An OMG Systems Modeling Language™ Publication

OMG Systems Modeling Language (OMG SysML™)

Version 1.6

OMG Document Number: formal/19-11-01

Date: November 2019

Standard document URL: https://www.omg.org/spec/SysML/1.6/

Machine Readable File(s):

Normative:

https://www.omg.org/spec/SysML/20181001/sysml.xmi

Non-normative:

https://www.omg.org/spec/SysML/20181001/sysmldi.xmi

https://www.omg.org/spec/SysML/20181001/qudv.xmi

https://www.omg.org/spec/SysML/20181001/iso800000.xmi

Refer to the Roadmap located in the Preface for a list of documents that were generated as part of the adoption, finalization, and revision process.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

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

GENERAL USE RESTRICTIONS

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

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING,

PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

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

TRADEMARKS

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

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

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG'S ISSUE REPORTING PROCEDURE

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

Table of Contents

Table of Figures

Table of Tables

Preface

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

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
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

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

Issues

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.

SysML Roadmap

Requirements for SysML were originally specified by:

ad/2003-03-41 (UML for Systems Engineering RFP)

The source documents for this specification include:

Alpha: ad/2006-03-01 (submission)

ad/2006-04-07 (errata)

ad/2006-03-04 (glossary)

Associated Schema files: ad/2006-03-02 (XMI)

The Finalization Task Force (FTF) process generated the following documents:

Beta 1: ptc/2006-05-04 (a.k.a. Final Adopted Specification)

Beta 2: ptc/2007-03-19 (FTF Report - full record of FTF votes and issue resolutions

ptc/2007-02-03, ptc/2007-03-04 (a.k.a. convenience document, with and without change bars)

ptc/2007-02-05 (XMI)

ptc/2007-03-09 (Annex E - Requirements Traceability)

Version 1.0 Formal Specification: formal/2007-09-01

The SysML 1.1 Revision Task Force (RTF) process generated the following documents:

ptc/2008-05-15 (RTF Report - full record of RTF votes and issue resolutions)

ptc/2008-05-16, ptc/2008-05-17 (a.k.a. convenience document, with and without change bars)

ptc/2008-05-18 (XMI)

Version 1.1 Formal Specification: formal/2008-11-01, formal/2008-11-02

Associated schema files for this specification, at https://www.omg.org/spec/SysML/20090501/, include the following files:

SysML-profile.xmi XMI 2.1 serialization of the SysML Profile

Activities-model.xmi XMI 2.1 serialization of the Activities model library

Blocks-model.xmi XMI 2.1 serialization of the Blocks model library

UML4SysML-metamodel.xmi XMI 2.1 serialization of the merged UML4SysML subset of UML 2 (used to define the SysML Profile)

The SysML 1.2 Revision Task Force (RTF) process generated the following documents:

ptc/2008-05-15 (RTF Report - full record of RTF votes and issue resolutions)

ptc/2008-05-16, ptc/2008-05-17 (a.k.a. convenience document, with and without change bars)

ptc/2008-05-18 (XMI)

Version 1.2 Formal Specification: formal/2010-06-01, formal/2010-06-02

Associated schema file for this specification, at https://www.omg.org/spec/SysML/20100301, include the following files:

SysML-profile.uml XMI 2.1 serialization of the SysML Profile

UML4SysML-metamodel.uml XMI 2.1 serialization of the merged UML4SysML subset of UML 2 (used to define the SysML Profile)

Activities-model.xmi XMI 2.1 serialization of the Activities model library

Blocks-model.xmi XMI 2.1 serialization of the Blocks model library

The SysML 1.3 Revision Task Force (RTF) process generated the following documents:

ptc/2011-08-08 (RTF Report - full record of RTF votes and issue resolutions)

ptc/2011-08-07 (Submission inventory document)

ptc/2011-08-09, ptc/2011-08-10 (Beta “convenience document,” with and without change bars)

ptc/2011-08-11, ptc/2011-08-12 (Normative and non-normative XMI)

ptc/2012-04-07, ptc/2012-04-08 (Normative and non-normative XMI)

Version 1.3 Formal Specification: formal/2012-06-01, formal/2012-06-02

Associated schema files for this specification, at https://www.omg.org/spec/SysML/20120401/, include the following files:

SysML.xmi (Normative)

ISO-80000-1-QUDV.xmi (Non-normative)

ISO-80000-1-SysML.xmi (Non-normative)

QUDV.xmi (Non-normative)

The SysML 1.4 Revision Task Force (RTF) process generated the following documents:

ptc/2013-12-08 (RTF Report - full record of RTF votes and issue resolutions)

ptc/2013-12-10, ptc/2013-12-09 (Beta “convenience document,” with and without change bars)

ptc/2013-12-11, ptc/2013-12-12 (Normative and non-normative XMI)

Version 1.4 Formal Specification: formal/2015-06-03, formal/2015-06-04

Associated schema files for this specification, at https://www.omg.org/spec/SysML/20131201/, include the following files:

SysML.xmi (Normative)

SysMLDI.xmi (Normative)

ISO-80000-1-QUDV.xmi (Non-normative)

ISO-80000-1-SysML.xmi (Non-normative)

QUDV.xmi (Non-normative)

The SysML 1.5 Revision Task Force (RTF) process generated the following documents:

ptc/2016-11-01 (RTF Report - full record of RTF votes and issue resolutions)

ptc/2016-11-02, ptc/2016-11-03 (Beta "convenience document," with and without change bars)

ptc/2016-11-05, ptc/2016-11-06, ptc/16-11-07, ptc/16-11-08 (Normative and non-normative XMI)

Associated schema files for this specification, at https://www.omg.org/spec/SysML/20161101/, include the following files:

SysML.xmi (Normative)

SysMLDI.xmi (Normative)

ISO-80000-1-QUDV.xmi (Non-normative)

ISO-80000-1-SysML.xmi (Non-normative)

QUDV.xmi (Non-normative)

The SysML 1.6 Revision Task Force (RTF) process generated the following documents:

ptc/2018-10-01 (RTF Report - full record of RTF votes and issue resolutions)

ptc/2018-10-02, ptc/2018-10-03 (Beta "convenience document," with and without change bars)

ptc/2018-10-04, ptc/2018-10-05, ptc/2018-10-06, ptc/2018-10-07, ptc/2018-10-08 (Normative and non-normative XMI)

Associated schema files for this specification, at https://www.omg.org/spec/SysML/20161101/, include the following files:

SysML.xmi (Normative)

ISO-80000-1-QUDV.xmi (Non-normative)

ISO-80000-1-SysML.xmi (Non-normative)

QUDV.xmi (Non-normative)

This page intentionally left blank.

1 Scope

The purpose of this International Standard is to specify the Systems Modeling Language (SysML), a general-purpose modeling language for systems engineering. Its intent is to specify the language so that systems engineering modelers may learn to apply and use SysML; modeling tool vendors may implement and support SysML; and both can provide feedback to improve future versions. Note that a definition of “system” and “systems engineering” can be found inISO/IEC 15288.

SysML reuses a subset of UML 2.5 and provides additional extensions to address the requirements in UML for SE. SysML uses the UML 2.5 extension mechanisms as further elaborated in Clause 17 as the primary mechanism to specify the extensions to UML 2.5. This revision of SysML relies on several new features incorporated into UML 2.5. Any use of the term “UML 2” or “UML” in this specification, unless otherwise noted, will refer to UML 2.5 in general and the UML 2.5 specification in particular.

Since SysML uses UML 2.5 as its foundation, systems engineers modeling with SysML and software engineers modeling with UML 2.5 will be able to collaborate on models of software-intensive systems. This will improve communication among the various stakeholders who participate in the systems development process and promote interoperability among modeling tools. It is anticipated that SysML will be customized to model domain-specific applications, such as automotive, aerospace, communication, and information systems.

SysML is designed to provide simple but powerful constructs for modeling a wide range of systems engineering problems. It is particularly effective in specifying requirements, structure, behavior, allocations, and constraints on system properties to support engineering analysis. The language is intended to support multiple processes and methods such as structured, object-oriented, and others, but each methodology may impose additional constraints on how a construct or diagram kind may be used. This version of the language supports most, but not all, of the requirements of the UML for Systems Engineering RFP, as shown in the Requirements Traceability referenced by Annex F. These gaps are intended to be addressed in future versions of SysML as indicated in the matrix.

The following sub clauses provide background information about this International Standard. Instructions for both systems engineers and tool vendors who read this International Standard are provided in “How to Read this International Standard.” The main body of this International Standard describes the normative technical content. The annexes include additional information to aid in understanding and implementation of this International Standard.

2 Normative References

The following normative documents contain provisions, which through reference in this text, constitute provisions of this International Standard. Subsequent amendments to, or revisions of, any of these publications do not apply.

ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards, 7th Edition 2016

ISO/IEC 10303-233:2012, STEP AP233, Product data representation and exchange: application protocol: Systems engineering

ISO/IEC IEEE 15288:2015, Systems and software engineering - System life cycle process

OMG Specification formal/2017-12-01, Unified Modeling Language, (UML) V2.5.1

(https://www.omg.org/spec/UML/2.5.1/)

OMG Specification formal/2014-02-03, Object Constraint Language (OCL), V2.4

(https://www.omg.org/spec/OCL/2.4/)

OMG Specification formal/2015-06-05, Meta Object Facility (MOF), V2.5

(https://www.omg.org/spec/MOF/2.5/)

OMG Specification formal/2015-06-01, Diagram Definition, V1.1

(http://www.omg.org/spec/DD/1.1/)

OMG Document ad/03-03-41, UML for Systems Engineering RFP

(https://www.omg.org/cgi-bin/doc?ad/2003-03-41)

OMG Document ormsc/2014-06-01, Model Driven Architecture (MDA) Guide rev. 2.0

(https://www.omg.org/cgi-bin/doc?ormsc/2014-06-01)

VIM Edition 3 (VIM3), “International vocabulary of metrology - Basic and general concepts and associated terms (VIM)”, JCGM 200:2012 (JCGM 200:2008 with minor corrections)

[Dybkaer-2010] Rene Dybkaer, “ISO terminological analysis of the VIM3 concepts of ‘quantity’ and ‘kind-of-quantity’”, Metrologia 47, (2010) 127-143

3 Additional Information

3.1 Relationships to Other Standards

SysML is defined as an extension of the OMG UML 2 standard. See Clause 2 for the current version of the UML 2 standard.

SysML is intended to be supported by two evolving interoperatility standards including the OMG XMI 2 model interchange standard for UML 2 modeling tools and the ISO 10303 STEP AP233 data interchange standard for systems engineering tools. Overviews of the approach to model interchange and relevant references are included in Annex G.

SysML supports the OMG's Model Driven Architecture (MDA) initiative by its reuse of UML and related standards. See OMG MDA Guide rev 2.0.

3.2 How to Read this International Standard

This International Standard is intended to be read by systems engineers so they may learn and apply SysML, and by modeling tool vendors so they may implement and support SysML.

Systems engineers should read the Overview, Diagram Elements, and Usage Examples sub clauses in each clause, and explore the UML Extensions as they see fit. Modeling tool vendors should read all clauses. In addition, systems engineers and vendors should read Annex D, "Sample Problem," to understand how the language is applied to an example, and the document referenced by Annex F, "Requirements Traceability," to understand how the requirements in the UML for SE RFP are satisfied by this International Standard.

Although the clauses are organized into logical groupings that can be read sequentially, this International Standard can be used for reference and may be read in a non-sequential manner.

3.2.1 Organization

This International Standard is organized as follows:

Preface

INTRODUCTION

1 Scope

2 Normative References

3 Additional Information - includes Relationships to Other Standards, How to Read this International Standard, and Acknowledgments

4 Language Architecture - General Information, Design Principles, Architecture, and SysML Diagrams

5 Conformance - General Information and Conformance Types

6 Language Formalism -

- Levels of Formalism
- Clause Structure
- Conventions and Typography

STRUCTURAL CONSTRUCTS

7 Model Elements - Refactors the kernel package from UML 2 and includes some extensions to provide some foundation capabilities for model management.

8 Blocks - Reuses and extends structured classes from UML 2 composite structures to provide the fundamental capability for describing system decomposition and interconnection, and to define different types of system properties including value properties with optional units of measure.

9 Ports and Flows - Provides the semantics for defining how blocks and parts interact through ports and how items flow across connectors.

10 Constraint Blocks - Defines how blocks are extended to be used on parametric diagrams. Parametric diagrams model a network of constraints on system properties to support engineering analysis, such as performance, reliability, and mass properties analysis.

BEHAVIORAL CONSTRUCTS

11 Activities - Defines the extensions to UML 2 activities, which represent the basic unit of behavior that is used in activity, sequence, and state machine diagrams. The activity diagram is used to describe the slow of control and flow of inputs and outputs among actions.

12 Interactions - Defines the constructs for describing message based behavior used in sequence diagrams.

13 State Machines - Describes the constructs used to specify state based behavior in terms of system states and their transitions.

14 Use Cases - Describes behavior in terms of the high level functionality and uses of a system, that are further specified in the other behavioral disgrams referred to above.

CROSSCUTTING CONSTRUCTS

15 Allocations

16 Requirements

17 Profiles & Model Libraries

ANNEXES

Annex A - Diagrams

Annex B - SysML Diagram Interchange

Annex C - Deprecated Elements

Annex D - Sample Problem

Annex E - Non-normative Extensions

Annex F - Requirements Traceability

Annex G - Model Interchange

3.3 Acknowledgments

The following companies and organizations submitted or supported parts of the original version of this International Standard:

Industry

- American Systems Corporation
- BAE SYSTEMS
- Boeing
- Deere & Company
- EADS Astrium
- Eurostep
- Israel Aircraft Industries
- Lockheed Martin Corporation
- Motorola
- Northrop Grumman
- oose Innovative Informatik GmbH
- PivotPoint Technology
- Raytheon
- THALES

US Government

- NASA/Jet Propulsion Laboratory
- National Institute of Standards and Technology (NIST)
- DoD/Office of the Secretary of Defense (OSD)

Vendors

- ARTiSAN Software Tools
- Ceira Technologies
- EmbeddedPlus Engineering
- Gentleware
- IBM
- I-Logix
- Mentor Graphics
- Telelogic
- Structured Software Systems Limited
- Sparx Systems
- Vitech

Academia

- Georgia Institute of Technology

Liaisons

- Consultative Committee for Space Data Systems (CCSDS)
- Embedded Architecture and Software Technologies (EAST)
- International Council on Systems Engineering (INCOSE)
- ISO STEP AP233
- Systems Level Design Language (SLDL) and Rosetta

The following persons were members of the team that designed and wrote this International Standard: Vincent Arnould, Laurent Balmelli, Ian Bailey, James Baker, Cory Bialowas, Conrad Bock, Carolyn Boettcher, Roger Burkhart, Murray Cantor, Bruce Douglass, Harald Eisenmann, Anders Ek, Brenda Ellis, Marilyn Escue, Sanford Friedenthal, Eran Gery, Hal Hamilton, Dwayne Hardy, James Hummel, Cris Kobryn, Michael Latta, John Low, Robert Long, Kumar Marimuthu, Alan Moore, Véronique Normand, Salah Obeid, Eldad Palachi, David Price, Bran Selic, Chris Sibbald, Joseph Skipper, Rick Steiner, Robert Thompson, Jim U'Ren, Tim Weilkiens, Thomas Weigert, and Brian Willard.

In addition, the following persons contributed valuable ideas and feedback that significantly improved the content and the quality of this International Standard: Perry Alexander, Michael Chonoles, Mike Dickerson, Orazio Gurrieri, Julian Johnson, Jim Long, Henrik Lönn, Stephen Mellor, Dave Oliver, Jim Schier, Matthias Weber, Peter Shames, and the Georgia Institute of Technology research team including Manas Bajaj, Injoong Kim, Chris Paredis, Russell Peak, and Diego Tamburini. The SysML team also wants to acknowledge Pavel Hruby and his contribution by providing the Visio stencil for UML 2 that was adapted for most of the figures throughout this International Standard.

Additional organizations and individuals have contributed to further revisions of this International Standard, as completed by Finalization and Revision Task Forces listed under the OMG SysML Roadmap in the Preface above. Besides those already acknowledged above for their contributions to the original International Standard, the following additional persons have contributed to the Finalization or Revision Task Forces: Dave Banham,Yves Bernard, Graham Bleakley, Fraser Chadburn, Chris Delp, Hans Peter de Koning, Sébastien Demathieu, Peter Denno, Huascar Espinoza, Allison Barnard Feeney, Sébastien Gérard, Matthew Hause, Kenn Hussey, Nerijus Jankevicius, Steve Jenkins, Robert Karban, Darren Kelly, Andreas Korff, Frédéric Mallet, Sam Mancarella, Julio Medina, Jishnu Mukerji, Chris Paredis, Axel Reichwein, Pete Rivett, Nicolas Rouquette, George Sawyer, Andrius Strazdauskas, Kritsana Uttamang, John Watson, Bernd Wenzel. Additional organizations who supported the work of contributors to the Finalization and Revision Task Forces, not already listed for the original submission above, include 88solutions, Adaptive, Atego, EADS, CEA LIST, European Southern Observatory, European Space Agency, Fachhochschule Vorarlberg, INRIA, Mathworks, Tecnalia Research and Innovation, No Magic, and Universidad de Cantabria.

4 Language and Architecture

4.1 General

SysML reuses a subset of UML 2 and provides additional extensions needed to address requirements in the UML for Systems Engineering RFP. This International Standard documents the language architecture in terms of the parts of UML 2 that are reused and the extensions to UML 2. This clause explains design principles and how they are applied to define the SysML language architecture.

To visualize the relationship between the UML and SysML languages, consider the Venn diagram shown in Figure 4-1, where the sets of language constructs that comprise the UML and SysML languages are shown as the circles marked "UML" and "SysML," respectively. The intersection of the two circles, shown by the region marked "UML reused by SysML," indicates the UML modeling constructs that SysML reuses, called the UML4SysML subset. The region marked "SysML extensions to UML" in Figure 4.1 indicates the new modeling constructs defined for SysML that have no counterparts in UML, or which replace UML constructs. Note that there is also a part of UML 2 that is not required to implement SysML, which is shown by the region marked "UML not required by SysML."

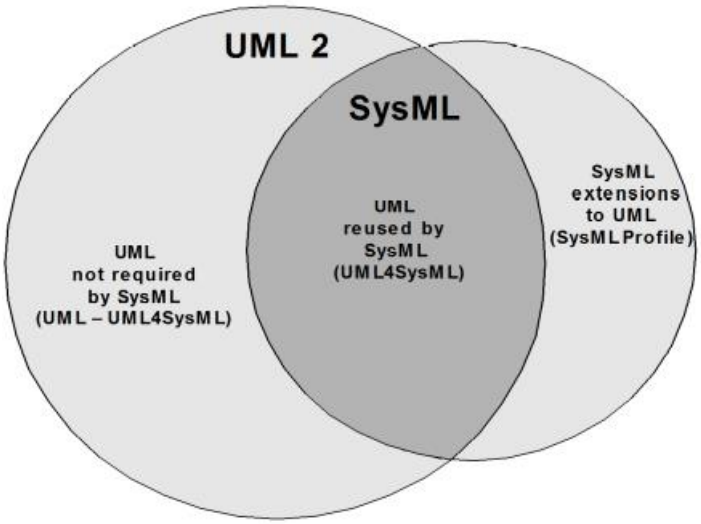


Figure 4-1: Overview of SysML/UML Interrelationship

Table 4-1 lists the metaclasses excluded from the UML4SysML subset. Table 4-2 lists the metaclasses and datatypes included in the UML4SysML subset. Table 4.3 lists the stereotypes, blocks, valuetypes, and datatypes included in SysML.

Table 4-1: UML2 metaclasses excluded from the UML4SysML subset

UML 2 metaclasses excluded from the UML4SysML subset
Artifact, ClassifierTemplateParameter, Collaboration, CollaborationUse, CommunicationPath, Component, ComponentRealization, ConnectableElementTemplateParameter, Deployment, DeploymentSpecification, Device, ExceptionHandler, ExecutionEnvironment, ExpansionNode, ExpansionRegion, Manifestation, Node, OperationTemplateParameter, ProtocolConformance, ProtocolStateMachine, ProtocolTransition, QualifierValue, ReadLinkObjectEndQualifierAction, RedefinableTemplateSignature, StringExpression, TemplateBinding, TemplateParameter, TemplateParameterSubstitution, TemplateSignature, UMLActivityDiagram, UMLAssociationEndLabel, UMLAssociationOrConnectorOrLinkShape, UMLAssociationOrConnectorOrLinkShapeKind, UMLBehaviorDiagram, UMLClassDiagram, UMLClassifierShape, UMLCompartment, UMLCompartmentableShape, UMLComponentDiagram, UMLCompositeStructureDiagram, UMLDeploymentDiagram, UMLDiagram, UMLDiagramElement, UMLDiagramWithAssociations, UMLEdge, UMLInteractionDiagram, UMLInteractionDiagramKind, UMLInteractionTableLabel, UMLKeywordLabel, UMLLabel, UMLMultiplicityLabel, UMLNameLabel, UMLNavigabilityNotationKind, UMLObjectDiagram, UMLPackageDiagram, UMLProfileDiagram, UMLRedefinesLabel, UMLShape, UMLStateMachineDiagram, UMLStateShape, UMLStereotypePropertyValueLabel, UMLStructureDiagram, UMLStyle, UMLTypedElementLabel, UMLUseCaseDiagram

Table 4-2: UML 2 metaclasses and datatypes included in the UML4SysML subsetTable

UML 2 metaclasses and datatypes included in the UML4SysML subset
Abstraction, AcceptCallAction, AcceptEventAction, Action, ActionExecutionSpecification, ActionInputPin, Activity, ActivityEdge, ActivityFinalNode, ActivityGroup, ActivityNode, ActivityParameterNode, ActivityPartition, Actor, AddStructuralFeatureValueAction, AddVariableValueAction, AggregationKind, AnyReceiveEvent, Association, AssociationClass, Behavior, BehaviorExecutionSpecification, BehavioralFeature, BehavioredClassifier, BroadcastSignalAction, CallAction, CallBehaviorAction, CallConcurrencyKind, CallEvent, CallOperationAction, CentralBufferNode, ChangeEvent, Class, Classifier, Clause, ClearAssociationAction, ClearStructuralFeatureAction, ClearVariableAction, CombinedFragment, Comment, ConditionalNode, ConnectableElement, ConnectionPointReference, Connector, ConnectorEnd, ConnectorKind, ConsiderIgnoreFragment, Constraint, Continuation, ControlFlow, ControlNode,

