time and flow properties in the parametrics diagram (from Figure 124) for a component in the physical interactions-only system IBD (from Figure 123).~~



~~**Figure 128: Relationship between simulation variables and flow properties in the parametric diagrams for components in the system IBD**~~

~~Further simplification of system models can determine whether simulation results are valid, especially when physical interactions are highly complex. One way is to temporarily remove parts, ports, and connectors until modelers have high confidence in what they expect from variable behavior. Once this simpler model produces expected simulations, the removed parts, ports, and connectors can be restored and checked (via simulations) in the reverse order that which they were removed.~~

~~For a complementary model of signal flows, if there is one, repeat the process of inspecting simulation variables in a similar fashion. However, start tracing with all parts that do not have in-flow properties or do not own ports that have in-flow properties, as chosen during static debugging. Replace remaining parts in a complementary model of signal flows that only have out-flow properties or only have ports with out-flow properties have their flow properties by PhSConstant-stereotyped properties with pre-specified values before debugging.~~

~~Errors that are found by debugging are corrected in the system model, then tested by translating to simulation models and executing them. Translating and testing system models to multiple simulation platforms is more robust, because fixes sometimes work for one simulation platform and not others. For example, a function call in a parametric diagram is domain-specific, and this might need to be replaced with a more universal function call. It is also possible that some modeling capabilities in SysML, such as state machines or different ways of defining initial values, cannot be replicated on some simulation platforms (see Clause 10 for more specific examples about translation differences between simulation platforms).~~ Analysis of simulation variable results is performed in simulation runs that are sufficiently long for their values to reach a steady-state or a recognizable pattern of changes. Check that changes follow the mathematical transformations specified in corresponding constraint equations and connector links in the system model, which can be modified to produce better results. Figure 128 shows the relationship between simulated variable values over time and flow properties in the parametrics diagram (from Figure 124) for a component in the physical interactions-only system IBD (from Figure 123).

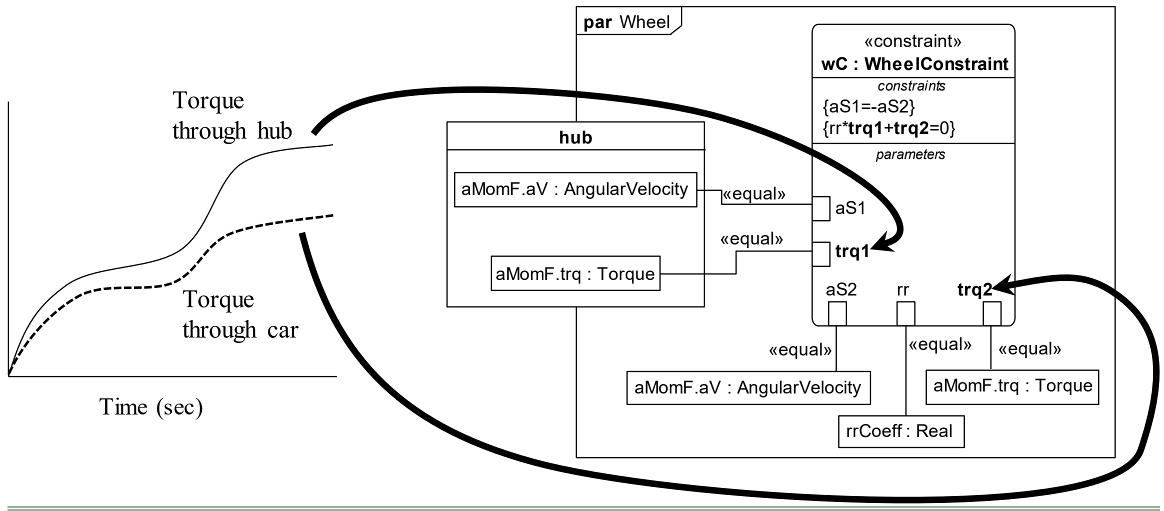


Figure 128: Relationship between simulation variables and flow properties in the parametric diagrams for components in the system IBD

Further simplification of system models can determine whether simulation results are valid, especially when physical interactions are highly complex. One way is to temporarily remove parts, ports, and connectors until modelers have high confidence in what they expect from variable behavior. Once this simpler model produces expected simulations, the removed parts, ports, and connectors can be restored and checked (via simulations) in the reverse order that which they were removed.

For a complementary model of signal flows, if there is one, repeat the process of inspecting simulation variables in a similar fashion. However, start tracing with all parts that do not have in-flow properties or do not own ports that have in-flow properties, as chosen during static debugging. Replace remaining parts in a complementary model of signal flows that only have out-flow properties or only have ports with out-flow properties have their flow properties by PhSConstant-stereotyped properties with pre-specified values before debugging.

Errors that are found by debugging are corrected in the system model, then tested by translating to simulation models and executing them. Translating and testing system models to multiple simulation platforms is more robust, because fixes sometimes work for one simulation platform and not others. For example, a function call in a parametric diagram is domain-specific, and this might need to be replaced with a more universal function call. It is also possible that some modeling capabilities in SysML, such as state machines or different ways of defining initial values, cannot be replicated on some simulation platforms (see Clause 10 for more specific examples about translation differences between simulation platforms).