CreateLinkAction, CreateLinkObjectAction, CreateObjectAction, DataStoreNode, DataType, DecisionNode, Dependency, DeployedArtifact, DeploymentTarget, DestroyLinkAction, DestroyObjectAction, DestructionOccurrenceSpecification, DirectedRelationship, Duration, DurationConstraint, DurationInterval, DurationObservation, Element, ElementImport, EncapsulatedClassifier, Enumeration, EnumerationLiteral, Event, ExecutableNode, ExecutionOccurrenceSpecification, ExecutionSpecification, Expression, Extend, Extension, ExtensionEnd, ExtensionPoint, Feature, FinalNode, FinalState, FlowFinalNode, ForkNode, FunctionBehavior, Gate, GeneralOrdering, Generalization, GeneralizationSet, Image, Include, InformationFlow, InformationItem, InitialNode, InputPin, InstanceSpecification, InstanceValue, Interaction, InteractionConstraint, InteractionFragment, InteractionOperand, InteractionOperatorKind, InteractionUse, Interface, InterfaceRealization, InterruptibleActivityRegion, Interval, IntervalConstraint, InvocationAction, JoinNode, Lifeline, LinkAction, LinkEndCreationData, LinkEndData, LinkEndDestructionData, LiteralBoolean, LiteralInteger, LiteralNull, LiteralReal, LiteralSpecification, LiteralString, LiteralUnlimitedNatural, LoopNode, MergeNode, Message, MessageEnd, MessageEvent, MessageKind, MessageOccurrenceSpecification, MessageSort, Model, MultiplicityElement, NamedElement, Namespace, ObjectFlow, ObjectNode, ObjectNodeOrderingKind, Observation, OccurrenceSpecification, OpaqueAction, OpaqueBehavior, OpaqueExpression, Operation, OutputPin, Package, PackageImport, PackageMerge, PackageableElement, Parameter, ParameterDirectionKind, ParameterEffectKind, ParameterSet, ParameterableElement, PartDecomposition, Pin, Port, PrimitiveType, PrimitiveTypes::Boolean, PrimitiveTypes::Integer, PrimitiveTypes::Real, PrimitiveTypes::String, PrimitiveTypes::UnlimitedNatural, PrimitiveValueTypes::Boolean, Profile, ProfileApplication, Property, Pseudostate, PseudostateKind, RaiseExceptionAction, ReadExtentAction, ReadIsClassifiedObjectAction, ReadLinkAction, ReadLinkObjectEndAction, ReadSelfAction, ReadStructuralFeatureAction, ReadVariableAction, Realization, Reception, ReclassifyObjectAction, RedefinableElement, ReduceAction, Region, Relationship, RemoveStructuralFeatureValueAction, RemoveVariableValueAction, ReplyAction, SendObjectAction, SendSignalAction, SequenceNode, Signal, SignalEvent, Slot, StartClassifierBehaviorAction, StartObjectBehaviorAction, State, StateInvariant, StateMachine, Stereotype, StructuralFeature, StructuralFeatureAction, StructuredActivityNode, StructuredClassifier, Substitution, TestIdentityAction, TimeConstraint, TimeEvent, TimeExpression, TimeInterval, TimeObservation, Transition, TransitionKind, Type, TypedElement, UnmarshallAction, Usage, UseCase, ValuePin, ValueSpecification, ValueSpecificationAction, Variable,VariableAction, Vertex, VisibilityKind, WriteLinkAction, WriteStructuralFeatureAction, WriteVariableAction

Table 4-3: SysML stereotypes, blocks, valuetypes, and datatypes

SysML stereotypes, blocks, valuetypes, and datatypes
AcceptChangeStructuralFeatureEventAction, AdjunctProperty, Allocate, AllocateActivityPartition, BindingConnector, Block, BoundReference, ChangeStructuralFeatureEvent, ClassifierBehaviorProperty, Conform, ConnectorProperty, ConstraintBlock, Continuous, ControlOperator, ControlValueKind, Copy, DeriveReqt, DirectedFeature, DirectedRelationshipPropertyPath, Discrete, DistributedProperty, ElementGroup, ElementPropertyPath, EndPathMultiplicity, Expose, FeatureDirectionKind, FlowProperty, FullPort, InterfaceBlock, InvocationOnNestedPortAction, ItemFlow, NestedConnectorEnd, NoBuffer, Optional, Overwrite, ParticipantProperty, PrimitiveValueTypes::Boolean, PrimitiveValueTypes::Complex, PrimitiveValueTypes::Integer, PrimitiveValueTypes::Number, PrimitiveValueTypes::Real, PrimitiveValueTypes::String, Probability, Problem, PropertySpecificType, ProxyPort, Rate, Rationale, Refine, Requirement, Satisfy, Stakeholder, TestCase, Trace, TriggerOnNestedPort, ValueType, VerdictKind, Verify, View, Viewpoint

4.2 Design Principle

The fundamental design principles for SysML are:

- Requirements-driven - SysML is intended to satisfy the requirements of the UML for SE RFP.
- UML reuse - SysML reuses UML wherever practical to satisfy the requirements of the RFP, and when modifications are required, they are done in a manner that strives to minimize changes to the underlying language. Consequently, SysML is intended to be relatively easy to implement for vendors who support UML 2.
- UML extensions - SysML extends UML as needed to satisfy the requirements of the RFP. The primary extension mechanism is the UML 2 profile mechanism as further refined in Clause 17, “Profiles & Model Libraries.”
- Partitioning - The package is the basic unit of partitioning in this International Standard. The packages partition the model elements into logical groupings that minimize circular dependencies among them.
- Layering - SysML packages are specified as an extension layer to the UML metamodel.
- Interoperability - SysML inherits the XMI interchange capability from UML. SysML is also intended to be supported by the ISO 10303-233 data interchange standard to support interoperability among other engineering tools.

SysML provides three model libraries:

- PrimitiveValueTypes, see 8.3.3.1
- UnitAndQuantityKind, see 8.4.6
- ControlValues, see 11.3.3

4.3 Architecture

The relationship between SysML and UML 2 is shown in Figure 4-1. SysML extends UML 2's StandardProfile (see Clause 22 in the UML 2.5 specification) whose Trace and Refine stereotypes provide the basis for Requirement traceability in SysML (see Clause 16, "Requirements" in this International Standard).

Although SysML indirectly imports the UML 2 PrimitiveTypes library (see Clause 21 in the UML 2.5 specification) due to the transitivity of package import, SysML provides a PrimitiveValueTypes model library that systems engineers can extend via SysML's ValueType stereotype. In the remainder of this document, the unqualified references to Boolean, Integer, Real, and String should be interpreted as follows:

In the context of the definition of a SysML Stereotype, the name refers to the definition of a UML::PrimitiveType in the UML 2 PrimitiveTypes library.

- In the context of the definition of a SysML Stereotype, the name refers to the definition of a UML::PrimitiveType in the UML 2 PrimitiveTypes library.
- Elsewhere, the name refers to the definition of a SysML::ValueType stereotype of UML::DataType in the SysML PrimitiveValueTypes library.

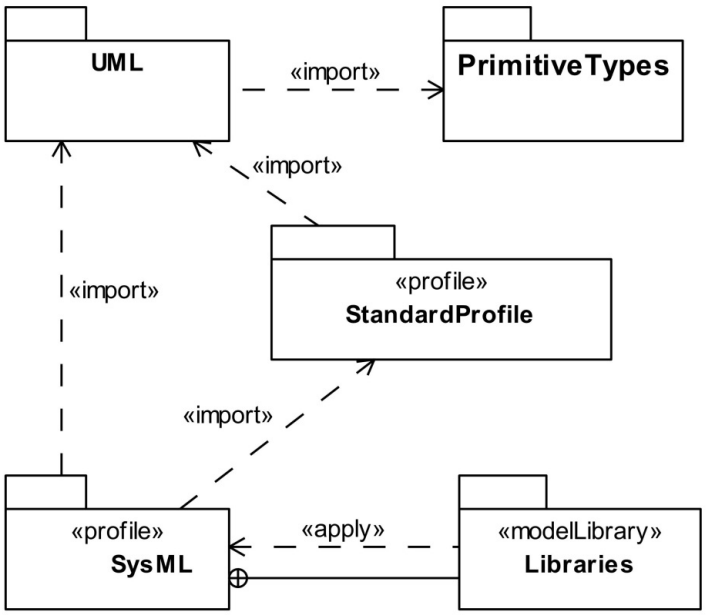


Figure 4-2: SysML Extension of UMLFigure

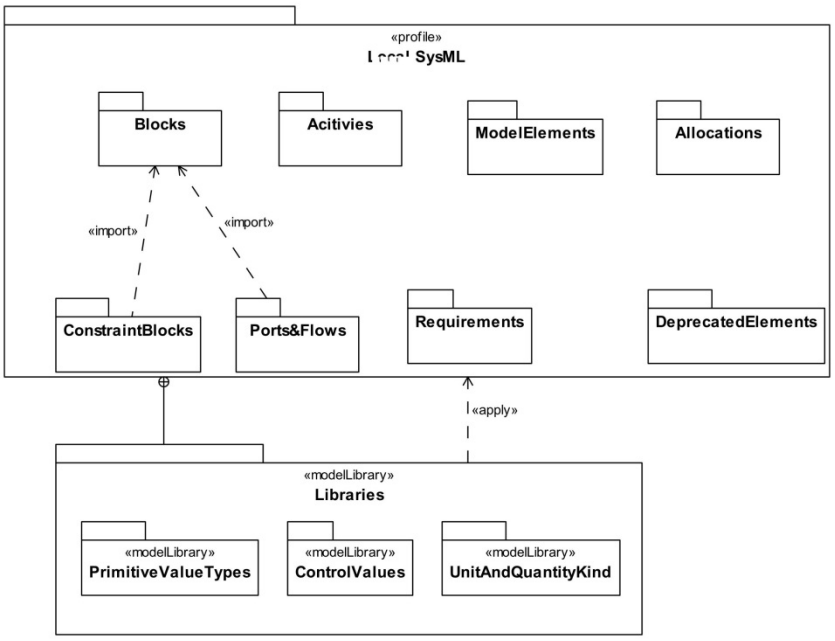


Figure 4-3: SysML Package Structure

As previously stated, the design approach for SysML is to reuse a subset of UML and create extensions to support the specific concepts needed to satisfy the requirements in the UML for SE RFP. The SysML package structure shown in Figure 4-2: SysML Extension of UMLFigure contains a set of packages that correspond to concept areas in SysML that have been extended.

The SysML packages extend UML as follows:

- SysML::Model Elements extends Classification, Common Structure
- SysML::Blocks extends Classification, Structured Classifiers, Common Structure, Simple Classifiers
- SysML::ConstraintBlocks extends Structured Classifiers
- SysML::Ports and Flows extends Actions, Common Behavior, Classification
- SysML::Activities extends Activities.
- SysML::Allocations extends Common Structure, Activities
- SysML::Requirements extends Common Structure, Classification, Common Behavior,
- Structured Classifiers
- SysML::DeprecatedElements extends Common Structure, Simple Classifiers, Classification, Structured Classifiers, Actions, and SysML Item Flows

Figure 4-4 shows non-normative packages in this International Standard that depend on SysML and UML. Note that the QUDV and ISO-80000 libraries are described in non-normative annexes to this specification.

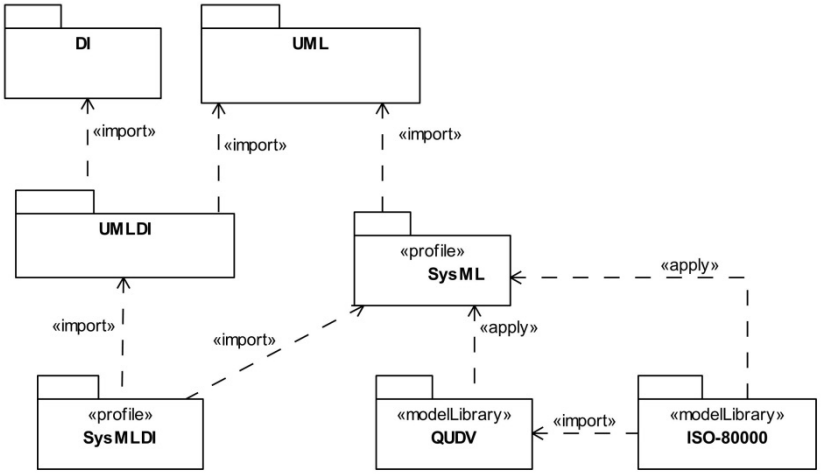


Figure 4-4: Non-normative Package Structure

4.4 Extension Mechanisms

This International Standard uses the following mechanisms to define the SysML extensions:

- UML stereotypes
- UML diagram extensions
- Model libraries

SysML stereotypes define new modeling constructs by extending existing UML 2 constructs with new properties and constraints. SysML diagram extensions define new diagram notations that supplement diagram notations reused from UML 2. SysML model libraries describe specialized model elements that are available for reuse. Additional non-normative extensions are included in Annex E “Non-normative Extensions.”

The SysML user model is created by instantiating its metamodel and applying the stereotypes specified in the SysML profile, and optionally referencing or subclassing the model elements in the SysML model library. Clause 17, “Profiles & Model Libraries” describes how profiles and model libraries are applied and how they can be used to further extend SysML.

4.5 SysML Diagrams

The SysML diagram taxonomy is shown in Figure A.1 in Annex A. The concrete syntax (notation) for the diagrams along with the corresponding specification of the UML extensions is described in Parts II - IV. The Diagrams Annex (Annex A) describes generalized features of diagrams, such as their frames and headings. A model of SysML diagrams to support interchange is in SysML Diagram Interchange Annex (Annex B).

5 Conformance

5.1 Overview

Conformance with SysML requires that the subset of UML required for SysML is implemented, and that the SysML extensions to this subset are implemented. SysML has three types of conformance, listed in 5.2, which shall all be supported to fully conform to SysML. Conformance does not include DeprecatedElements.

5.2 Conformance Types

An implementation of SysML shall comply with both the subset of UML4SysML and the SysML extensions. The types of SysML conformance extend corresponding types in UML as follows:

- Abstract syntax conformance. A tool demonstrating abstract syntax conformance provides a user interface and/or API that enables instances of concrete SysML stereotypes (which are applications of stereotypes to instances of UML metaclasses) and model library elements to be created, read, updated, and deleted. The tool shall also provide a way to validate the well-formedness of models that corresponds to the constraints defined in SysML.
- Concrete syntax conformance. A tool demonstrating concrete syntax conformance provides a user interface and/or API that enables instances of SysML notation to be created, read, updated, and deleted. This includes conformance to the notation defined in the “Diagram Elements” tables and diagrams extension sub clauses in each clause of this International Standard. Note that a conforming tool may provide the ability to create, read, update, and delete additional diagrams and notational elements that are not defined in SysML.
- Model interchange conformance. A tool demonstrating model interchange conformance can import and export conformant XMI for all valid SysML models, including models with profiles defined and/or applied. Model interchange conformance implies abstract syntax conformance. See more information in Annex G.

6 Language Formalisms

6.1 Levels of Formalism

SysML is specified using a combination of UML modeling techniques and precise natural language to balance rigor and understandability. Use of more formal constraints and semantics may be applied in future versions to further increase the precision of the language.

6.2 Clause Structure

The clauses are organized according to the SysML packages as described in the language architecture and selected reusable portions of UML 2 packages. This sub clause provides information about how each clause is organized.

6.2.1 Overview

This sub clause provides an overview of the SysML modeling constructs defined in the subject package, which are usually associated with one or more SysML diagram types.

6.2.2 Diagram Elements

This sub clause provides tables that summarize the concrete syntax (notation) and abstract syntax references for the graphic nodes and paths associated with the relevant diagram types. The diagram elements tables are intended to include all of the diagrammatic constructs used in SysML. However, they do not represent all the different combinations in which they can be used. The reader should refer to the usage examples in the clauses and the sample problem (Annex D:) for typical usages of the concrete syntax. General diagram information on the use of diagram frames and headings can be found in Annex A.

The diagram elements tables and the additional usage examples fill an important role in defining the scope of SysML. As described in Clause 4, “Language Architecture,” SysML imports many entire packages from the UML metamodel, which it then reuses and extends. Only a subset of the entire UML metamodel, however, is required to support the notations included in SysML.

Unless a type of diagram element is shown in some form in one of the SysML diagram elements tables, or in a usage example in one of the normative SysML clauses, it is not considered to be part of the subset of UML included within SysML, even if the UML metamodel packages support additional constructs. For example, SysML imports the entire Dependencies package from UML, but it includes diagram elements for only a subset of the dependency types defined in this package.

6.2.3 UML Extensions

This sub clause specifies the SysML extensions to UML in terms of diagram extensions and semantic extensions. Diagram extensions are included when the concrete syntax uses notation other than the standard stereotype notation as defined in the Profiles & Model Libraries clause. Semantic extensions consist of stereotype and model library extensions. Stereotype extensions always include the abstract syntax that identifies which metaclasses a stereotype extends. Each stereotype includes a general description with a definition and semantics, along with stereotype properties (attributes), and constraints. Each constraint consists of a textual description and may be followed by a formal constraint expressed in

Object Constraint Language (OCL). If there is any ambiguity between the two, the OCL statement of the constraint takes precedence. The model libraries are defined as subclasses of existing metaclasses.

6.2.4 Usage Examples

This sub clause shows how the SysML modeling constructs can be applied to solve systems engineering problems and is intended to reuse and/or elaborate the sample problem in Annex D.

6.3 Conventions and Typography

In the description of SysML, the following conventions have been used:

- When referring to stereotypes, metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are used.
- No visibilities are presented in the diagrams, since all elements are public.
- If a sub clause is not applicable, it is not included, except for the top-level sub clauses outlined in sub clause 6.2.
- Stereotype, metaclass, and metaassociation names: initial embedded capitals are used (e.g., “ModelElement,” “ElementReference”).
- Boolean metaattribute names: always start with “is” (e.g., “isComposite”).
- Enumeration types: always end with “Kind” (e.g., “DependencyKind”).

STRUCTURAL CONSTRUCTS

This page intentionally left blank.

7 Model Elements

7.1 Overview

The ModelElements package of SysML defines general-purpose constructs that may be shown on multiple SysML diagram types. These include package, model, various types of dependencies (e.g., import, access, refine, realization), constraints, and comments. The package diagram defined in this clause is used to organize the model by partitioning model elements into packageable elements and establishing dependencies between the packages and/or model elements within the package. The package defines a namespace for the packageable elements. Model elements from one package can be imported and/or accessed by another package. This organizational principle is intended to help establish unique naming of the model elements and avoid overloading a particular model element name. Packages can also be shown on other diagrams such as the block definition diagram, requirement diagram, and behavior diagrams.

Constraints are used to capture simple constraints associated with one or more model elements and can be represented on several SysML diagrams. The constraint can represent a logical constraint such as an XOR, a condition on a decision branch, or a mathematical expression. The constraint has been significantly enhanced in SysML as specified in Clause 10, “Constraint Blocks” to enable it to be reused and parameterized to support engineering analysis.

Comments can be associated with any model element and are quite useful as an informal means of documenting the model. SysML has introduced an extension to a comment called rationale to facilitate the system modeler in capturing decisions. The rationale may be attached to any entity, such as a system element (block), or to any relationship, such as the satisfy relationship between a design element and a requirement. In the latter case, it may be used to capture the basis for the design decision and may reference an analysis report or trade study for further elaboration of the decision. In addition, SysML includes an extension of a comment to reflect a problem or issue that can be attached to any other model element.

7.1.1 View and Viewpoint

The concepts of viewpoint and view are articulated in ISO-42010 (formerly IEEE-1471). SysML viewpoint and view constructs are consistent with the ISO-42010 standard. Typical examples may include an operational, manufacturing, or security viewpoint and view.

Systems engineers use SysML to make models of systems-the result is the system model, which is what we mean most of the time when we speak of “the model.” Along with that model, systems engineers may also use SysML to make a model of the information to be presented to the stakeholders to address their concerns. The result is the viewpoint and view model, which helps systems engineers assure that stakeholders get the understanding they need from the system model.

The viewpoint and view model can also be thought of as a description model, which augments a system model. A viewpoint and view model exposes elements of one or more system models. In particular, a viewpoint is a specification of rules for constructing a view to address a set of stakeholder concerns. The view is intended to represent the system from this viewpoint. This enables stakeholders to specify aspects of the system model that are important to them from their viewpoint, and then represent those aspects of the system in a specific view.

The viewpoint describes the point of view of a set of stakeholders by framing the concerns of the stakeholders along with the method for producing a view that addresses those concerns. The method describes the expectation of what stakeholder(s) wish to see exposed from the model, how the stakeholder wishes the information to be structured and presented, and in what kind of artifact the stakeholder wants to consume the information. In other words, the method is the set of rules that describe how the view should express the information from the model to address the stakeholder

concerns. The method can be specified as a process and/or a set of constraints for producing a view, which may include rules or instructions for analyzing or verifying the view content.

The view is the modeling element that represents the artifact that is presented to the stakeholder. A view conforms to only one viewpoint to ensure that only one method is applied to the view. The view shall be related to the model that contains the information and the method that produces the view. The view is used by a rendering application to generate the artifact, such as a document.

In summary, the viewpoint description specifies the following:

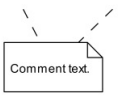
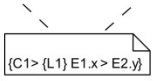
1. What kind of information the view should contain.
2. How the information should be expressed, i.e., what modeling language is required for the model that will appear in the view. (Note: this is not to be confused with the language used for specifying the viewpoint method).
3. The presentation format that specifies how the information should be presented in an artifact, e.g., specifying that data values should be plotted on a graph or a particular tabular style, or that both English and Spanish text should be provided, or that photographs be shows in color with minimum dimensions of 100 millimeters square.
4. The file format of the artifacts that are generated from the view (e.g., set of slides in ppt, a PDF, a Word document, a web viewable format, ...).

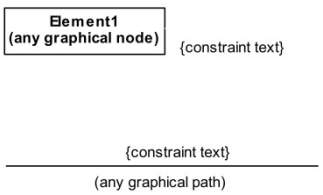
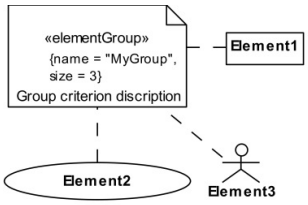
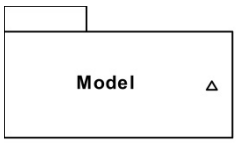
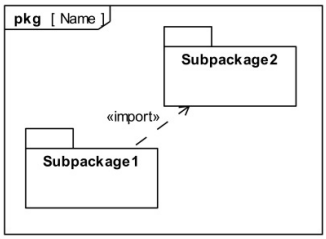
It is important to understand that while the view is a SysML construct that exists within a SysML model, artifacts generated from views potentially live outside of the modeling environment as the means to satisfy stakeholder concerns. An artifact such as a movie or a PDF document is not directly incorporated in a SysML model, while the view which represents the artifact does reside in the model as a specification of that artifact. The relationship between the viewpoint and view model and the corresponding artifact is similar to the relationship between the system model and the system that is the subject of the model.

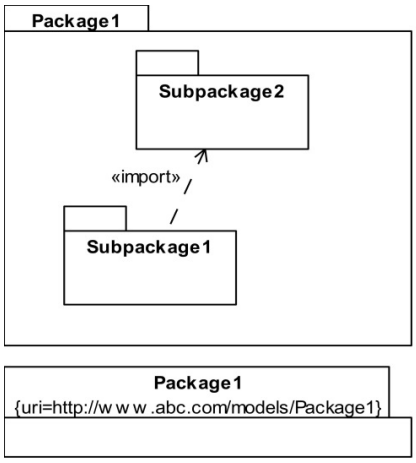
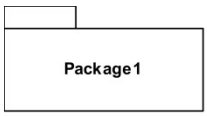
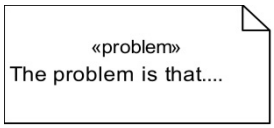
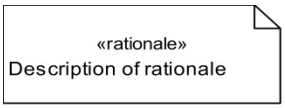
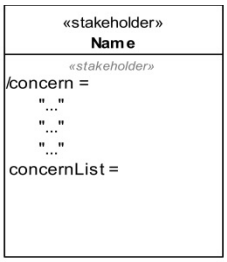
7.2 Diagram Elements

Many of the diagram elements defined in this clause, specifically comments, constraints, problem, rationale, and dependencies, including the dependency subtypes Conform, Realization, and Refine, may be shown on all SysML diagram types, in addition to the diagram elements that are specific to each diagram type.

Table 7-1: Graphical nodes by ModelElements package

Element Name	Concrete Syntax Example	Abstract Syntax Reference
Comment		**UML4SysML::Comment**
ConstraintNote		**UML4SysML::Constraint**

Element Name	Concrete Syntax Example	Abstract Syntax Reference
ConstraintTextualNote		**UML4SysML::Constraint**
ElementGroup		**SysML::ModelElements::Element Group**
Model		**UML4SysML::Model**
PackageDiagram		**UML4SysML::Package**

Element Name	Concrete Syntax Example	Abstract Syntax Reference
PackageWithNameInTab		**UML4SysML::Package**
PackageWithNameInside		**UML4SysML::Package**
Problem		**SysML::ModelElements::Problem**
Rationale		**SysML::ModelElements::Rationale**
Stakeholder		**SysML::ModelElements::Stakeholder**

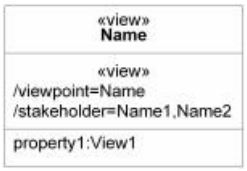
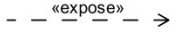
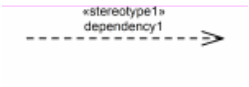
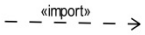
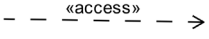
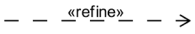
Element Name	Concrete Syntax Example	Abstract Syntax Reference
View		**SysML::ModelElements::View**
Viewpoint		**SysML::ModelElements::Viewpoint**

Table 7-2: Graphical paths defined by ModelElements package

Element Name	Concrete Syntax Example	Abstract Syntax Reference
Conform		**UML4SysML::Conform**
Expose		**SysML::ModelElements::Expose**
Dependency		**UML4SysML::Dependency**
PublicPackageImport		**UML4SysML::PackageImport with visibility = public**
PrivatePackageImport		**UML4SysML::PackageImport with visibility = private**

Element Name	Concrete Syntax Example	Abstract Syntax Reference
PackageContainment		**UML4SysML::Package::ownedElement**
Realization		**UML4SysML::Realization**
Refine		**UML4SysML::Refine**

7.3 UML Extensions

7.3.1 Diagram Extensions

7.3.1.1 UML Diagram Elements not Included in SysML

The notation for a “merge” dependency between packages, using a «merge» keyword on a dashed-line arrow, is not included in SysML. UML uses package merge in the definition of its own metamodel, which SysML builds on, but SysML does not support this capability for user-level models.

NOTE: Combining packages that have the same named elements, resulting in merged definitions of the same names, could cause confusion in user models and adds no inherent modeling capability, and so has been left out of SysML.

7.3.2 Stereotypes

Package ModelElements

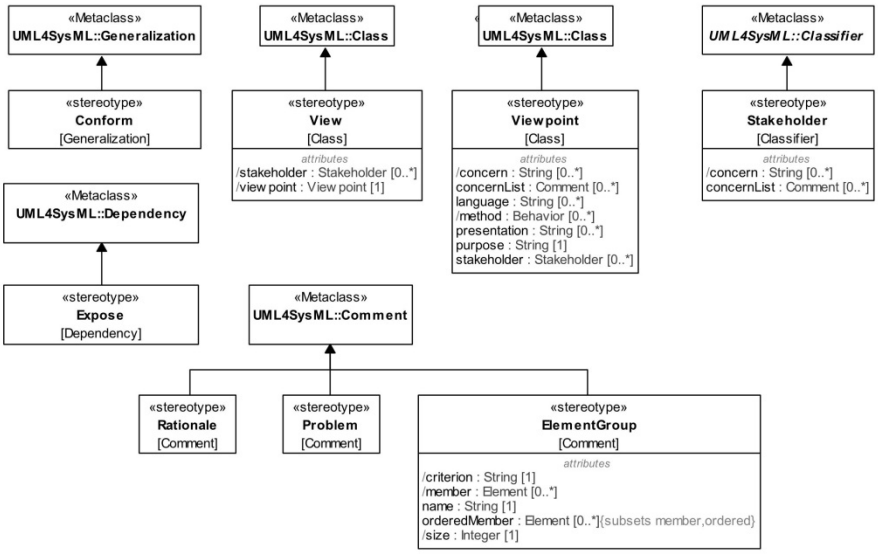


Figure 7-1: Stereotypes defined in package ModelElements

7.3.2.1 Conform

Description

A Conform relationship is a generalization between a view and a viewpoint. The view conforms to the specified rules and conventions detailed in the viewpoint. When this is done, the view is said to conform to the viewpoint. Conform extends UML generalization.

Association Ends

- base_Generalization : Generalization [1]

Constraints

- `1_general_is_viewpoint`
  The general classifier shall be an element stereotyped by Viewpoint

```
Viewpoint.allInstances()->exists(v | v.base_Class = self.base_Generalization.general)
```

- `2_specific_is_view`
  The specific classifier shall be an element that is stereotyped by View

```
View.allInstances()->exists(v | v.base_Class = self.base_Generalization.specific)
```

7.3.2.2 ElementGroup

Description

The ElementGroup stereotype provides a lightweight mechanism for grouping various and possibly heterogeneous model elements by extending the capability of comments to refer to multiple annotated elements. For example, it can group elements that are associated with a particular release of the model, have a certain risk level, or are associated with a

legacy design. The semantics of ElementGroup is modeler-defined. In particular, the body text is not restricted. It can describe the grouped elements as well as elements or values related to the grouped elements.

Element groups are named using the name property. The criterion for membership in an element group is specified by the body of the comment the stereotype is applied to. By grouping elements, the modeler asserts that the criterion of the group applies to the member. Optionally, members of an element group can be ordered using its orderedMember property.

ElementGroups appear in diagrams as comments, and properties of the stereotype appear in the notation for stereotype properties. Grouped elements are the annotated elements of the comment to which the stereotype is applied. This has several implications:

- Element groups do not own their elements and thus an element can participate in an unlimited number of groups.
- The elements in a group are identified by the modeler, as opposed to being the result of a query, as in views.
- Element groups can be members of other element groups, but this does not imply that members of the first are members of the second.

Elements related to the grouped elements are not included in the group, even though the body text can address them. In particular, element groups annotating deeply nested properties or properties with bindings are grouping only the properties, rather than their nesting or their bound properties.

Grouped elements are also limited to elements of models, rather than instances of values of those model elements. In particular, element groups annotating blocks or properties are not grouping the instances of the blocks or the values of the properties. However, since the semantics of ElementGroup is left to the modeler, the body text can refer to related elements outside the group, such as instances and values of the grouped elements, or to bound properties. The modeler is then responsible for writing body text that explains the implications for the related elements. For instance:

- A group with the criterion: "Authored by John" could annotate any model element added in the model by John. This body text does not address any related elements. For example, if the annotated element is a property bound to another property, the group would not imply authorship of the second property.
- A group with the criterion: "Instances are manufactured in a foreign country" could annotate Blocks to indicate that any instances of those Blocks are produced in a foreign country. This body text does not address the Block itself, which is not necessarily "manufactured" in a foreign country.
- A group with criterion: "Values are manufactured in a foreign country" could annotate properties, including part properties, to indicate the values of the property are produced in a foreign country. This body text does not address the property itself, which is not necessarily "manufactured" in a foreign country. Since the text is about values of the property, it is also about values of other properties that might be bound to the annotated property, because the values of bound properties are the same.

Attributes

- /criterion : String [0..1]
  Specifies the rationale for being member of the group. Adding an element to the group asserts that the criterion applies to this element. Derived from Comment::body.
  (derived)

- /member : Element [0..*]
  Set specifying the members of the group. Derived from Comment::annotatedElement.
  (derived)
- name : String [1]
  Name of the element group
- orderedMember : Element [0..*]
  Organize member according to an arbitrary order. Optional.
  (subsets: ElementGroup::member)
- /size : Integer [1]
  Number of members in the group. Derived.
  (derived)

Association Ends

- base_Comment : Comment [1]

Operations

- allGroups (in e : Element) : ElementGroup [0..*]
  The query allGroups() returns the set of all the groups an element is member of.
- criterion () : String [0..1]
  The query criterion() returns the text describing the criterion defining the group.
- member () : Element [0..*]
  The query member() returns the set of all the members of the group.
- size () : Integer [1]
  The query size() returns the number of elements which are members of the group.

7.3.2.3 Expose

Description

The expose relationship relates a view to one or more model elements. Each model element is an access point to initiate the query. The view and the model elements related to the view are passed to the constructor when it is invoked. The method describes how the exposed elements are navigated to extract the desired information.

Association Ends

- base_Dependency : Dependency [1]

Constraints

- `1_client_is_view`
  The client shall be an element stereotyped by View.

```
View.allInstances()->exists(v | v.base_Class = self.base_Dependency.client)
```

7.3.2.4 Problem

Description

A Problem documents a deficiency, limitation, or failure of one or more model elements to satisfy a requirement or need, or other undesired outcome. It may be used to capture problems identified during analysis, design, verification, or

manufacture and associate the problem with the relevant model elements. Problem is a stereotype of comment and may be attached to any other model element in the same manner as a comment.

Association Ends

- base_Comment : Comment [1]

7.3.2.5 Rationale

Description

A Rationale documents the justification for decisions and the requirements, design, and other decisions. A Rationale can be attached to any model element including relationships. It allows the user, for example, to specify a rationale that may reference more detailed documentation such as a trade study or analysis report. Rationale is a stereotype of comment and may be attached to any other model element in the same manner as a comment.

Association Ends

- base_Comment : Comment [1]

7.3.2.6 Stakeholder

Description

A stakeholder represents a role, group, or individual who has concerns that will be addressed by the View of the model.

Attributes

- /concern : String [0..*]
  (derived)
- concernList : Comment [0..*]

Association Ends

- base_Classifier : Classifier [1]

Constraints

- 1_not_association
  A Stakeholder stereotype can only be applied to UML::Actor or UML::Class which are not a UML::Association.

```
self.base_Classifier.oclIsKindOf(UML::Actor)
or
(self.base_Classifier.oclIsKindOf(UML::Class)
and
not self.base_Classifier.oclIsKindOf(UML::Association))
```

- not_association
  The stakeholder stereotype can only be applied to UML::Actor or UML::Class which are not a UML::Association

```
(self.base_Classifier.oclIsKindOf(UML::Actor) or
self.base_Classifier.oclIsKindOf(UML::Class))
and not self.base_Classifier.oclIsKindOf(UML::Association)
```

7.3.2.7 View

Description

A View is a model element that represents a real world artifact that can be presented to stakeholders. The view is the result of querying one or more models that are defined by a viewpoint method. The view shall conform to the viewpoint in terms of the viewpoint stakeholders, concerns, method, language, and presentation requirements.

It is sometimes desirable to construct views from other views, and to establish an order for presenting the views. Views may include one or more views as properties, each of which conforms to their viewpoint. The order of the referenced views is reflected in the property order.

The information may be presented to the stakeholder in any format specified by the viewpoint, which may include figures, tables, plots, entire documents, presentation slides, or video.

Attributes

- /stakeholder : Stakeholder [0..*]
  The list of stakeholders is derived from the viewpoint the view conforms to.
  (derived)
- /viewpoint : Viewpoint [1]
  The viewpoint for this View is derived from the conform relationship.
  (derived)

Association Ends

- base_Class : Class [1]

Constraints

- 1_single_viewpoint
  A view shall only conform to a single viewpoint

```
Conform.allInstances()->select(base_Generalization.specific = self.base_Class)
->size() = 1
```

- 2_viewpoint_derived_from_conform
  The derived value of the viewpoint shall be the classifier stereotyped by Viewpoint that is the general classifier of the generalization relationship stereotyped by Conform for which the View is the specific classifier

```
self.viewpoint = Viewpoint.allInstances()->any(base_Class = Conform.allInstances()
->any(base_Generalization.specific = self.base_Class).base_Generalization.general)
```

- 3_stakeholder_derived_from_conform
  The derived values of the stakeholder attribute shall be the classifiers stereotyped by Stakeholder that are the values of the stakeholder attribute of the general classifier of the generalization relationship stereotyped by Conform for which the View is the specific classifier.

```
self.stakeholder = Viewpoint.allInstances()->any(base_Class = Conform.allInstances()
->any(base_Generalization.specific =
self.base_Class).base_Generalization.general).stakeholder
```

7.3.2.8 Viewpoint

Description

A Viewpoint is a specification of the conventions and rules for constructing and using a view for the purpose of addressing a set of stakeholder concerns. They specify the elements expected to be represented in the view, and may be formally or informally defined. For example, the security viewpoint may require the security requirements, security functional and physical architecture, and security test cases.

Attributes

- /concern : String [0..*]
  The interest of the stakeholders displayed as the body of the comments from concernList.
  (derived)

- concernList : Comment [0..*]
  The interests of the stakeholders addressed by this viewpoint.
- language : String [0..*]
  The languages used to express the models that represent content which is represented by the view. The language specification such as its metamodel, profile, or other language specification is referred to by its URI.
- /method : Behavior [0..*]
  The behavior is derived from the method of the operation with the Create stereotype.
  (derived)
- presentation : String [0..*]
  The specifications prescribed for formatting and styling the view.
- purpose : String [1]
  The purpose addresses the stakeholder concerns.
- stakeholder : Stakeholder [0..*]
  Set of stakeholders whose concerns are to be addressed by the viewpoint.

Association Ends

- base_Class : Class [1]

Constraints

- `1_method_derived_from_create_operations`
  The derived values of the method attribute shall be the names of the methods of the operations stereotyped by the UML Create stereotype on the classifier stereotyped by Viewpoint.

```
self.method = self.base_Class.allFeatures()->select(f |
f.oclIsKindOf(UML::Operation))->select(o |
Standard::Create.allInstances().base_BehavioralFeature
->includes(o)).oclAsType(UML::Operation).method
```

- `2_create_view_operation`
  The property ownedOperation shall include at least one operation named "View" with the UML Create stereotype applied.

```
self.base_Class.ownedOperation->exists(o | o.name='View' and
Standard::Create.allInstances().base_BehavioralFeature->includes(o))
```

8 Blocks

8.1 Overview

Blocks are modular units of system description. Each block defines a collection of features to describe a system or other element of interest. These may include both structural and behavioral features, such as properties and operations, to represent the state of the system and behavior that the system may exhibit.

Blocks provide a general-purpose capability to model systems as trees of modular components. The specific kinds of components, the kinds of connections between them, and the way these elements combine to define the total system can all be selected according to the goals of a particular system model. SysML blocks can be used throughout all phases of system specification and design, and can be applied to many different kinds of systems. These include modeling either the logical or physical decomposition of a system, and the specification of software, hardware, or human elements. Parts in these systems may interact by many different means, such as software operations, discrete state transitions, flows of inputs and outputs, or continuous interactions.

The Block Definition Diagram in SysML defines features of blocks and relationships between blocks such as associations, generalizations, and dependencies. It captures the definition of blocks in terms of properties and operations, and relationships such as a system hierarchy or a system classification tree. The Internal Block Diagram in SysML captures the internal structure of a block in terms of properties and connectors between properties. A block can include properties to specify its values, parts, and references to other blocks. Ports are a special class of property used to specify allowable types of interactions between blocks, and are described in Clause 9, “Ports and Flows.” Constraint Properties are a special class of property used to constrain other properties of blocks, and are described in Clause 10 “Constraint Blocks.” Various notations for properties are available to distinguish these specialized kinds of properties on an internal block diagram.

A property can represent a role or usage in the context of its enclosing block. A property has a type that supplies its definition. A part belonging to a block, for example, may be typed by another block. The part defines a local usage of its defining block within the specific context to which the part belongs. For example, a block that represents the definition of a wheel can be used in different ways. The front wheel and rear wheel can represent different usages of the same wheel definition. SysML also allows each usage to define context-specific values and constraints associated with the individual usage, such as 25 psi for the front tires and 30 psi for the rear tires.

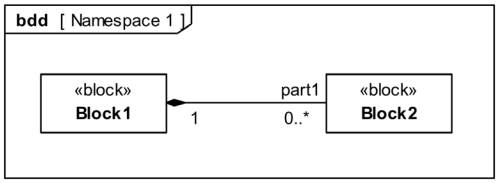
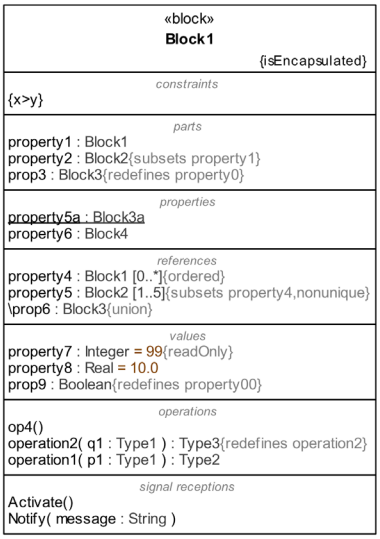
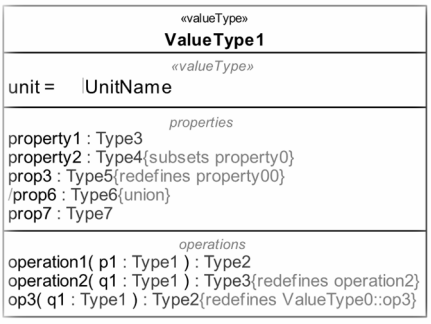
Blocks may also specify operations or other features that describe the behavior of a system. Except for operations, this clause deals strictly with the definition of properties to describe the state of a system at any given point in time, including relations between elements that define its structure. Clause 9, “Ports and Flows” specifies specific forms of interactions between blocks, and the Behavioral Constructs including activities, interactions, and state machines can be applied to blocks to specify their behavior. Clause 15, “Allocations” describes ways to allocate behavior to parts and blocks.

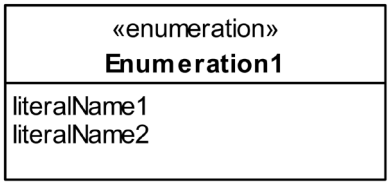
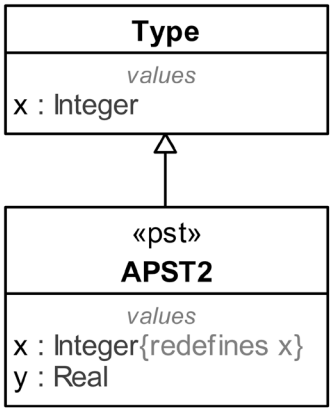
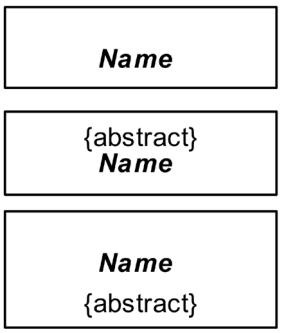
SysML blocks are based on UML classes as extended by UML composite structures. Some capabilities available for UML classes, such as more specialized forms of associations, have been excluded from SysML blocks to simplify the language. SysML blocks always include an ability to define internal connectors, regardless of whether this capability is needed for a particular block. SysML Blocks also extend the capabilities of UML classes and connectors with reusable forms of constraints, multi-level nesting of connector ends, participant properties for composite association classes, and connector properties. SysML blocks include several notational extensions as specified in this clause.

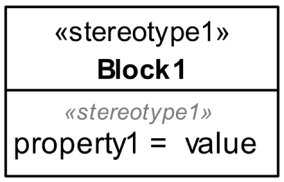
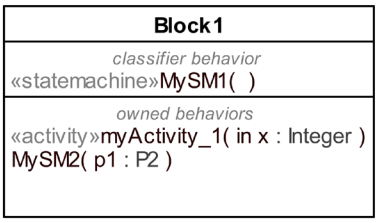
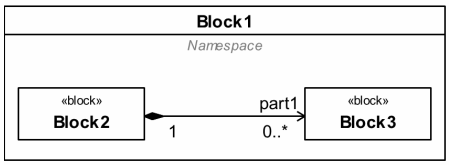
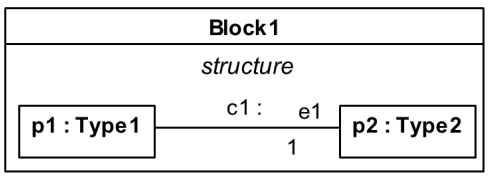
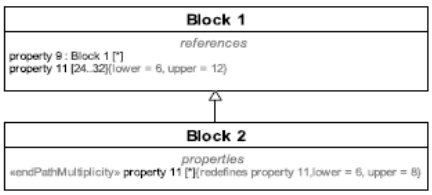
8.2 Diagram Elements

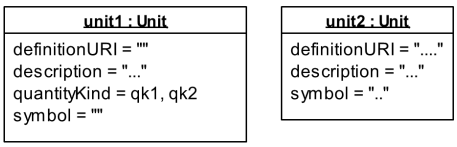
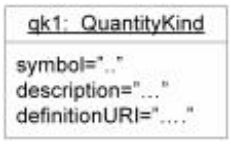
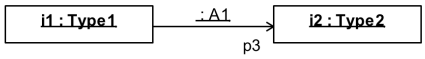
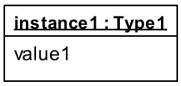
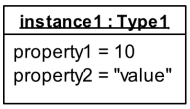
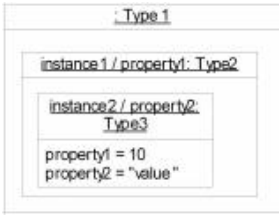
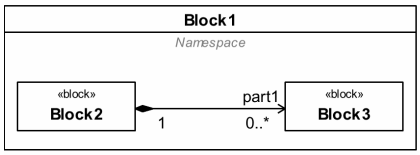
8.2.1 Block Definition Diagram

Table 8-1: Graphical nodes defined in Block Definition diagrams table

Element Name	Concrete Syntax Example	Abstract Syntax Reference
BlockDefinition Diagram		**SysML::Blocks::Block** **UML4SysML::Package**
Block		**SysML::Blocks::ValueType**
ValueType		**SysML::Blocks::ValueType**

Enumeration		**UML4SysML::Enumeration**
PropertySpecific Type		**SysML::Blocks:: PropertySpecificType**
AbstractDefinition		**UML4SysML::Classifier with isAbstract equal true**

StereotypeProperty Compartment		**UML4SysML::Stereotype**
Behavior Compartment		**SysML::Blocks::Block**
Namespace Compartment		**SysML::Blocks::Blocks**
Structure Compartment		**SysML::Blocks::Blocks**
BoundReference		**SysML::Blocks::Blocks,** **SysML::Blocks::BoundReference,** **SysML::Blocks::EndPathMultiplicity**

Unit		**UML4SysML::InstanceSpecification**
QuantityKind		**UML4SysML::InstanceSpecification**
InstanceSpecification		**UML4SysML::InstanceSpecification**
InstanceSpecification		**UML4SysML::InstanceSpecification**
InstanceSpecification		**UML4SysML::InstanceSpecification**
InstanceSpecification		**UML4SysML::InstanceSpecification**
Namespace Compartment		**SysML::Blocks::Blocks**

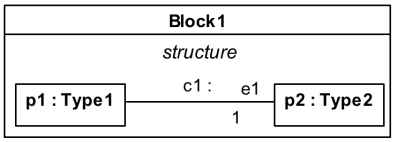
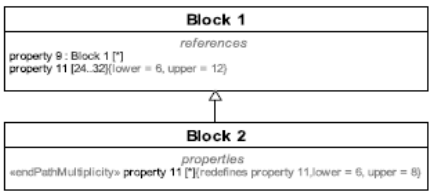
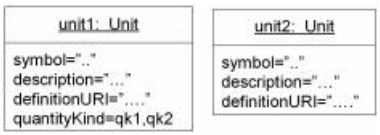
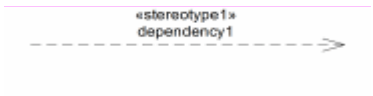
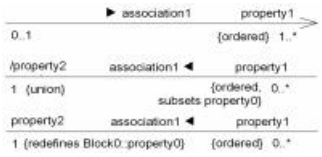
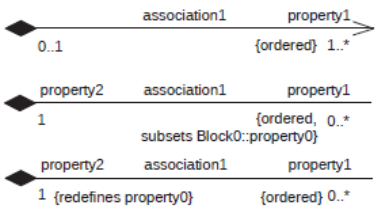
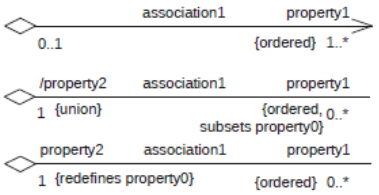
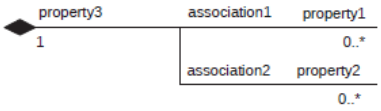
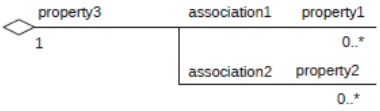
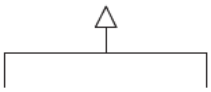
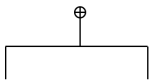
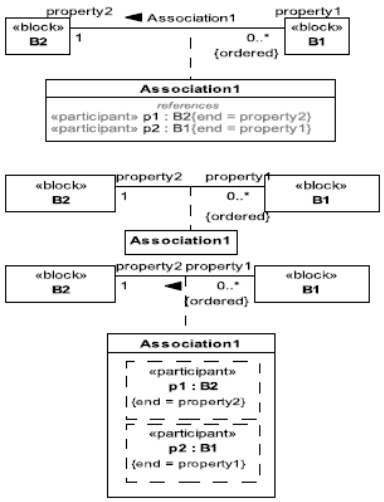
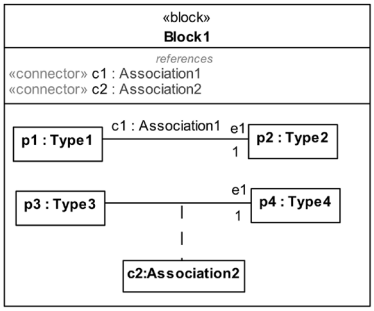
Structure Compartment		**SysML::Blocks::Blocks**
BoundReference		**SysML::Blocks::Blocks,** **SysML::Blocks::BoundReference,** **SysML::Blocks::EndPathMultiplicity**
Unit		**UML4SysML::InstanceSpecification**

Table 8-2: Graphical paths defined in Block Definition diagrams

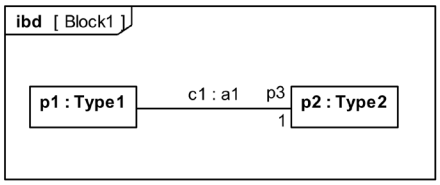
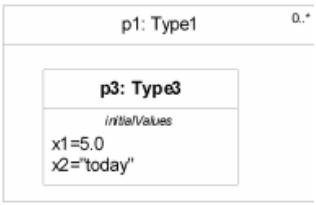
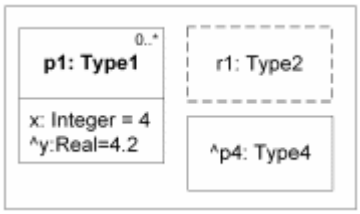
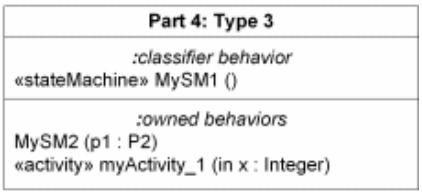
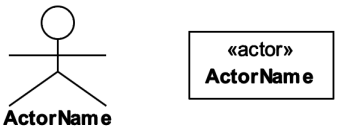
Element Name	Concrete Syntax Example	Abstract Syntax Reference
Dependency		**UML4SysML::Dependency**
Reference Association		**UML4SysML::Association and UML4SysML::Property with aggregationKind = none**

Element Name	Concrete Syntax Example	Abstract syntax Reference
PartAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = composite**
SharedAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = shared**
MultibranchPart Associations		**UML4SysML::Association and UML::Kernel::Property with aggregationKind = composite**
MultibranchShared Associations		**UML4SysML::Association and UML::Kernel::Property with aggregationKind = shared**
Generalization		**UML4SysML::Generalization**
Multibranch Generalization		**UML4SysML:Generalization**
GeneralizationSet		**UML4SysML:: GeneralizationSet**

Element Name	Concrete Syntax Example	Abstract Syntax Reference
BlockNamespace Containment		**UML4SysML::Class::nestedClassifier**
ParticipantProperty		**UML4SysML:: Property,** **UML4SysML:: AssociationClass**
ConnectorProperty		**UML4SysML:: Property,** **UML4SysML:: Connector**

8.2.2 Internal Block Diagram

Table 8-3: Graphical nodes defined in Internal Block diagrams

Element Name	Concrete Syntax Example	Abstract Syntax Reference
InternalBlockDiagram		**SysML::Blocks::Block**
Property	  	**UML4SysML::Property**
ActorPart		**SysML::Blocks::PartProperty typed by UML4SysML::Actor**

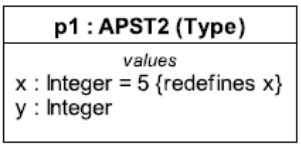
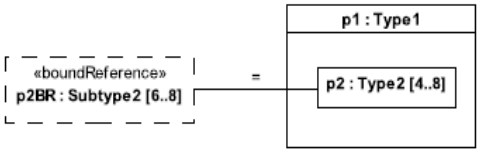
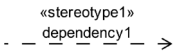
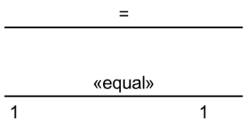
Element Name	Concrete Syntax Example	Abstract Syntax Reference
PropertySpecificType		**SysML::Blocks:: PropertySpecificType**
BoundReference		**SysML::Blocks::BoundReference**

Table 8-4: Graphical paths defined in internal Block diagrams

Element Name	Concrete Syntax Example	Abstract Syntax Reference
InternalBlockDiagram		**UML4SysML::Dependency**
BindingConnector		**UML4SysML::Connector**
BidirectionalConnector		**UML4SysML::Connector**
UnidirectionalConnector		**UML4SysML::Connector**

8.3 UML Extensions

8.3.1 Diagram Extensions

8.3.1.1 Block Definition Diagram

A block definition diagram is based on the UML class diagram, with restrictions and extensions as defined by SysML.

8.3.1.1.1 Block and Value Type Definitions

A SysML Block defines a collection of features to describe a system or other element of interest. A SysML ValueType defines values that may be used within a model. SysML blocks are based on UML classes, as extended by UML composite structures. SysML value types are based on UML data types. Diagram extensions for SysML blocks and value types are described by other subheadings of this sub clause.

8.3.1.1.2 Default «block» stereotype on unlabeled box

If no stereotype keyword appears within a definition box on a block definition diagram (including any stereotype property compartments), then the definition is assumed to be a SysML block, exactly as if the «block» keyword had appeared before the name in the top compartment of the definition.

8.3.1.1.3 Labeled compartments

SysML allows blocks to have multiple compartments, each optionally identified with its own compartment name. The compartments may partition the features shown according to various criteria. Some standard compartments are defined by SysML itself, and others can be defined by the user using tool-specific facilities. Compartments may appear in any order. SysML defines two additional compartments, namespace and structure compartments, which may contain graphical nodes rather than textual constraint or feature definitions. See separate sub clauses for a description of these compartments.
Compartment names shall comply with the following notation:
Shown in italics, where permitted by the font in use.

1. Centered
2. All lower case
3. Words separated by spaces

8.3.1.1.4 Behavior compartment

A compartment with the label “classifier behavior” or “owned behaviors” may appear as part of a block definition to list the classifier behavior or owned behaviors, respectively. This compartment may contain text representations of any kind of behavior.
Behaviors represented in this compartment are shown as a text string of the form:
<name> ‘(’ [<parameter-list>] ‘)’ [‘:’ [<return-type-list>]] [<behavior-constraint>]
where:

- <name> is the name of the Behavior.
- <parameter-list> is a list of Parameters of the Behavior in the format defined in UML.
- <return-type-list> is list of types, multiplicities, and other properties of parameters with return direction.

  <return-type-list> ::= <return-type-mult-prop> [‘,‘ <return-type-mult-prop>] *

<return-type-mult-prop> :=

<return-type> [‘[‘ <multiplicity-range> ‘]’] [‘{‘ <param-prop-list> ‘}’]]

(see UML for definition of <multiplicity-range>)

<param -prop-list> ::= <param -prop> [‘,’ <param -prop>]*

<param -prop> ::= ‘ordered’ | ‘unordered’ | ‘unique’ | ‘nonunique’ | ‘seq’ | ‘sequence’

- <behavior-constraint> is a constraint that applies to the behavior.

Other syntax defined by UML can be included, such as for applied stereotypes or the behavior's metaclass as a keyword before the name (for example «stateMachine»).

8.3.1.1.5 Constraints compartment

SysML defines a special form of compartment, with the label “constraints,” which may contain one or more constraints owned by the block. A constraint owned by the block may be shown in this compartment using the standard text-based notation for a constraint, consisting of a string enclosed in brace characters. The use of a compartment to show constraints is optional. The note-based notation, with a constraint shown in a note box outside the block and linked to it by a dashed line, may also be used to show a constraint owned by a block.

A constraints compartment may also contain declarations of constraint properties owned by the block. A constraint property is a property of the block that is typed by a ConstraintBlock, as defined in Clause 10. Only the declaration of the constraint property may be shown within the compartment, not the details of its parameters or binding connectors that link them to other properties.

8.3.1.1.6 Namespace compartment

A compartment with the label “namespace” may appear as part of a block definition to show blocks that are defined in the namespace of a containing block. This compartment may contain any of the graphical elements of a block definition diagram. All blocks or other named elements defined in this compartment belong to the namespace of the containing block.

Because this compartment contains graphical elements, a wider compartment than typically used for feature definitions may be useful. Since the same block can appear more than once in the same diagram, it may be useful to show this compartment as part of a separate definition box than a box that shows only feature compartments. Both namespace and structure compartments, which may both need a wide compartment to hold graphical elements, could also be shown within a common definition box.

8.3.1.1.7 Structure compartment

A compartment with the label “structure” may appear as part of a block definition to show connectors and other internal structure elements for the block being defined. This compartment may contain any of the graphical elements of an internal block diagram.

Because this compartment contains graphical elements, a wider compartment than typically used for feature definitions may be useful. Since the same block can appear more than once in the same diagram, it may be useful to show this compartment as part of a separate definition box than a box that shows only feature compartments. Both namespace and structure compartments, which may both need a wide compartment to hold graphical elements, could also be shown within a common definition box.

8.3.1.1.8 BoundReference compartment

A compartment with the label “bound references” may appear as part of a block definition to show properties with the BoundReference stereotype applied. The properties omit the “«boundReference»” prefix.

8.3.1.1.9 Receptions compartment

A compartment with the label “receptions” may appear as part of a block definition to show signal receptions. The “«signal»” keyword is optional in this compartment.

8.3.1.1.10 Default multiplicities

SysML defines defaults for multiplicities on the ends of specific types of associations. A part or shared association has a default multiplicity of [0..1] on the black or white diamond end. A unidirectional association has a default multiplicity of 1 on its target end. These multiplicities may be assumed if not shown on a diagram. To avoid confusion, any multiplicity other than the default should always be shown on a diagram.

8.3.1.1.11 Property-specific type

The notation for properties typed by a property-specific type shows the name of the most specific generalization of the property-specific type that is not a property-specific type (or nothing if there is no generalization) between parentheses after the name of the property-specific type (or after the colon if the property-specific type has no name).

The keyword for PropertySpecificType is «pst».

8.3.1.1.12 Unit Notation

Units on value properties

Value properties can optionally display the unit’s symbol in parentheses if value type has a unit defined.

If no unit symbol is defined, then the unit name can optionally be displayed.

<vpname> ":" <valueTypename> [" (" <unitSymbol | unitName> ")"]

e.g., distance:Length (m)

8.3.1.1.13 Units on values

Any ValueSpecification can optionally display the unit's symbol if it has a type which is a ValueType.

If ValueSpecification has no type and it is used as a value of a slot, then it takes the unit from defining feature type.

If ValueSpecification has no type and it is used as a default value of a value property, it takes the unit from that property type.

If no unit symbol is defined, then the unit name may be displayed.

<value> [" " <unitSymbol | unitName>]

e.g., distance:Length = 10 m

8.3.1.2 Internal Block Diagram

An internal block diagram is based on the UML composite structure diagram, with restrictions and extensions as defined by SysML.

8.3.1.2.1 Property types

Four general categories of properties of blocks are recognized in SysML: parts, references, value properties, and constraint properties. (See 8.3.2.4 for definitions of these property types.) A part or value property is always shown on an internal block diagram with a solid-outline box. A reference property is shown by a dashed-outline box, consistent with

UML. Ports are special cases of properties, and have a variety of notations as defined in Clause 9, “Ports and Flows.” Constraint properties and their parameters also have their own notations as defined in Clause 10, “Constraint Blocks.”

8.3.1.2.2 Block reference in diagram frame

The diagram heading name for an internal block diagram (the string contained in the tab in the upper-left-hand corner of the diagram frame) shall identify the name of a SysML block as its modelElementName. (See Annex A for the definition of a diagram heading name including the modelElementName component.) All the properties and connectors that appear inside the internal block diagram belong to the block that is named in the diagram heading name.

8.3.1.2.3 Compartments on internal properties

SysML permits any property shown on an internal block diagram to also show compartments within the property box. These compartments may be given standard or user-customized labels just as on block definitions. All features shown within these compartments shall match those of the block or value type that types the property. An unlabeled compartment on an internal property box is by default a structure compartment. A behavior compartment label and content shall match the corresponding behavior compartment of the block that types the part. A compartment with the label “classifier behavior” or “owned behaviors” may contain the classifier behavior or owned behaviors of the block that types the part which will then appear as specified in 8.3.3.1.4, Behavior compartment.

The label of any compartment shown on the property box that displays contents belonging to the type of the property is shown with a colon character (“:”) preceding the compartment label. The compartment name is otherwise the same as it would appear on the type on a block definition diagram.

8.3.1.2.4 Compartments on a diagram frame

SysML permits compartments to be shown across the entire width of the diagram frame on an internal block diagram. These compartments shall always follow an initial compartment that always shows the internal structure of a referenced block. These compartments may have all the same contents as could be shown on a block definition diagram for the block defined at the top level of the diagram frame.

8.3.1.2.5 Property path name

A property name shown inside or outside the property box may take the form of a multi-level name. This form of name references a nested property accessible through a sequence of intermediate properties from a referencing context. The name of the referenced property is built by a string of names separated by “.”, resulting in a form of path name that identifies the property in its local context. A colon and the type name for the property may optionally be shown following the dotted name string. If any of the properties named in the path name string identifies a reference property, the property box is shown with a dashed-outline box, just as for any reference property on an internal block diagram.

This notation is purely a notational shorthand for a property that could otherwise be shown within a structure of nested property boxes, with the names in the dotted string taken from the name that would appear at each level of nesting. In other words, the internal property shown with a path name in the left-hand side of
Figure 8-1 is equivalent to the innermost nested box shown at the right.

If the property has no name, the property’s type name can be used instead.

e.g., car:Engine:Cylinder:Piston.length
car.e.c.p.length

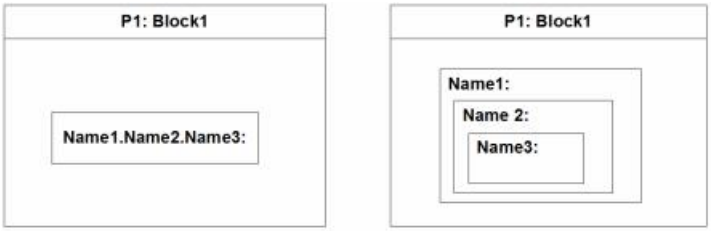


Figure 8-1: Nested property reference

8.3.1.2.6 Nested connector end

Connectors may be drawn that cross the boundaries of nested properties to connect to properties within them. The connector is owned by the most immediate block that owns both ends of the connector. A NestedConnectorEnd stereotype of a UML ConnectorEnd is automatically applied to any connector end that is nested more than one level deep within a containing context.

Use of nested connector ends does not follow strict principles of encapsulation of the parts or other properties that a connector line may cross. The need for nested connector ends can be avoided if additional properties can be added to the block at each containing level. Nested connector ends are available for cases where the introduction of these intermediate properties is not feasible or appropriate.

The ability to connect to nested properties within a containing block requires that multiple levels of decomposition be shown on the same diagram.

8.3.1.2.7 Property-specific type

The notation for properties typed by a property-specific type shows the name of the most specific generalization of the property-specific type that is not a property-specific type (or nothing if there is no generalization) between parentheses after the name of the property-specific type (or after the colon if the property-specific type has no name).

8.3.1.2.8 Initial values compartment

A compartment with a label of “initialValues” may be used to show values of properties belonging to a containing block. These values override any default values that may have been previously specified on these properties on their originally defining block. Initial value compartments may be specified within nested properties, which then apply only in the particular usage context defined by the outermost containing block.

Values are specified in an initialValues compartment by lines in the form <property-name> = <value-specification> or <property-name> : <type> = <value-specification>, each line of which specifies the initial value for one property owned either by the block that types the property or by any of its supertypes. This portion of concrete syntax is the same as may be shown for values within the UML instance specification notation, but this is the only element of UML InstanceSpecification notation that may be shown in an initial values compartment. See 8.3.2.4 for details of how values within initialValues compartments are represented in the SysML metamodel.

8.3.1.2.9 Default multiplicities

SysML defines default multiplicities of 1 on each end of a connector. These multiplicities may be assumed if not shown on a diagram. To avoid confusion, any multiplicity other than the default should always be shown on a diagram.

8.3.1.3 UML Diagram Elements not Included in SysML Block Definition Diagrams

The supported variety of notations for associations and association annotations has been reduced to simplify the burden of teaching, learning, and interpreting SysML diagrams for the systems engineering user. Notational and metamodel support for n-ary associations and qualified associations has been excluded from SysML. The use of navigation arrowheads on an association has been simplified by excluding the case of arrowheads on both ends, and requiring that such an association always be shown without arrowheads on either end. An “X” on a single end of an association to indicate that an end is not navigable has similarly been dropped, as has the use of a small filled dot at the end of an association to indicate that the end is owned by the associated classifier.

UML allows representing owned attributes using an association-like notation (see UML 2.5 Figure 9-12). This notation does not show any multiplicity on the opposite end since there is no corresponding property. In such a case, the multiplicity on the opposite side of the association-like notation is the less constrained possible. That is: "0..1" if the attribute has a composite aggregation and "0..*" otherwise. However, it is a common practice for modelers to assume that, when not shown, the multiplicity of an association end is the default multiplicity (i.e., "1..1"). This might create ambiguity because there is no practical way to distinguish between the association-like notation and a "true" association. The association-like notation is excluded from SysML to avoid it.

The use of a «primitive» keyword on a value type definition (which in UML specifies the PrimitiveType specialization of UML DataType) is not supported. Whether or not a value type definition has internal structure can be determined from the value type itself.

8.3.1.4 UML Diagram Elements not Included in SysML Internal Block Diagrams

The UML Composite Structure diagram has many notations not included in the subset defined in this clause. Other SysML clauses add some of these notations into the supported contents of an internal block diagram.

8.3.2 Stereotypes

8.3.2.1 Package Blocks

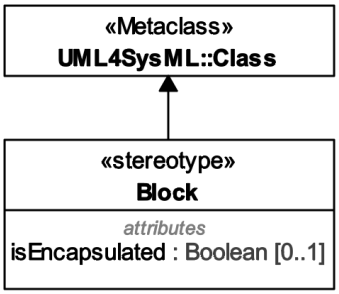


Figure 8-2: Abstract syntax extensions for SysML blocks

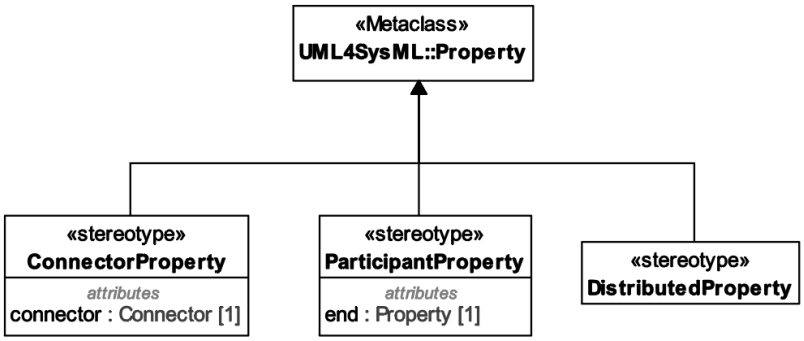


Figure 8-3: Abstract syntax extensions for SysML properties

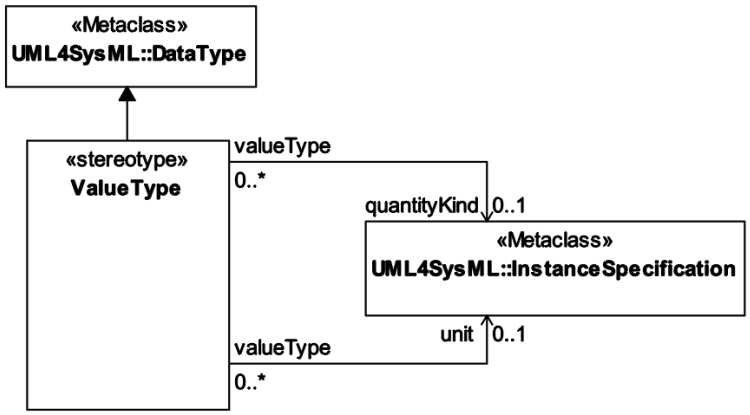


Figure 8-4: Abstract syntax extensions for SysML value types

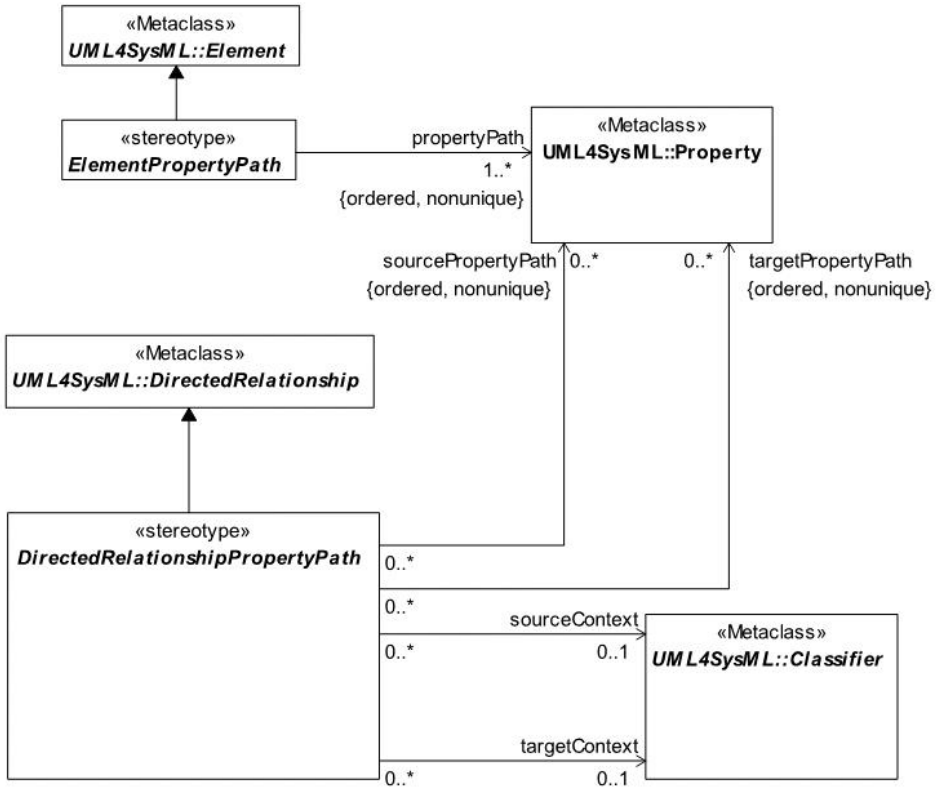


Figure 8-5: Abstract syntax extensions for SysML property paths

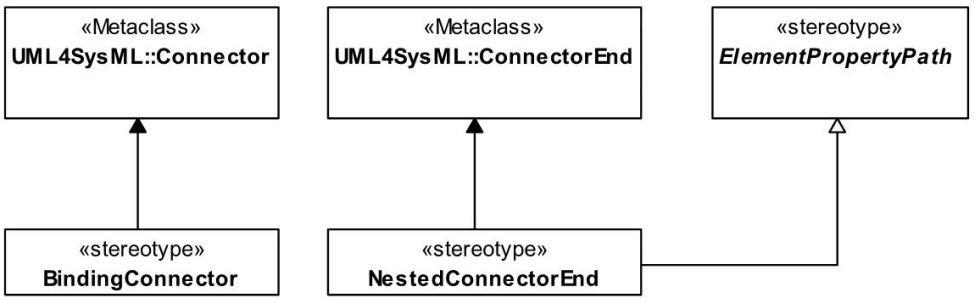


Figure 8-6: Abstract syntax extensions for SysML connector ends

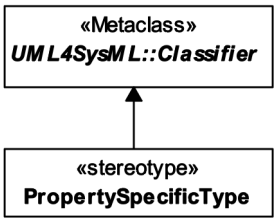


Figure 8-7: Abstract syntax extensions for SysML property-specific types

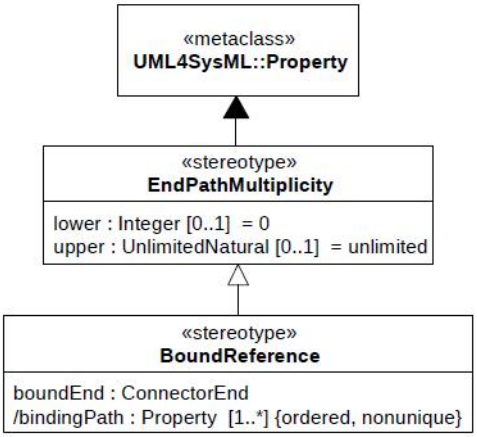


Figure 8-8: Abstract syntax extensions for SysML bound references

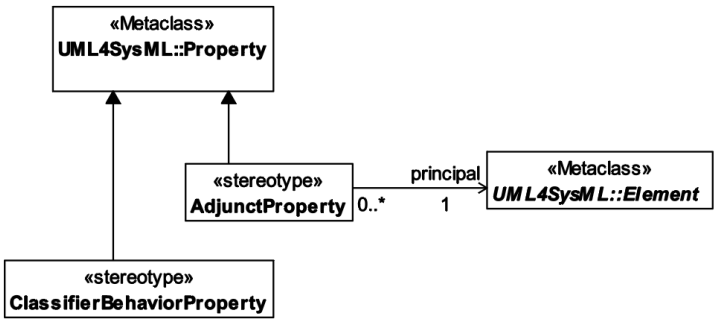


Figure 8-9: Abstract syntax extensions for SysML adjunct properties and classifier behavior properties

8.3.2.2 AdjunctProperty

Description

The AdjunctProperty stereotype can be applied to properties to constrain their values to the values of connectors typed by association blocks, call actions, object nodes, variables, parameters, interaction uses, and submachine states. The values of connectors typed by association blocks are the instances of the association block typing a connector in the block having the stereotyped property. The values of call actions are the executions of behaviors invoked by the behavior having the call action and the stereotyped property (see 11.3.1.1.1, Notation for more about this use of the stereotype). The values of object nodes are the values of tokens in the object nodes of the behavior having the stereotyped property (see 11.3.1.4.1, Notation for more about this use of the stereotype). The values of variables are those assigned by executions of activities that have the stereotyped property. The values of parameters are those assigned by executions of behaviors that have the stereotyped property. The keyword «adjunct» before a property name indicates the property is stereotyped by AdjunctProperty.

Association Ends

- base_Property : Property [1]
- principal : Element [1]
  Gives the element that determines the values of the property.

Constraints

- 10_multiplicity_same_or_less_restrictive
  Properties with AdjunctProperty applied that have a Variable or Parameter as principal shall have a lower multiplicity the same as or lower than the lower multiplicity of their principal, and an upper multiplicity the same as or higher than the upper multiplicity of their principal

```
self.principal.oclIsKindOf(UML::MultiplicityElement) implies self.base_Property.lower
<= self.principal.oclAsType(UML::MultiplicityElement).lower and
self.base_Property.upper >= self.principal.oclAsType(UML::MultiplicityElement).upper
```

- 11_submachine_and_interactionuse_composite_and _compatible_type
  Properties with AdjunctProperty applied that have an InteractionUse or submachine State as principal shall be composite and be typed by the interaction or state machine invoked by the interaction use or submachine State or one of their generalizations.

```
self.principal.oclIsKindOf(UML::InteractionUse) or
self.principal.oclIsKindOf(UML::State) implies let behavior: UML::Behavior = if
self.principal.oclIsKindOf(UML::InteractionUse) then
self.principal.oclAsType(UML::InteractionUse).refersTo else
self.principal.oclAsType(UML::State).submachine endif in if behavior.oclIsUndefined()
then self.base_Property.type->isEmpty() else self.base_Property.type->notEmpty() and
behavior->closure(generalization)->including(behavior)
->includes(self.base_Property.type) endif
```

- 1_principal_kind
  The principal of an applied AdjunctProperty shall be a Connector, CallAction, ObjectNode, Variable, Parameter, submachine State, or InteractionUse.

```
self.principal.oclIsKindOf(UML::Connector) or
self.principal.oclIsKindOf(UML::CallAction) or
self.principal.oclIsKindOf(UML::ObjectNode) or
```

```
self.principal.oclIsKindOf(UML::Variable) or
self.principal.oclIsKindOf(UML::Parameter) or
self.principal.oclIsKindOf(UML::InteractionUse) or
(self.principal.oclIsKindOf(UML::State) and
self.principal.oclAsType(UML::State).isSubmachineState)
```

- 2_same_name
  Properties to which AdjunctProperty applied shall have the same name as the principal, if the principal is a NamedElement.

```
self.principal.oclIsKindOf(UML::NamedElement) implies self.base_Property.name =
self.principal.oclAsType(UML::NamedElement).name
```

- 3_connector_and_callaction_composite
  Properties with AdjunctProperty applied that have a Connector or CallAction as principal shall be composite.

```
self.principal.oclIsKindOf(UML::Connector) or
self.principal.oclIsKindOf(UML::CallAction) implies self.base_Property.isComposite()
```

- 4_same_owner
  Properties with AdjunctProperty applied shall be owned by an element that owns the principal, at least indirectly, or one of that elements specializations.

```
let owners: Set(UML::Element) = self.principal->closure(owner) in let
specializations: Set(UML::Element) = UML::Classifier.allInstances()->select(c |
c->closure(general)->intersection(owners)->notEmpty()) in owners
->union(specializations)->includes(self.base_Property.owner)
```

- 5_compatible_type
  Properties with AdjunctProperty applied that have as principal a Connector, ObjectNode, Variable, or Parameter shall have the same type as the principal or one of that types generalizations.

```
self.principal.oclIsKindOf(UML::Connector) or
self.principal.oclIsKindOf(UML::Variable) or
self.principal.oclIsKindOf(UML::Parameter) implies let principal_type:
UML::Classifier = if self.principal.oclIsKindOf(UML::Connector) then
self.principal.oclAsType(UML::Connector).type else
self.principal.oclAsType(UML::TypedElement).type.oclAsType(UML::Classifier) endif in
principal_type->closure(general)->including(principal_type)
->includes(self.base_Property.type)
```

- 6_connector_principal_associationblock
  Connectors that are principals of an applied AdjunctProperty shall have association blocks as types

```
self.principal.oclIsKindOf(UML::Connector) implies let type: UML::Association =
self.principal.oclAsType(UML::Connector).type in Block.allInstances().base_Class
->includes(type)
```

- 7_adjunctproperty_connectorproperty_consistent
  AdjunctProperty and ConnectorProperty applied to the same property shall have the same values for principal and connector, respectively.

```
AdjunctProperty.allInstances()->forAll(ap | let cp: ConnectorProperty =
ConnectorProperty.allInstances()->any(base_Property=ap.base_Property) in (not
cp.oclIsUndefined()) implies cp.connector = ap.principal)
```

- 8_callAction_composite_and_consitent_type
  Properties with AdjunctProperty applied that have a CallAction as principal shall be composite and be typed by the behavior invoked by the call action or one of that behaviors generalizations (for CallOperationAct ions, this shall generalize all behaviors that might be dispatched), and an upper multiplicity of one if the CallAction invokes a nonreentrant behavior.

```
self.principal.oclIsKindOf(UML::CallAction) implies if
self.principal.oclIsKindOf(UML::CallOperationAction) then let called:
Set(UML::Behavior) =
self.principal.oclAsType(UML::CallOperationAction).operation.method in if called
->isEmpty() then self.base_Property.type->isEmpty() else self.base_Property.type
->notEmpty() and called->forAll(b | b.general->including(b)
->includes(self.base_Property.type)) endif else let called: UML::Behavior = if
self.principal.oclIsKindOf(UML::CallBehaviorAction) then
self.principal.oclAsType(UML::CallBehaviorAction).behavior else
self.principal.oclAsType(UML::StartObjectBehaviorAction).behavior() endif in if
called.oclIsUndefined() then self.base_Property.type.oclIsUndefined() else let
behaviors: Set(UML::Behavior) = called
->closure(generalization).oclAsType(UML::Behavior)->including(called)->asSet() in
self.base_Property.type->notEmpty() and behaviors->includes(self.base_Property.type)
endif endif
```

- 9_objectnode_multiplicity
  Properties with AdjunctProperty applied that have an ObjectNode as principal shall have a lower multiplicity of zero and an upper multiplicity the same as or higher than the upperBound of the ObjectNode.

```
self.principal.oclIsKindOf(UML::ObjectNode) implies self.base_Property.lower = 0 and
self.base_Property.upper >=
self.principal.oclAsType(UML::ObjectNode).upperBound.unlimitedValue()
```

8.3.2.3 BindingConnector

Description

A Binding Connector is a connector which specifies that the properties at both ends of the connector have equal values. If the properties at the ends of a binding connector are typed by a ValueType, the connector specifies that the instances of the properties shall hold equal values, recursively through any nested properties within the connected properties. If the properties at the ends of a binding connector are typed by a Block, the connector specifies that the instances of the properties shall refer to the same block instance. As with any connector owned by a SysML Block, the ends of a binding connector may be nested within a multi-level path of properties accessible from the owning block. The NestedConnectorEnd stereotype is used to represent such nested ends just as for nested ends of other SysML connectors.

Association Ends

- base_Connector : Connector [1]

Constraints

- 1_compatible_types
  The two ends of a binding connector shall have either the same type or types that are compatible so that equality of their values can be defined.

```
self.base_Connector.end->at(1).role.type.conformsTo(self.base_Connector.end
->at(2).role.type) or self.base_Connector.end
->at(2).role.type.conformsTo(self.base_Connector.end->at(1).role.type)
```

8.3.2.4 Block

Description

A Block is a modular unit that describes the structure of a system or element. It may include both structural and behavioral features, such as properties and operations, which represent the state of the system and behavior that the system may exhibit. Some of these properties may hold parts of a system, which can also be described by blocks that type the properties. Properties without types do not restrict the instances that can be values of the properties, as if they had the most general type possible. A block may include a structure of connectors between its properties to indicate how its parts or other properties relate to one another.

SysML blocks provide a general-purpose capability to describe the architecture of a system. They provide the ability to represent a system hierarchy, in which a system at one level is composed of systems at a more basic level. They can describe not only the connectivity relationships between the systems at any level, but also quantitative values or other information about a system.

SysML does not restrict the kind of system or system element that may be described by a block. Any reusable form of description that may be applied to a system or a set of system characteristics may be described by a block. Such reusable descriptions, for example, may be applied to purely conceptual aspects of a system design, such as relationships that hold between parts or properties of a system.

Connectors owned by SysML blocks may be used to define relationships between parts or other properties of the same containing block. Connectors can be typed by associations, which can specify more detail about the links between parts or other properties of a system, along with the types of the connected properties. Associations can also be blocks, and when used to type connectors give relationships their own interconnected parts and other properties. Connectors without types do not restrict the way the connected properties are linked together, as if they had the most general type possible. Connectors have both structural and behavioral functions, which can be used together or separately. Connectors as structure specify links between parts or other properties of a system. Connectors as behavior specify communication and item flow between parts or other properties. Connected properties can be linked without specifying communication and item flow, or can specify communication and item flow without specifying a particular kind of link, or both.

SysML excludes variations of associations in UML in which navigable ends can be owned directly by the association. In SysML, navigation is equivalent to a named property owned directly by a block. The only form of an association end that SysML allows an association to own directly is an unnamed end used to carry an inverse multiplicity of a reference property. This unnamed end provides a metamodel element to record an inverse multiplicity, to cover the specific case of a unidirectional reference that defines no named property for navigation in the inverse direction. SysML enforces its equivalence of navigation and ownership by means of constraints that the block stereotype enforces on the existing UML metamodel.

SysML establishes four basic classifications of properties belonging to a SysML Block or ValueType. A property typed by a SysML Block that has composite aggregation is classified as a part property, except for the special case of a constraint property. Constraint properties are further defined in Clause 10. A port is another category of property, as further defined in Section 9. A property typed by a Block that does not have composite aggregation is classified as a reference property. A property typed by a SysML ValueType is classified as a value property, and always has composite aggregation. Part, reference, value, and constraint properties may be shown in block definition compartments with the labels "parts," "references," "values," and "constraints" respectively. Properties of any type may be shown in a "properties" compartment or in additional compartments with user-defined labels.

On a block definition diagram, a part property is shown by a black diamond symbol on an association. As in UML, an instance of a block may be included in at most one instance of a block at a time, though possibly as a value of more than one part property of the containing block. A part property holds instances that belong to a larger whole. Typically, a part-whole relationship means that certain operations that apply to the whole also apply to each of the parts. For example, if a whole represents a physical object, a change in position of the whole could also change the position of each of the parts. A property of the whole such as its mass could also be implied by its parts. Operations and relationships that apply to parts typically apply transitively across all parts of these parts, through any number of levels. A particular application domain may establish its own interpretation of part-whole relationships across the blocks defined in a particular model, including the definition of operations that apply to the parts along with the whole. For software objects, a typical interpretation is that delete, copy, and move operations apply across all parts of a composite object.

SysML also supports properties with shared aggregation, as shown by a white diamond symbol on an association. Like UML, SysML defines no specific semantics or constraints for properties with shared aggregation, but particular models or tools may interpret them in specific ways.

In addition to the form of default value specifications that SysML supports on properties of a block (with an optional "=" <value-specification> string following the rest of a property definition), SysML supports an additional form of value specification for properties using initialValue compartments on an internal block diagram (see Internal Block Diagram on page 4). An entire tree of context-specific values can be specified on a containing block to carry values of nested properties as shown on an internal block diagram.

Context-specific values are represented in the SysML metamodel by means of the InstanceValue subtype of UML ValueSpecification. Selected slots of UML instance specifications referenced by these instance values carry the individual values shown in initialValue compartments.

If a property belonging to a block has a specification of initial values for any of the properties belonging to its type, then the default value of that property shall be a UML InstanceValue element. This element shall reference a UML InstanceSpecification element created to hold the initial values of the individual properties within its usage context.

Selected slots of the referenced instance specification shall contain value specifications for the individual property values specified in a corresponding initialValues compartment. If a value of a property is shown by a nested property box with its own initialValues compartment, then the slot of the instance specification for the containing property shall hold a new InstanceValue element. Selected slots of the instance specification referenced by this value shall contain value specifications for any nested initial values, recursively through any number of levels of nesting. A tree of instance values referencing instance specifications, each of which may in turn hold slots carrying instance values, shall exist until self-contained value specifications are reached at the leaf level.

Attributes

- isEncapsulated : Boolean [0..1]

  If true, then the block is treated as a black box; a part typed by this black box can only be connected via its ports or directly to its outer boundary. If false, or if a value is not present, then connections can be established to elements of its internal structure via deep-nested connector ends.

Association Ends

- base_Class : Class [1]

Constraints

- 1_associations_binary
  For an association in which both ends are typed by blocks, the number of ends shall be exactly two.

```
UML::Association.allInstances()->select(a| a.memberEnd->forAll(e| e.type->notEmpty()
and Block.allInstances().base_Class->includes(e.type)))->forAll(a | a.memberEnd
->size()=2)
```

- 2_connectors_binary
  The number of ends of a connector owned by a block shall be exactly two. (In SysML, a binding connector is not typed by an association, so this constraint is not implied entirely by the preceding constraint.)

```
self.base_Class.ownedConnector->forAll(c | c.end->size()=2 )
```

- 5_uml_connector_constraint_removed
  The following constraint under 11.8, "Connector" in the UML 2 standard is removed by SysML: "The ConnectableElements attached as roles to each ConnectorEnd owned by a Connector must be roles of the Classifier that owned the Connector, or they must be ports of such roles.

```
-- Cannot be expressed in OCL
```

- 6_valueproperties_composite
  If a property owned by a SysML Block or SysML ValueType is typed by a SysML ValueType, then the aggregation attribute of the property shall be "composite."

```
self.base_Class.ownedAttribute->select(a| ValueType.allInstances().base_DataType
->includes(a.type))->forAll(a|a.isComposite())
```

- 7_composition_acyclic
  Within an instance of a SysML Block, the values of any property with composite aggregation (aggregation = composite) shall not contain the block in any of its own properties that also have composite aggregation, or within any unbroken chain of properties that all have composite aggregation. (Within an instance of a SysML Block, the instances of properties with composite aggregation shall form an acyclic graph.)

```
self.base_Class->closure(part-
>select(p|p.type.oclIsKindOf(UML::Class)).type.oclAsType(UML::Class))
->excludes(self.base_Class)
```

- 8_specializations_are_blocks
  Any classifier that specializes a Block shall also have the Block stereotype or one of its specializations applied.

```
UML::Classifier.allInstances()->select(c | c.general->includes(self.base_Class))
->forAll(c | Block.allInstances()->includes(c))
```

- 9_uml constraint_removed
  The following constraint under 11.8,"ConnectorEnd" in the UML 2 standard is removed by SysML: "[3] The property held in self.partWithPort must not be a Port."

```
-- cannot be expressed in OCL
```

8.3.2.5 BoundReference

Description

The BoundReference stereotype can be applied to properties that have binding connectors, to highlight their usage as constraining other properties. The bound end of the stereotype is a connector end of one of the binding connectors,

opposite the stereotyped property. The binding path includes the property at the bound end, and before that, the property path of the bound end, if it is a nested connector end.

The type of stereotyped property constrains the type of the values of the bound properties. The multiplicity of the stereotyped property constrains the number of values of the bound properties, which is the total number of values reached by navigation through property paths of nested connector ends, if any. The multiplicities at the end of path can be constrained, because bound references are end path multiplicities (see 8.3.2.11, EndPathMultiplicity).

Properties with BoundReference applied and upper multiplicity greater than one are ordered, with values ordered according to when they are reached in navigating the binding path (and how they are ordered within their blocks), and non-unique, to support paths that lead to or pass through the same object.

Generalizations

- EndPathMultiplicity (from Blocks)

Attributes

- /bindingPath : Property [1..*]
  Gives the propertyPath of the NestedConnectorEnd applied, if any, to the boundEnd, appended to the role of the boundEnd.
  (derived)
- boundEnd : ConnectorEnd [1]
  Gives a connector end of a binding connector opposite to the end linked to the stereotyped property, or linked to a property that generalizes the stereotyped one through redefinition.

Constraints

- 1_bindingconnector_end
  Properties to which BoundReference is applied shall be the role of a connector end of at least one binding connector, or generalized by such a property through redefinition

```
BindingConnector.allInstances().base_Connector.end.role->exists(r |
r=self.base_Property or self.base_Property->closure(redefinedElement)->includes(r))
```

- 2_opposite_bindingconnector_end
  The value of boundEnd shall be a connector end of a binding connector, as identified in constraint 1, opposite the property, as identified in constraint 1.

```
let opposite: UML::ConnectorEnd = BindingConnector.allInstances().base_Connector.end
->any(e | e.role=self.base_Property or self.base_Property->closure(redefinedElement)
->includes(e.role)) in self.boundEnd = opposite.owner.oclAsType(UML::Connector).end
->any(e | e<>opposite)
```

- 3_navigable
  The role of boundEnd shall be a property accessible by navigation from instances of the block owning the property to which BoundReference is applied, but shall not be the property to which BoundReference is applied, or one that it is related to by redefinition.

```
self.base_Property.association->notEmpty() and self.boundEnd.definingEnd->notEmpty()
and self.base_Property.association.navigableOwnedEnd
->includes(self.boundEnd.definingEnd)
```

- 4_propertypath_consistency
  The last value of bindingPath shall be the role of boundEnd, and the other values shall be the propertyPath of the NestedConnectorEnd applied to boundEnd, if any.

```
self.boundEnd = self.bindingPath->last() and (let nce: NestedConnectorEnd =
NestedConnectorEnd.allInstances()->any(n| n.base_ConnectorEnd=self.boundEnd) in nce
->oclIsUndefined() or self.bindingPath->subSequence(1, self.bindingPath->size()-1) =
nce.propertyPath)
```

- 5_reference_or_valueproperty
  Properties to which BoundReference is applied shall either be reference properties or value properties.

```
ValueType.allInstances().base_DataType->includes(self.base_Property.type) or not
self.base_Property.isComposite()
```

- 6_ordered_nonunique
  Properties with BoundReference applied that have an upper multiplicity greater than one shall be ordered and non-unique.

```
self.base_Property.upper > 1 implies self.base_Property.isOrdered and not
self.base_Property.isUnique
```

- 7_cannot_redefine_boundreference
  BoundReferences shall not be applied to properties that are related by redefinition to other properties with BoundReference applied.

```
self.base_Property.redefinedElement->notEmpty() implies
BoundReference.allInstances().base_Property
->excludesAll(self.base_Property.redefinedElement)
```

- 8_notbounded_to_itslef
  The binding connector identified in constraint 1 shall not have the same property on both ends, or properties related by redefinition.

```
let e1: UML::ConnectorEnd = self.boundEnd.owner.oclAsType(UML::Connector).end->at(1)
in let e2: UML::ConnectorEnd = self.boundEnd.owner.oclAsType(UML::Connector).end
->at(2) in e1.role <> e2.role and (e1.role.oclIsKindOf(UML::Property) and
e2.role.oclIsKindOf(UML::Property) implies
e1.role.oclAsType(UML::Property).redefinedElement->excludes(e2.role) and
e2.role.oclAsType(UML::Property).redefinedElement->excludes(e1.role))
```

8.3.2.6 ClassifierBehaviorProperty

Description

The ClassifierBehaviorProperty stereotype can be applied to properties to constrain their values to be the executions of classifier behaviors. The value of properties with ClassifierBehaviorProperty applied are the executions of classifier behaviors invoked by instantiation of the block that owns the stereotyped property or one of its specializations.

Association Ends

- base_Property : Property [1]

Constraints

- 1_owner_classifierbehavior
  ClassifierBehaviorProperty shall only be applied to properties owned (not inherited) by blocks that have classifier behaviors.

```
Block.allInstances().base_Class->exists(c | c.ownedAttribute
->includes(self.base_Property) and c.classifierBehavior->notEmpty())
```

- 2_composite
  Properties to which ClassifierBehaviorProperty is applied shall be composite

```
self.base_Property.isComposite
```

- 3_typed_by_classifierbehavior
  Properties to which ClassifierBehaviorProperty applied shall be typed by the classifier behavior of their owning block or a generalization of the classifier behavior.

```
let clBehavior: UML::Behavior =
self.base_Property.owner.oclAsType(UML::Class).classifierBehavior in
self.base_Property.type->notEmpty() and clBehavior->closure(general)
->including(clBehavior)->includes(self.base_Property.type)
```

8.3.2.7 ConnectorProperty

Description

Connectors can be typed by association classes that are stereotyped by Block (association blocks, see ParticipantProperty on page 68). These connectors specify instances of the association block created within the instances of the block that owns the connector. The values of a connector property are instances of the association block created due to the connector referred to by the connector property.

A connector property can optionally be shown in an internal block diagram with a dotted line from the connector line to a rectangle notating the connector property. The keyword «connector» before a property name indicates the property is stereotyped by ConnectorProperty.

Attributes

- connector : Connector [1]
  A connector of the block owning the property on which the stereotype is applied.

Association Ends

- base_Property : Property [1]

Constraints

- 1_block_property
  ConnectorProperty shall only be applied to properties of classes stereotyped by Block.

```
Block.allInstances().base_Class->exists(c | c.ownedAttribute
->includes(self.base_Property))
```

- 2_owned_or_inherited
  The connector attribute of the applied stereotype shall refer to a connector owned or inherited by a block owning the property on which the stereotype is applied.

```
let owner: UML::Class = Block.allInstances().base_Class->any(c | c.ownedAttribute
->includes(self.base_Property)) in owner->closure(general)
->select(oclIsKindOf(UML::Class)).oclAsType(UML::Class).ownedConnector->flatten()
->includes(self.connector)
```

- 3_composite
  The aggregation of a property stereotyped by ConnectorProperty shall be composite.

```
self.base_Property.isComposite
```

- 4_typed_by_associationblock
  The type of the connector referred to by a connector attribute shall be an association class stereotyped by Block.

```
Block.allInstances().base_Class->exists(c | c.oclIsKindOf(UML::AssociationClass) and
self.connector.type = c)
```

- 5_same_name
  A property stereotyped by ConnectorProperty shall have the same name and type as the connector referred to by the connector attribute.

```
self.base_Property.name = self.connector.name
```

8.3.2.8 DirectedRelationshipPropertyPath

Description

The DirectedRelationshipPropertyPath stereotype based on UML DirectedRelationship enables directed relationships to identify their sources and targets by a multi-level path of properties accessible from context blocks for the sources and targets. Context blocks are typically the owner of the first property in the path of properties, but can be specializations of the owner to limit the scope of the relationship.

Association Ends

- base_DirectedRelationship : DirectedRelationship [1]
- sourceContext : Classifier [0..1]
  Gives the context for sourcePropertyPath to begin from.
- sourcePropertyPath : Property [0..*]
  A series of properties that identifies the source of the directed relationship in the context of the block specified by the sourceContext property. The ordering of properties is from a property of the sourceContext block, through a property of each intermediate block that types the preceding property, ending in a property with a type that owns or inherits the source of the directed relationship. The source is not included in the propertyPath list. The same property might appear more than once because a block can own a property with the same or specialized block as a type.
- targetContext : Classifier [0..1]
  Gives the context for targetPropertyPath to begin from.
- targetPropertyPath : Property [0..*]
  A series of properties that identifies the target of the directed relationship in the context of the block specified by

the targetContext property. The ordering of properties is from a property of the targetContext block, through a property of each intermediate block that types the preceding property, ending in a property with a type that owns or inherits the target of the directed relationship. The target is not included in the propertyPath list. The same property might appear more than once because a block can own a property with the same or specialized block as a type.

Constraints

- 1_sourcecontext_iif_property
  sourceContext shall have a value when source is a property, otherwise it shall not have a value

```
self.base_DirectedRelationship.source->exists(s | s.oclIsKindOf(UML::Property)) xor
self.sourceContext->isEmpty()
```

- 2_targetcontext_iif_property
  targetContext shall have a value when target is a property, otherwise it shall not have a value.

```
self.base_DirectedRelationship.source->exists(s | s.oclIsKindOf(UML::Property)) xor
self.sourceContext->isEmpty()
```

- 3_sourcepropertypath_implies_property
  source shall be a property when sourcePropertyPath has a value.

```
self.sourcePropertyPath->notEmpty() implies self.base_DirectedRelationship.source
->forAll(s | s.oclIsKindOf(UML::Property))
```

- 4_targetpropertypath_implies_property
  target shall be a property when targetPropertyPath has a value.

```
self.targetPropertyPath->notEmpty() implies self.base_DirectedRelationship.target
->forAll(s | s.oclIsKindOf(UML::Property))
```

- 5_sourcecontext_owns_sourcepath_first
  The property in the first position of the sourcePropertyPath list, if any, shall be owned by the sourceContext or one of its generalizations

```
self.sourcePropertyPath->notEmpty() implies self.sourceContext.allAttributes()
->includes(self.sourcePropertyPath->first())
```

- 6_targetcontext_owns_targetpath_first
  The property in the first position of the targetPropertyPath list, if any, shall be owned by the targetContext or one of its generalizations.

```
self.targetPropertyPath->notEmpty() implies self.targetContext.allAttributes()
->includes(self.targetPropertyPath->first())
```

- 7_path_and_owners_consistency
  The property at each successive position of the sourcePropertyPath and targetPropertyPath, following the first position, shall be owned by the Block or ValueType that types the property at the immediately preceding position, or a generalization of the Block or ValueType.

```
(self.sourcePropertyPath->size() >1 implies self.sourcePropertyPath->subSequence(2,
self.sourcePropertyPath->size())->forAll(p | let pp: UML::Property =
self.sourcePropertyPath->at(self.sourcePropertyPath->indexOf(p)-1) in let owners:
Set(UML::Classifier) = pp.type.oclAsType(UML::Classifier)
->including(pp.type.oclAsType(UML::Classifier)) in owners->includes(p.owner))) and
```

```
(self.targetPropertyPath->size() >1 implies self.targetPropertyPath->subSequence(2,
self.targetPropertyPath->size())->forAll(p | let pp: UML::Property =
self.targetPropertyPath->at(self.targetPropertyPath->indexOf(p)-1) in let owners:
Set(UML::Classifier) = pp.type.oclAsType(UML::Classifier)
->including(pp.type.oclAsType(UML::Classifier)) in owners->includes(p.owner)))
```

- 8_sourcepath_last_type_owns_source
  The type of the property at the last position of the sourcePropertyPath list shall own or inherit the source of the stereotyped directed relationship.

```
self.sourcePropertyPath->notEmpty() implies self.sourcePropertyPath
->last().type.oclAsType(UML::Classifier).allAttributes()
->includesAll(self.base_DirectedRelationship.source)
```

- 9_targetpath_last_type_owns_target
  The type of the property at the last position of the targetPropertyPath list shall own or inherit the target of the stereotyped directed relationship.

```
self.targetPropertyPath->notEmpty() implies self.targetPropertyPath
->last().type.oclAsType(UML::Classifier).allAttributes()
->includesAll(self.base_DirectedRelationship.target)
```

8.3.2.9 DistributedProperty

Description

DistributedProperty is a stereotype of Property used to apply a probability distribution to the values of the property. Specific distributions should be defined as subclasses of the DistributedProperty stereotype with the operands of the distributions represented by properties of those stereotype subclasses. A sample set of probability distributions that could be applied to value properties is given in E.7.

Association Ends

- base_Property : Property [1]

Constraints

- 1_block_or_valuetype
  The DistributedProperty stereotype shall only be applied to properties of classifiers stereotyped by Block or ValueType.

```
Block.allInstances().base_Class.oclAsType(UML::Classifier)
->union(ValueType.allInstances().base_DataType)->includes(self.base_Property.owner)
```

8.3.2.10 ElementPropertyPath

Description

The ElementPropertyPath stereotype based on UML Element enables elements to identify other elements by a multi-level path of properties accessible from a context block. The context block is described in specializations of ElementPropertyPath.

Association Ends

- base_Element : Element [1]
- propertyPath : Property [1..*]
  A series of properties that identifies elements in the context of a block described in specializations of ElementPropertyPath. The ordering of properties is from a property of the context block, through a property of each intermediate block that types the preceding property, ending in a property with a type that owns or inherits the fully nested property. The fully nested property is not included in the propertyPath list, but is given by the element to which the ElementPropertyPath is applied in a way described in specializations of ElementPropertyPath. The same property might appear more than once because a block can own a property with the same or
  specialized block as a type.

Constraints

- 1_path_consistency
  The property at each successive position of the propertyPath attribute, following the first position, shall be owned by the Block or ValueType that types the property at the immediately preceding position, or a generalization of the Block or ValueType.

```
self.propertyPath->size() >1 implies self.propertyPath->subSequence(2,
self.propertyPath->size())->forAll(p | let pp: UML::Property = self.propertyPath
->at(self.propertyPath->indexOf(p)-1) in let owners: Set(UML::Classifier) =
pp.type.oclAsType(UML::Classifier)->including(pp.type.oclAsType(UML::Classifier)) in
owners->includes(p.owner))
```

8.3.2.11 EndPathMultiplicity

Description

The EndPathMultiplicity stereotype can be applied to properties that are related by redefinition to properties that have BoundReference applied. The lower and upper properties of the stereotype give the minimum and maximum number of values, respectively, of the property at the bound end of the related bound reference, for each object reached by navigation along its binding path.

Attributes

- lower : Integer [0..1]
  Gives the minimum number of values of the property at the end of the related bindingPath, for each object reached by navigation along the bindingPath from an instance of the block owning the property to which EndPathMultiplicity is applied.
- upper : UnlimitedNatural [0..1]
  Gives the maximum number of values of the property at the end of the related bindingPath, for each object reached by navigation along the bindingPath from an instance of the block owning the property to which EndPathMultiplicity is applied.

Association Ends

- base_Property : Property [1]

Constraints

- 1_redefinition
  Properties to which EndPathMultiplicity is applied shall be related by redefinition to a property to which BoundReference is applied.

```
self.base_Property.redefinedProperty->notEmpty() and
BoundReference.allInstances().base_Property->exists(p |
self.base_Property.redefinedProperty->includes(p))
```

- 2_non_negative
  endPathLower shall be non-negative.

```
self.lower >= 0
```

8.3.2.12 NestedConnectorEnd

Description

The NestedConnectorEnd stereotype of UML ConnectorEnd extends a UML ConnectorEnd so that the connected property may be identified by a multi-level path of accessible properties from the block that owns the connector. The propertyPath inherited from ElementPropertyPath gives a series of properties that identifies the connected property in the context of the block that owns the connector. The ordering of properties is from a property of the block that owns the connector, through a property of each intermediate block that types the preceding property, ending in a property with a type that owns or inherits the property that is the role of the connector end (the property that the connector graphically attaches to at that end). The property that is the role of the connector end is not included in the propertyPath list.

Generalizations

- ElementPropertyPath (from Blocks)

Association Ends

- base_ConnectorEnd : ConnectorEnd [1]
  (redefines: ElementPropertyPath::base_Element)

Constraints

- 1_propertypath_first_owned_by_connector_owner
  The first property in propertyPath shall be owned by the block that owns the connector, or one of the blocks generalizations.

```
let owningBlock: UML::Class =
self.base_ConnectorEnd.owner.oclAsType(UML::Connector).owner.oclAsType(UML::Class)
in (not owningBlock.oclIsUndefined()) and owningBlock->closure(general)
->including(owningBlock)->includes(self.propertyPath->first().owner)
```

- 2_propertypath_last_type_owns_role
  The type of the property at the last position of the propertyPath list shall own or inherit the role property of the stereotyped connector end

```
let type: UML::Classifier =
self.propertyPath->last().type.oclAsType(UML::Classifier) in
not type.oclIsUndefined()) andtype.allFeatures()
->includes(self.base_ConnectorEnd.role)
```

8.3.2.13 ParticipantProperty

Description

The Block stereotype extends Class, so it can be applied to any specialization of Class, including Association Classes. These are informally called "association blocks." An association block can own properties and connectors, like any other block. Each instance of an association block can link together instances of the end classifiers of the association.

To refer to linked objects and values of an instance of an association block, it is necessary for the modeler to specify which (participant) properties of the association block identify the instances being linked at which end of the association. The value of a participant property on an instance (link) of the association block is the value or object at the end of the link corresponding to this end of the association.

Participant properties can be the ends of connectors owned by an association block. The association block can be the type of multiple other connectors to reuse the same internal structure for all the connectors. The keyword «participant» before a property name indicates the property is stereotyped by ParticipantProperty. They are always the same as the corresponding association end type.

Attributes

- end : Property [1]
  A member end of the association block owning the property on which the stereotype is applied.

Association Ends

- base_Property : Property [1]

Constraints

- 1_associationblock
  ParticipantProperty shall only be applied to properties of association classes stereotyped by Block.
  ```
  self.base_Property.class.oclIsKindOf(UML::AssociationClass) and
  Block.allInstances().base_Class->includes(self.base_Property.class)
  ```

- 2_memberend
  ParticipantProperty shall not be applied to properties that are member ends of an association.
  ```
  UML::Association.allInstances().memberEnd->flatten()->excludes(self.base_Property)
  ```

- 3_aggregationkind_none
  The aggregation of a property stereotyped by ParticipantProperty shall be none.
  ```
  self.base_Property.aggregation = UML::AggregationKind::none
  ```

- `4_end_owner`
  The end attribute of the applied stereotype shall refer to a member end of the association block owning the property on which the stereotype is applied.
  `self.base_Property.association.memberEnd->includes(self.end)`

- `5_same_type`
  A property stereotyped by ParticipantProperty shall have the same type as the property referred to by the end attribute.
  `self.base_Property.type = self.end.type`

- `6_multiplicity_1`
  A property to which the ParticipantProperty is applied shall have a multiplicity of 1.
  `self.base_Property.lower = 1 and self.base_Property.upper = 1`

8.3.2.14 PropertySpecificType

Description

The PropertySpecificType stereotype can be applied to classifiers that type exactly one property and that are owned by the owner of that property. Classifiers with this stereotype applied shall be generalized by at most one other classifier.

Instances of a property-specific type are exactly those that are values of the property it types, in all instances of the property owner. Values are (de)classified under property-specific types as they are (removed from) added to the property they type:

- quantityKind : InstanceSpecification [0..1]
  A kind of quantity, represented by an InstanceSpecification classified by a kind of SysML QuantityKind, that may be stated by means
- Added values are classified as instances of the property-specific type.
- Removed values are:
  - Declassified as instances of the property-specific type.
  - Classified as instances of the most specific generalization of the property-specific type that is not a property-specific type, unless the instances are indirectly classified by that generalization already. If there is no such property-specific type, unless the instances are indirectly classified by that generalization already. If there is no such property-specific type, removed values are not additionally classified.
- This enables values of the property to:
  - Support more features than they would when they are not values of the property.
  - Have redefined or constrained features only while they are values of the property.

Association Ends

base Classifier : Classifier [1]

Constraints

1_only_one_property
A classifier to which the PropertySpecificType stereotype is applied shall be referenced as the type of one and only one property.

```
UML::Property.allInstances()->select(p | p.type = self.base_Classifier)->size() = 1
```

8.3.2.15 ValueType

Description

A ValueType defines types of values that may be used to express information about a system, but cannot be identified as the target of any reference. Since a value cannot be identified except by means of the value itself, each such value within a model is independent of any other, unless other forms of constraints are imposed.

Value types may be used to type properties, operation parameters, or potentially other elements within SysML. SysML defines ValueType as a stereotype of UML DataType to establish a more neutral term for system values that may never be given a concrete data representation. For example, the SysML "Real" ValueType expresses the mathematical concept of a real number, but does not impose any restrictions on the precision or scale of a fixed or floating-point representation that expresses this concept. More specific value types can define the concrete data representations that a digital computer can process, such as conventional Float, Integer, or String types.

SysML ValueType adds an ability to carry a unit of measure and quantity kind associated with the value. A quantity kind is a kind of quantity that may be stated in terms of defined units, but does not restrict the selection of a unit to state the value. A unit is a particular value in terms of which a quantity of the same quantity kind may be expressed. A SysML ValueType and its quantityKind establishes, via UML typing, the associative relationship between a particular "quantity" [VIM3-1.1] (modeled as a SysML value property typed by a ValueType) and a "kind of quantity" [VIM3-1.2] (the ValueType::quantityKind of the SysML value propertys type). This UML/SysML associative relationship reflects the terminological distinction made in VIM3 between the concepts of "quantity" [VIM3-1.1] and "kind-of-quantity" [VIM3-1.2] that "cannot be in a generic or partitive hierarchical relation to each other" [Dybkaer-2010].

A SysML ValueType may define its own properties and/or operations, just as for a UML DataType. See 8.3.2.4, Block for property classifications that SysML defines for either a Block or ValueType.

Association Ends

- base_DataType : DataType [1]
- quantityKind : InstanceSpecification [0..1]
  A kind of quantity, represented by an InstanceSpecification classified by a kind of SysML QuantityKind, that may be stated by means of units. A value type may optionally specify a quantity kind without any unit. Such a value type may be used to type a value specification to represent it in an abstract form independent of any specific units.

  Value types may be used to type properties, operation parameters, or potentially other elements within SysML. SysML defines ValueType as a stereotype of UML DataType to establish a more neutral term for system values that may never be given a concrete data representation. For example, the SysML "Real" ValueType expresses the mathematical concept of a real number, but does not impose any restrictions on the precision or scale of a fixed or floating-point representation that expresses this concept. More specific value types can define the concrete data representations that a digital computer can process, such as conventional Float, Integer, or String types.

SysML ValueType adds an ability to carry a unit of measure and quantity kind associated with the value. A quantity kind is a kind of quantity that may be stated in terms of defined units, but does not restrict the selection of a unit to state the value. A unit is a particular value in terms of which a quantity of the same quantity kind may be expressed. A SysML ValueType and its quantityKind establishes, via UML typing, the associative relationship between a particular "quantity" [VIM3-1.1] (modeled as a SysML value property typed by a ValueType) and a "kind of quantity" [VIM3-1.2] (the ValueType::quantityKind of the SysML value propertys type). This UML/SysML associative relationship reflects the terminological distinction made in VIM3 between the concepts of "quantity" [VIM3-1.1] and "kind-of-quantity" [VIM3- 1.2] that "cannot be in a generic or partitive hierarchical relation to each other" [Dybkaer-2010].

A SysML ValueType may define its own properties and/or operations, just as for a UML DataType. See 8.3.2.4, Block for property classifications that SysML defines for either a Block or ValueType.

- unit : InstanceSpecification [0..1]
  A unit, represented by an InstanceSpecification classified by a kind of SysML Unit, in terms of which the magnitudes of other quantities that have the same quantity kind can be stated.

Constraints

- 1_specializations_are_valuetypes
  Any classifier that specializes a ValueType shall also have the ValueType stereotype applied.

```
UML::Classifier.allInstances()->forAll(c | c.general->includes(self.base_DataType)
implies ValueType.allInstances().base_DataType->includes(c))
```

- 2_unit
  The unit of a ValueType, if any, shall be an InstanceSpecification classified by SysMLs Unit block in the UnitAndQuantityKind model library or a specialization of it.

```
self.unit->notEmpty() and self.unit.classifier->notEmpty() implies
self.unit.classifier->forAll(c |
c.oclIsKindOf(Libraries::UnitAndQuantityKind::Unit))
```

- 3_quantitykind
  The quantityKind of a ValueType, if any, shall be an InstanceSpecification classified by SysMLs QuantityKind block in the UnitAndQuantityKind model library or a specialization of it.

```
self.quantityKind->notEmpty() and self.quantityKind.classifier->notEmpty() implies
self.quantityKind.classifier->forAll(c |
c.oclIsKindOf(Libraries::UnitAndQuantityKind::QuantityKind))
```

8.3.3 Model Libraries

8.3.3.1 Package PrimitiveValueTypes

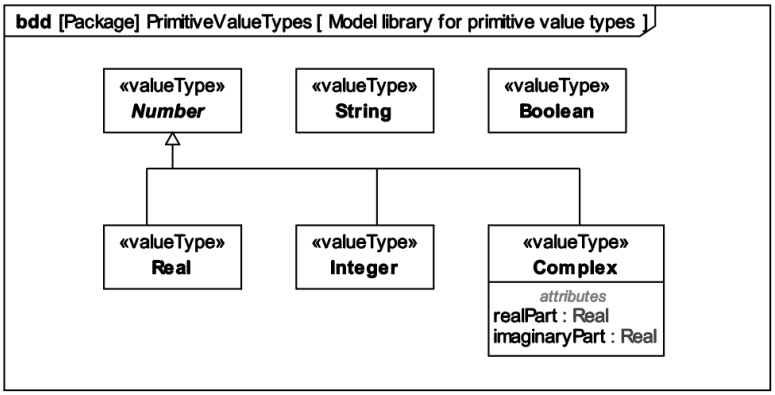


Figure 8-10: Model library for primitive value types

8.3.3.1.1 Boolean

Description

A Boolean value type consists of the predefined values true and false.

8.3.3.1.2 Complex

Description

A Complex value type represents the mathematical concept of a complex number. A complex number consists of a real part defined by a real number, and an imaginary part defined by a real number multiplied by the square root of -1. Complex numbers are used to express solutions to various forms of mathematical equations.

Generalizations

- Number (from PrimitiveValueTypes)

Attributes

- imaginaryPart : Real [1]
  A real number used to express the imaginary part of a complex number.
- realPart : Real [1]
  A real number used to express the real part of a complex number.

8.3.3.1.3 Integer

Description

An Integer value type represents the mathematical concept of an integer number. An Integer value type may be used to type values that hold negative or positive integer quantities, without committing to a specific representation such as a binary or decimal digits with fixed precision or scale.

Generalizations

- Number (from PrimitiveValueTypes)

8.3.3.1.4 Number

Description

Number is an abstract value type from which other value types that express concepts of mathematical numbers are specialized.

8.3.3.1.5 Real

Description

A Real value type represents the mathematical concept of a real number. A Real value type may be used to type values that hold continuous quantities, without committing a specific representation such as a floating point data type with restrictions on precision and scale.

Generalizations

- Number (from PrimitiveValueTypes)

8.3.3.1.6 String

Description

A String value type consists of a sequence of characters in some suitable character set. Character sets may include non-Roman alphabets and characters.

8.3.3.2 Package UnitAndQuantityKind

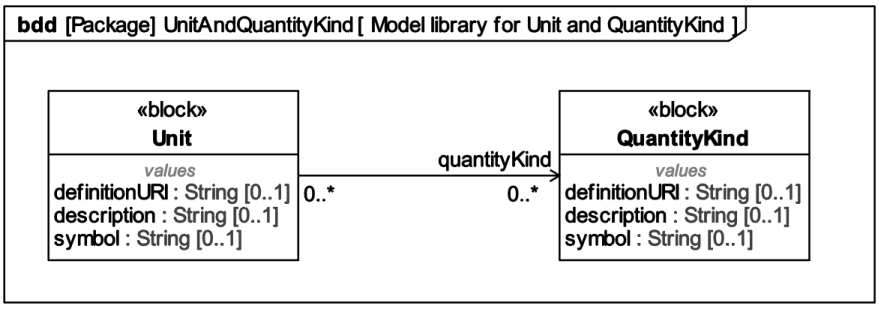


Figure 8-11: Model library for Unit and QuantityKind

QuantityKind

Description

A QuantityKind is a kind of quantity that may be stated by means of defined units. For example, the quantity kind of length may be measured by units of meters, kilometers, or feet. QuantityKind is defined as a non-abstract SysML Block defined in the SysML UnitAndQuantityKind model library. QuantityKind, or a specialization of it, classifies an InstanceSpecification to define a particular "kind-of-quantity" in the sense of an "aspect common to mutually comparable quantities" [VIM3-1.2], where a SysML value property is understood to correspond to the VIM concept of "quantity" defined as a "property of a phenomenon, body or substance, where the property has a magnitude that can be expressed as a number and a reference" [VIM3-1.1]. Modelers specialize QuantityKind as done in SysMLs QUDV model library or in a similar manner in other model libraries.

The definitionURI of an InstanceSpecification classified by a kind of QuantityKind identifies the particular "kind-of-quantity" [VIM3-1.2] that the InstanceSpecification represents. Two such InstanceSpecifications represent the same "kind-of-quantity" if and only if their definitionURIs have values and their values are equal. The only valid use of a QuantityKind instance is to be referenced by the quantityKind property of a ValueType or Unit.

See the non-normative model library in E.5 for an optional way to specify more comprehensive definitions of units and quantity kinds as part of systems of units and systems of quantities. The name of a QuantityKind, its definitionURI, or other means may be used to link individual quantity kinds to additional sources of documentation such as this optional model library.

Attributes

- definitionURI : String [0..1]
- description : String [0..1]
- symbol : String [0..1]

8.3.3.2.2 Unit

Description

QuantityKind is a kind of quantity that may be stated by means of defined units. For example, the quantity kind of length may be measured by units of meters, kilometers, or feet. QuantityKind is defined as a non-abstract SysML Block defined in the SysML UnitAndQuantityKind model library. QuantityKind, or a specialization of it, classifies an InstanceSpecification to define a particular "kind-of-quantity" in the sense of an "aspect common to mutually comparable quantities" [VIM3-1.2], where a SysML value property is understood to correspond to the VIM concept of "quantity" defined as a "property of a phenomenon, body or substance, where the property has a magnitude that can be expressed as a number and a reference" [VIM3-1.1]. Modelers specialize QuantityKind as done in SysMLs QUDV model library or in a similar manner in other model libraries.

The definitionURI of an InstanceSpecification classified by a kind of QuantityKind identifies the particular "kind-of-quantity" [VIM3-1.2] that the InstanceSpecification represents. Two such InstanceSpecifications represent the same "kind-of-quantity" if and only if their definitionURIs have values and their values are equal. The only valid use of a QuantityKind instance is to be referenced by the quantityKind property of a ValueType or Unit.

See the non-normative model library in E.5 for an optional way to specify more comprehensive definitions of units and quantity kinds as part of systems of units and systems of quantities. The name of a QuantityKind, its definitionURI, or

other means may be used to link individual quantity kinds to additional sources of documentation such as this optional model library.

A Unit is a quantity in terms of which the magnitudes of other quantities that have the same quantity kind can be stated. A unit often relies on precise and reproducible ways to measure the unit. For example, a unit of length such as meter may be specified as a multiple of a particular wavelength of light. A unit may also specify less stable or precise ways to express some value, such as a cost expressed in some currency, or a severity rating measured by a numerical scale.
Unit is defined as a non-abstract SysML Block defined in the SysML UnitAndQuantityKind model library. Unit, or a specialization of it, classifies an InstanceSpecification to define a particular "measurement unit" in the sense of a "real scalar quantity, defined and adopted by convention, with which any other quantity of the same kind can be compared to express the ratio of the two quantities as a number" [VIM3-1.9], where a SysML value property is understood to correspond to the VIM concept of "quantity" defined as a "property of a phenomenon, body or substance, where the property has a magnitude that can be expressed as a number and a reference" [VIM3-1.1]. Modelers specialize Unit as done in SysMLs QUDV model library or in a similar manner in other model libraries.

The definitionURI of an InstanceSpecification classified by a kind of Unit identifies the particular "measurement unit" [VIM3-1.9] that the InstanceSpecification represents. Two such InstanceSpecifications represent the same "measurement unit" if and only if their definitionURIs have values and their values are equal.

The only valid use of a Unit instance is to be referenced by the unit property of a ValueType stereotype.
See the non-normative model library in E.5 for an optional way to specify more comprehensive definitions of units and quantity kinds as part of systems of units and systems of quantities. The name of a Unit, its definitionURI, or other means may be used to link individual units to additional sources of documentation such as this optional model library.

Attributes

- definitionURI : String [0..1]
- description : String [0..1]
- symbol : String [0..1]

Association Ends

- quantityKind : QuantityKind [0..*]

8.4 Usage Examples

8.4.1 Wheel Hub Assembly

In Figure 8-12 a block definition diagram shows the blocks that comprise elements of a Wheel. The block property LugBoltJoint.torque has a specialization of DistributedProperty applied to describe the uniform distribution of its values. Examples of such distributions can be found in E.5. Connectors from the lugBoltJoints part go to nested parts, and use NestedConnectorEnd to specify the path of properties to reach those parts. For the threadedHole end of the connector going to part h, the property path is (hub). For the mountingHole end of the connector going to mountingHoles, the property path is (wheel, w). Similarly, the connector between the rim and bead parts has property paths (w) and (t) on its ends.

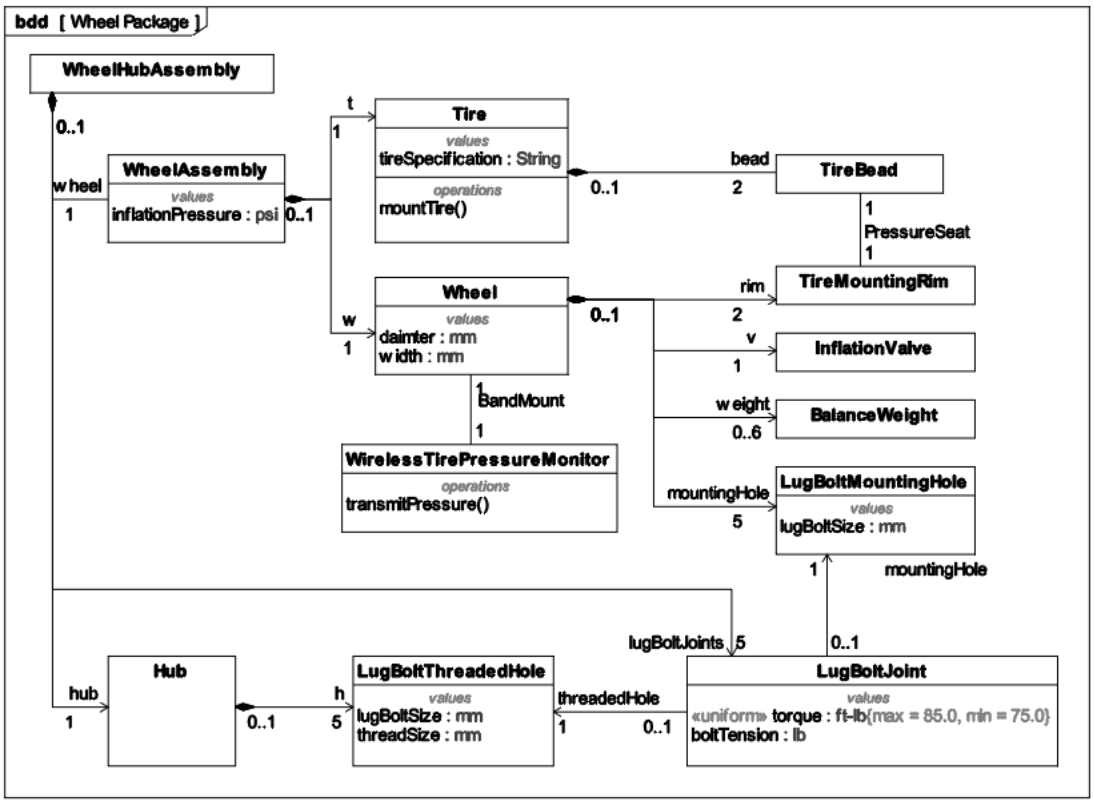


Figure 8-12: Block diagram for the Wheel Package

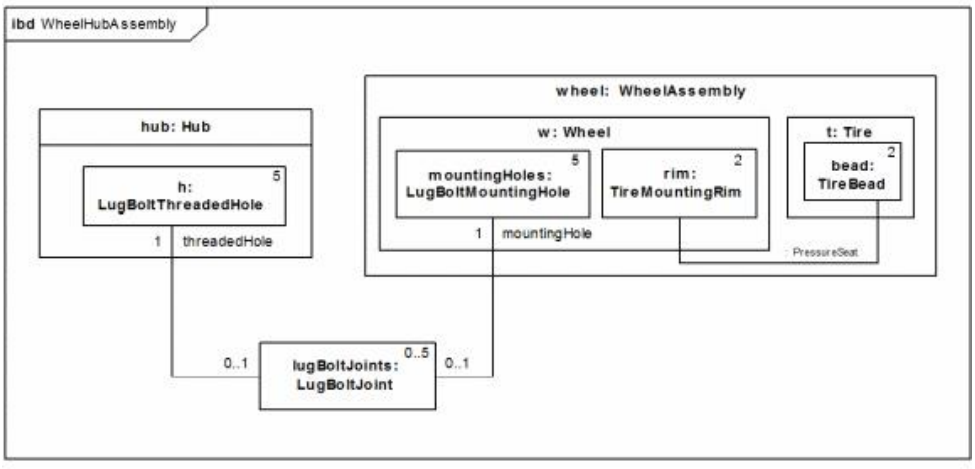


Figure 8-13: Internal Block Diagram for WheelHubAssembly

In Figure 8-13 an internal block diagram (ibd) shows how the blocks defined in the Wheel package are used. This ibd is a partial view that focuses on particular parts of interest and omits others from the diagram, such as the "v" InflationValve and "weight" BalanceWeight, which are also parts of a Wheel.

8.4.2 Example Value Type Definitions

In Figure 8-14, several value types that use standard units of measure from the International System of Units (SI) are defined to be available in the Example Value Type Definitions package. The value types in this package could be imported into other contexts for typing properties of SysML Blocks. Because a SysML Unit can already identify a type of quantity, or QuantityKind, that the unit measures, a value type only needs to identify the unit to identify a quantity kind as well. The value types in this example refer to units that are assumed to be defined in an imported package, such as the Model Library defined in E.6.

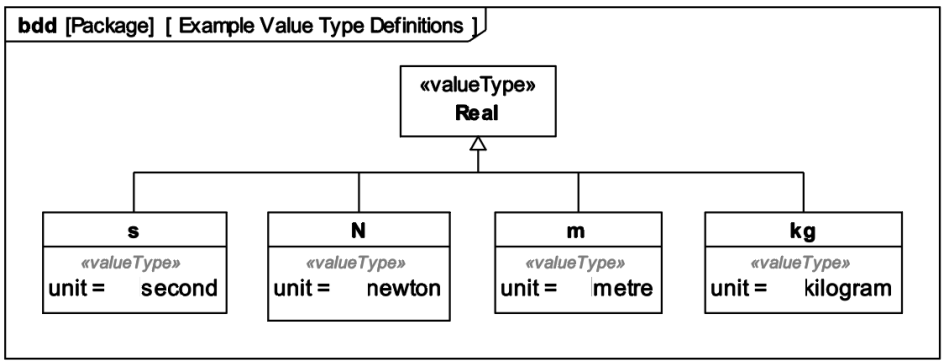


Figure 8-14: Defining Value Types with units of measure from the International System of Units (SI)

8.4.3 Design Configuration for SUV EPA Fuel Economy Test

SysML internal block diagrams may be used to specify blocks with unique identification and property values. Figure D.41 shows an example used to specify a unique vehicle with a vehicle identification number (VIN) and unique properties such as its weight, color, and horsepower. This concept is distinct from the UML concept of instance specifications in that it does not imply or assume any run-time semantic, and can also be applied to specify design configurations.

In SysML, one approach is to capture system configurations by creating a context for a configuration in the form of a context block. The context block may capture a unique identity for the configuration, and utilizes parts and initial value compartments to express property design values within the specification of a particular system configuration. Such a context block may contain a set of parts that represent the block instances in this system configuration, each containing specific values for each property. This technique also provides for configurations that reflect hierarchical system structures, where nested parts or other properties are assigned design values using initial value compartments. The following example illustrates the approach.

8.4.4 Water Delivery

Association blocks can be decomposed into connectors between properties of the associated blocks. These properties can be ports, as in the water delivery example in 9.4.5, Association and Port Decomposition.

8.4.5 Constraining Decomposition

Figure 8-15 shows an example decomposition for vehicles in a block definition diagram. Figure 8-16 shows the same decomposition in an internal block diagram that includes bound references. The binding connectors have nested connector ends, because they link inside the parts of the vehicle.

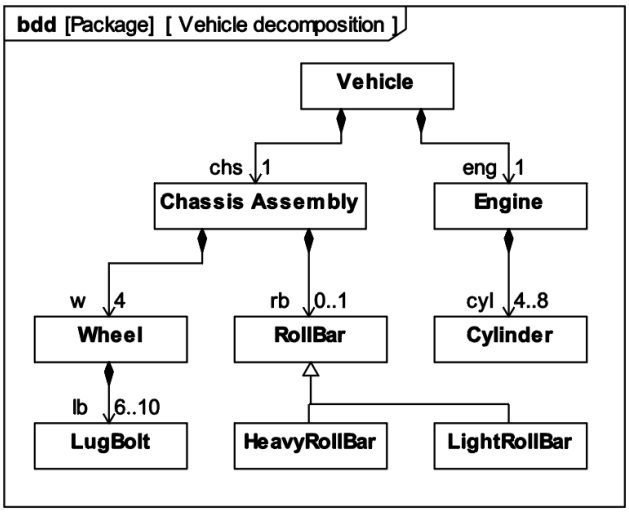


Figure 8-15: Vehicle Decomposition

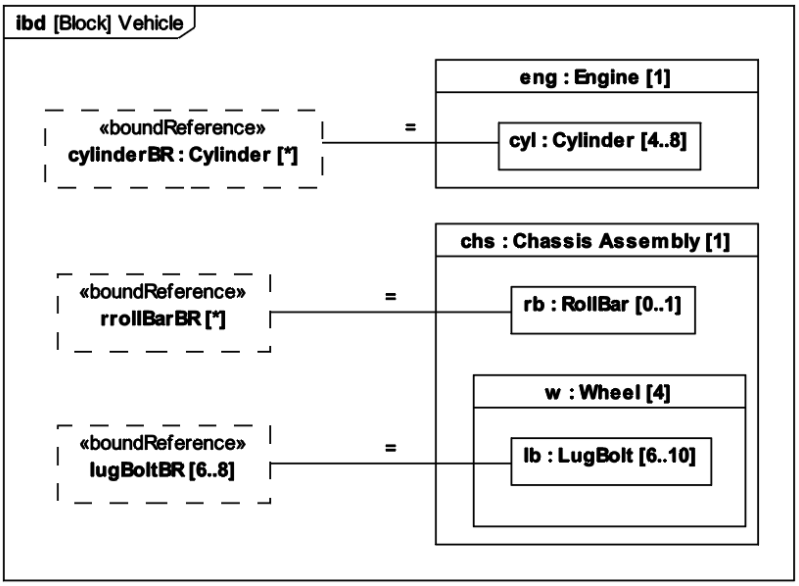


Figure 8-16: Vehicle internal structure

Figure 8-17 shows specializations for vehicles that restrict aspects of nested parts by redefining bound references. Paths for bound references are based on the property paths of the corresponding binding connectors. The general block on the top does not restrict the bound properties, except the total number of lug bolts is required to be between 24 and 32, rather than 24 and 40 as the associations in Figure 8-15 allow. The specialization on the lower left restricts the number of cylinders to four, requires a light roll bar, and a total of 24 lug bolts over all the wheels. The specialization on the lower right restricts the number of cylinders to between six and eight, rules out any roll bar, and limits lug bolts per wheel to between 6 and 7, by giving the end path upper and lower values.

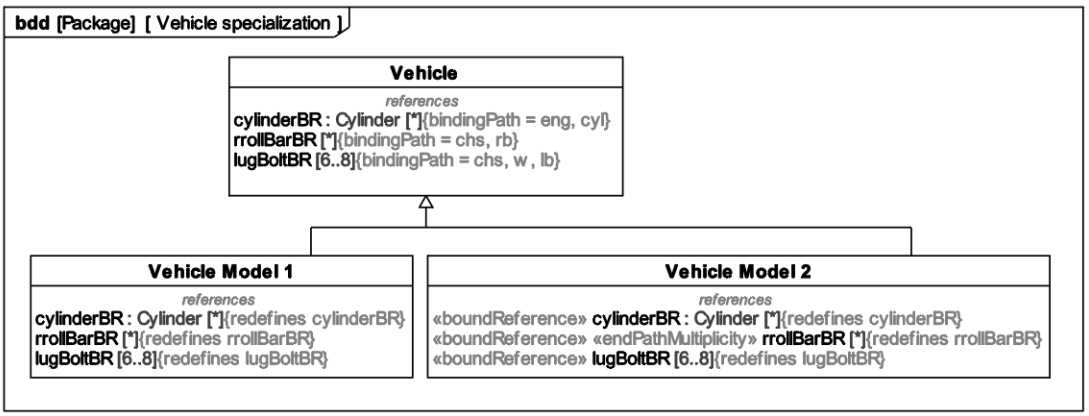


Figure 8-17: Vehicle specialization

8.4.6 Units and Quantity Kinds

The following shows a minimal example of definitions a Unit, QuantityKind, and ValueType based on them.

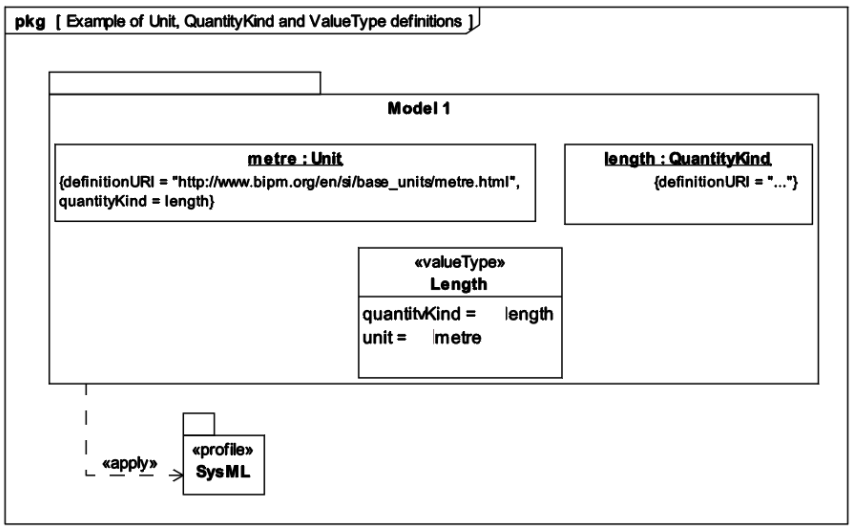


Figure 8-18: Example of Unit, QuantityKind and ValueType definitions

In terms of the UML4SysML metamodel and of the SysML profile, the following figure shows a partial account of the instance-level representation of the above example. This instance-level representation is important for model interchange, particularly across different implementations of SysML.

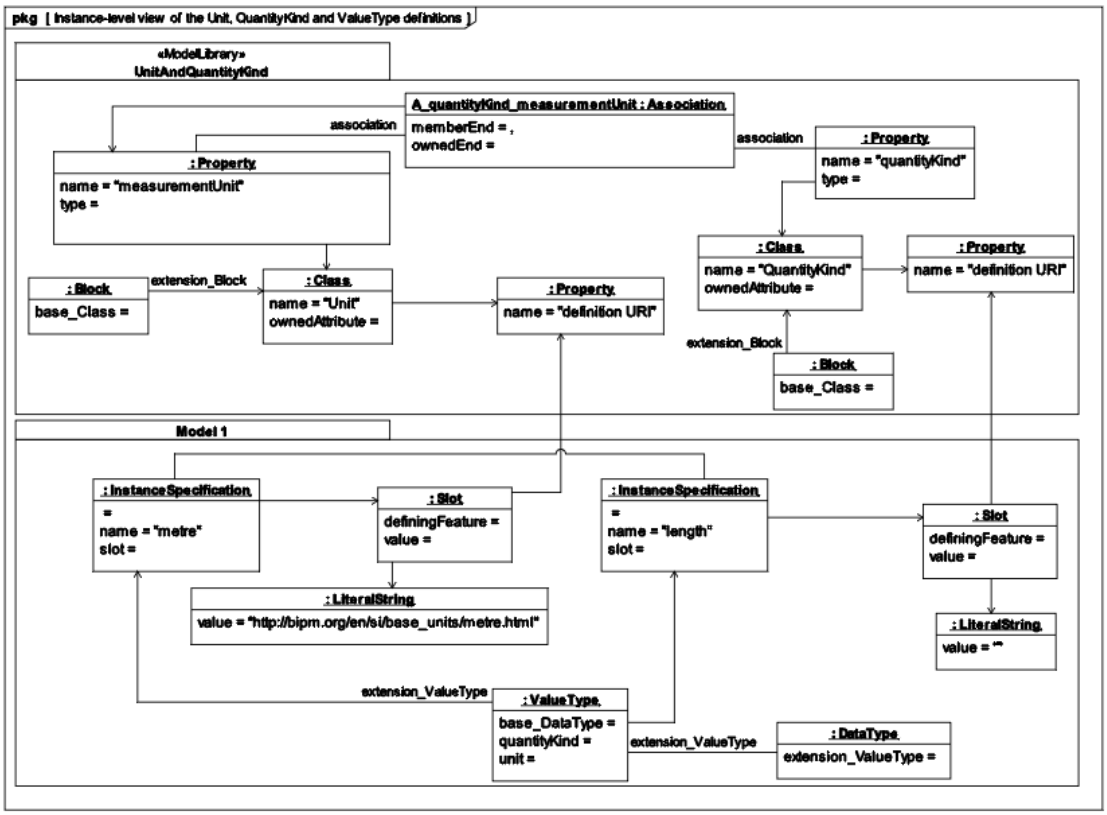


Figure 8-19: Instance-level view of the Unit, QuantityKind and Value Type definitions

The following example shows a minimal example of the semantics of Unit equivalence (A similar example for QuantityKind is omitted).

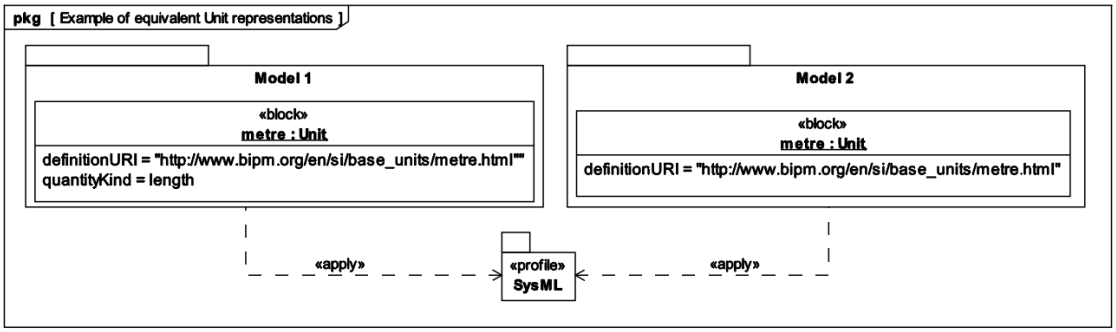


Figure 8-20: Example of equivalent Unit representations

In terms of the UML4SysML metamodel and of the SysML profile, the following figure shows a partial account of the instance-level representation of the above example. This instance-level representation is important for model interchange, particularly across different implementations of SysML.

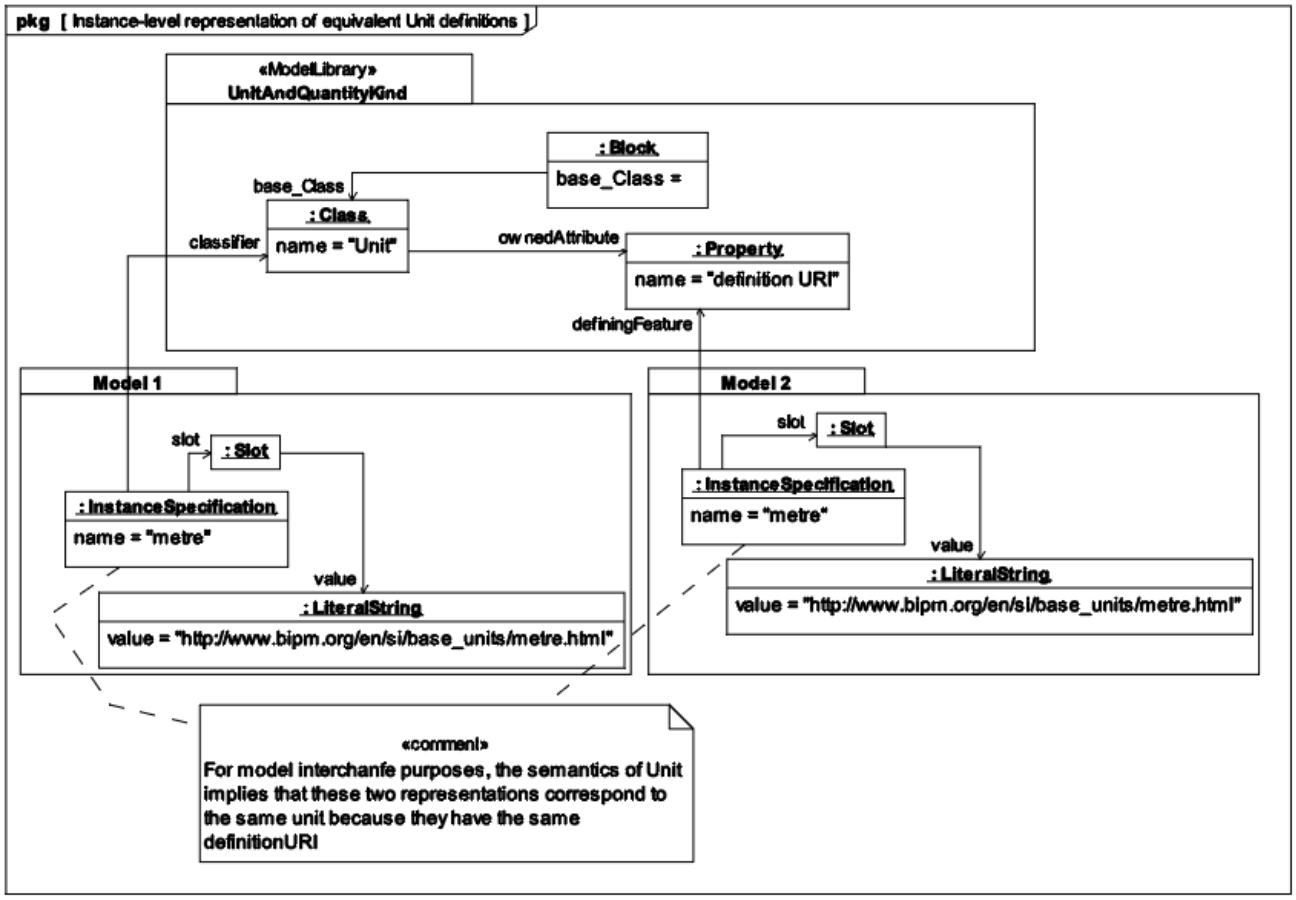


Figure 8-21: Instance-level representation of equivalent Unit definitions

8.4.7 Property-Specific Types

Figure 8-22 shows property-specific types in a model of facilities that includes factories and warehouses. Items flow through facilities, while resources operate on items. Items in warehouses are assigned a location, while resources in factories indicate own much they are being used as a percentage of time. Only objects that are items in warehouses or resources in factories have these location and utilization properties. The properties appear when an item arrives in a warehouse or a resource is used in a factory, because they are classified as WarehouseItems and FactoryResources at that time, respectively. The properties disappear once an item leaves a warehouse or a resource is no longer used in a factory, because they are declassified as WarehouseItems and FactoryResources at that time, respectively.

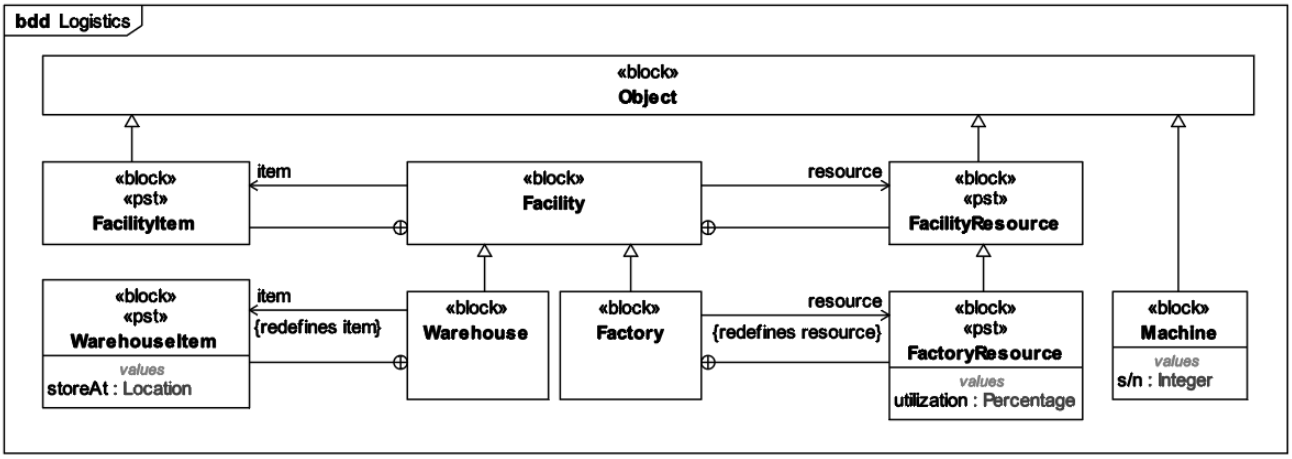


Figure 8-22: Property-specific types in facility example

Figure 8-23 shows the classification of a particular machine over time, identified by its serial number. At first it is not an item or resource and is classified only as a machine. Before delivery to the factory, a new machine is stored in a warehouse, classified additionally as a warehouse item, and is assigned a storage location. Then it is delivered to a factory, reclassified from a warehouse item to a factory resource (while still being a machine), and records the percentage of time it is operating.

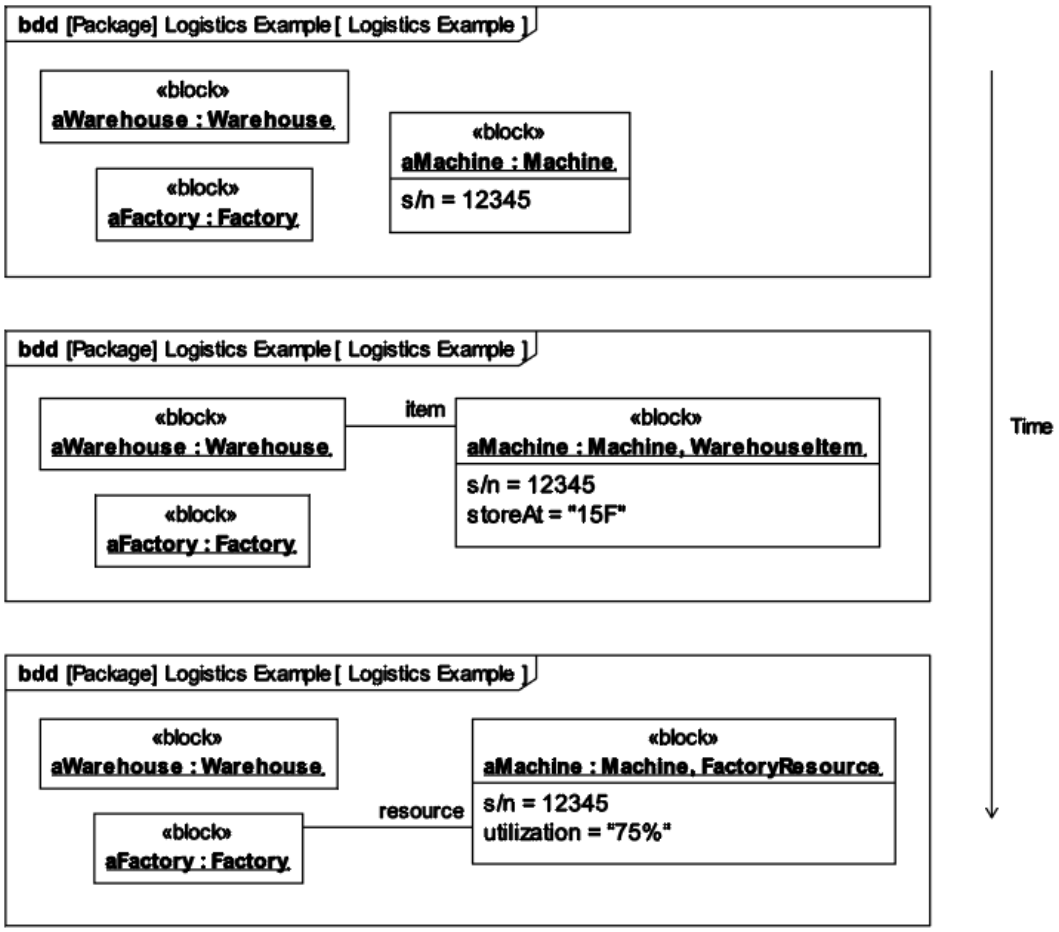


Figure 8-23: Changes in classification over time due to property-specific types

9 Ports and Flows

9.1 Overview

The main motivation for specifying ports and flows is to enable design of modular, reusable blocks with clearly defined ways of connecting and interacting with their context of use. This clause extends UML ports to support nested ports, and extends blocks to support flow properties, and required and provided features, including blocks that type ports. Ports can be typed by blocks that support operations, receptions, and properties as in UML. SysML defines a specialized form of Block (InterfaceBlock) that can be used to support nested ports. SysML identifies two kinds of ports, one that exposes features of the owning block or its internal parts (proxy ports), and another that supports its own features (full ports). Default compatibility rules are defined for connecting block usages, such as parts and ports. These can be overridden with association blocks specifying connections. These additional capabilities in SysML enable modelers to specify a wide variety of interconnectable components, which can be implemented through many engineering and social techniques, such as software, electrical or mechanical components, and human organizations. This clause also extends UML information flows for specifying item flows across connectors and associations.

9.1.1 Ports

Ports are points at which external entities can connect to and interact with a block in different or more limited ways than connecting directly to the block itself. They are properties with a type that specifies features available to the external entities via connectors to the ports. The features might be properties, including flow properties and association ends, as well as operations and receptions. The remaining overview sub clauses introduce other aspects of ports and flows.

9.1.2 Flow Properties, Provided and Required Features, and Nested Ports

SysML extends blocks to support flow properties and provided and required features. Blocks with ports can type other ports (nested ports). Flow properties specify the kinds of items that might flow between a block and its environment, whether it is data, material, or energy. The kind of items that flow is specified by typing flow properties. For example, a block specifying a car's automatic transmission could have a flow property for Torque as an input, and another flow property for Torque as an output. Required and provided features are operations, receptions, and non-flow properties that a block supports for other blocks to use, or requires other blocks to support for its own use, or both. For example, a block might provide particular services to other blocks as operations, or have a particular geometry accessible to other block, or it might require services and geometries of other blocks. Ports nest other ports in the same way that blocks nest other blocks. The type of the port is a block (or one of its specializations) that also has ports. For example, the ports supporting torque flows in the transmission example might have nested ports for physical links to the engine or the driveshaft.

9.1.3 Proxy Ports and Full Ports

SysML identifies two usage patterns for ports, one where ports act as proxies for their owning blocks or its internal parts (proxy ports), and another where ports specify separate elements of the system (full ports). Both are ways of defining the boundary of the owning block as features available through external connectors to ports. Proxy ports define the boundary by specifying which features of the owning block or internal parts are visible through external connectors, while full ports define the boundary with their own features. Proxy ports are always typed by interface blocks, a specialized kind of block that has no behaviors or internal parts. Full ports cannot be behavioral in the UML sense of standing in for the

owning object, because they handle features themselves, rather than exposing features of their owners, or internal parts of their owners. Ports that are not specified as proxy or full are simply called “ports.”

In either case, users of a block are only concerned with the features of its ports, regardless of whether the features are surfaced by proxy ports, or handled by full ports directly. Proxy and full ports support the capabilities of ports in general, but these capabilities are also available on ports that are not declared as proxy or full. Modelers can choose between proxy or full ports at any time in the development lifecycle, or not at all, depending on their methodology.

9.1.4 Item Flows

Item flows specify the things that flow between blocks and/or parts and across associations or connectors. Whereas flow properties specify what “can” flow in or out of a block, item flows specify what “does” flow between blocks and/or parts in a particular usage context. This important distinction enables blocks to be interconnected in different ways depending on its usage context. For example, tanks might include a flow property that can accept fluid as an input. In a particular use of tanks, “gasoline” flows across a connector into a tank, and in another use of tanks, “water” flows across a connector into a tank. The item flow in each case specifies what “does” flow on the connector in the particular usage (e.g., gas, water) and the flow property specifies what can flow (e.g., fluid). This enables type matching between the item flows and between flow properties to assist in interface compatibility analysis.

Item flows may be allocated from object nodes in activity diagrams or signals sent from state machines across a connector. Flow allocation is described in Clause 15, “Allocations,” and can be used to help ensure consistency across the different parts of the model.

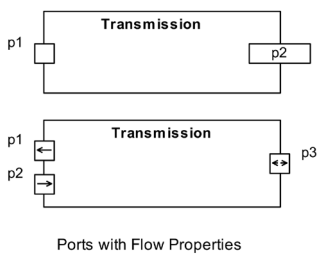
9.1.5 Deprecation of Flow Ports and Flow Specifications

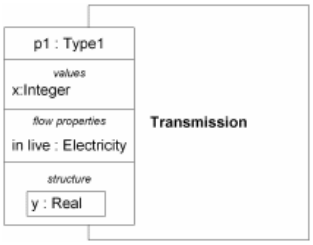
Flow ports and flow specifications are included in SysML, but are deprecated. Annex C defines them, along with transition guidelines to non-deprecated elements. In particular, the functionality of non-atomic flow ports is supported with proxy ports typed by interface blocks owning flow properties. Flow properties are not deprecated.

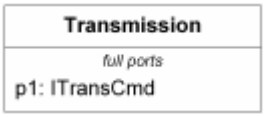
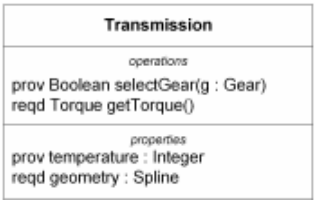
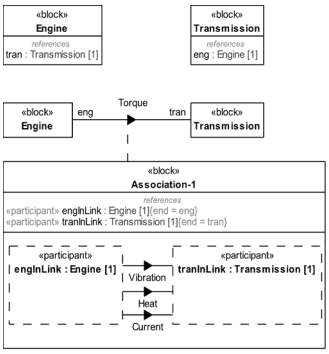
9.2 Diagram Elements

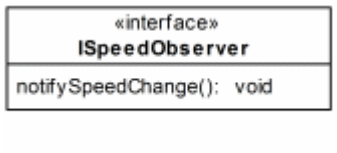
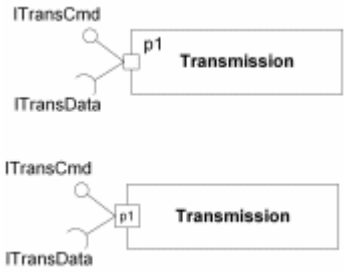
9.2.1 Block Definition Diagram

Table 9-1: Graphical nodes defined in Block Definition diagram

Node Name	Concrete Syntax	Abstract Syntax Reference
Port	 Ports with Flow Properties	**UML4SysML::Port**

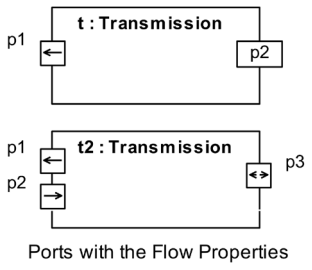
Node Name	Concrete Syntax	Abstract Syntax Reference
Port (Compartment Notation)		**UML4SysML::Port**
Port (with Compartment)		**UML4SysML::Port**
Port (Nested)		**UML4SysML::Port**
ProxyPort		**SysML::PortsAndFlows::ProxyPort**
ProxyPort (Compartment Notation)		**SysML::PortsAndFlows::ProxyPort**
FullPort		**SysML::PortsAndFlows::FullPort**

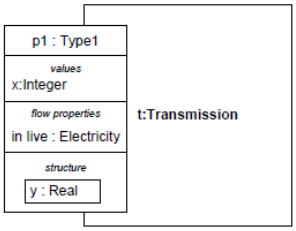
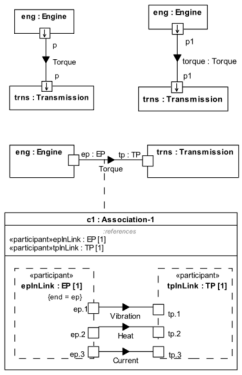
Node Name	Concrete Syntax	Abstract Syntax Reference
FullPort (Compartment Notation)		**SysML::PortsAndFlows::FullPort**
FlowProperty		**SysML::PortsAndFlows::FlowProperty**
Required and Provided Features		**SysML::PortsAndFlows::Directed Feature**
InterfaceBlock		**SysML::PortsAndFlows::InterfaceBlock**
ItemFlow		**SysML::PortsAndFlows::ItemFlow**

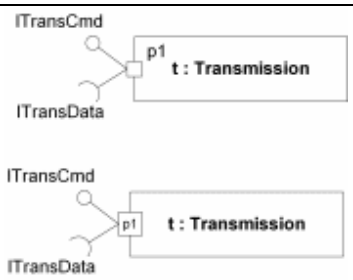
Node Name	Concrete Syntax	Abstract Syntax Reference
Interface		**UML4SysML::Interfaces::Interface**
Required and Provided Interfaces		**UML4SysML::Interface**

9.2.2 Internal Block Diagram

Table 9-2: Graphical nodes defined in Internal Block diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Port		**UML4SysML::Port**

Node Name	Concrete Syntax	Abstract Syntax Reference
Port (Nested)		**UML4SysML::Port**
Port (with Compartment)		**UML4SysML::Port**
ProxyPort		**SysML::PortsAndFlows::ProxyPort**
ProxyPort (isBehavior = true)		**SysML::PortsAndFlows::ProxyPort**
FullPort		**SysML::PortsAndFlows::ProxyPort**
ItemFlow		**SysML::PortsAndFlows::ItemFlow**

Node Name	Concrete Syntax	Abstract Syntax Reference
Required and Provided Interfaces		**UML4SysML::Interface**

9.3 UML Extensions

9.3.1 Diagram Extensions

9.3.1.1 DirectedFeature

A DirectedFeature has the same notation as other non-flow properties and behavioral features with a feature direction prefix (prov | reqd | provreqd), which corresponds to one the FeatureDirectionKind literals “provided,” “required,” and “providedrequired,” respectively. Directed features can appear in compartments for the various kinds of properties and behavioral features.

9.3.1.2 FlowProperty

A FlowProperty signifies a single flow element to/from a block. A flow property has the same notation as a Property only with a direction prefix (in | out | inout). Flow properties are listed in a compartment labeled flow properties.

9.3.1.3 FullPort

Full ports can appear in block compartments labeled full ports. The keyword «full» before a property name can also indicate the property is stereotyped by FullPort.

9.3.1.4 InvocationOnNestedPortAction

The nested port path is notated with a string “‘via’ <port-name> [‘,’ <port-name>]+” in the name string of the icon for the invocation action. It shows the values of the onNestedPort property in order, and the value of the onPort property at the end.

9.3.1.5 ItemFlow

An ItemFlow describes the flow of items across a connector or an association. The notation of an item flow is a black arrowhead on the connector or association. The arrowhead is towards the target element. For an item flow with an item property, the label shows the name and type of the item property (in name: type format). Otherwise the item flow is labeled with the name of the classifier of the conveyed items. When several item flows having the same direction are represented, only one triangle is shown, and the list of item flows, separated by a comma is presented.

9.3.1.6 Port

Ports are notated by rectangles overlapping the boundary of their owning blocks or properties (parts or ports) typed by the owning block. Port labels appear in the same format as properties on the end of an association. Port labels can appear inside port rectangles. Nested ports that are not on proxy ports can appear anywhere on the boundary of the owning port rectangle that does not overlap the boundary of the rectangle the owning port overlaps.

Port rectangles can have port rectangles overlapping their boundaries, to notate a port type that has ports (nested ports).

Ports with types that have flow properties all in the same direction, either all in or all out, can have an arrow inside them indicating the direction of the properties with respect to the owning block. (See FlowProperty on page 90 for definition of owning block of proxy ports in this case.) This includes the direction of flow properties on nested ports, and if the port is full and its type is unencapsulated, ports on parts of the port, recursively. The arrows are perpendicular to the boundary lines they overlap. Ports with types that have flow properties in different directions or flow properties that are all in both directions, including have two open arrow heads inside them facing away from each other (<>). This includes the directions of nested and contained flow properties as described above for one-way arrows. Ports appearing in block compartments can have their direction appear textually before the port name as “in,” “out,” or “inout” determined in the same way as the arrow direction.

Ports that are not proxy or full can appear in block compartments labeled ports.

Ports are specialized kinds of properties, and can be shown in same way as other properties. They can appear in block compartments in the same format as other properties of their owning blocks, or as the ends of associations, with the port appearing in the same format as other association ends, on the end opposite the owning block.

All ports and nested ports (i.e., proxy, full, and ports with no stereotype applied), and their type definitions (e.g., interface blocks, blocks) can include compartments with textual and graphical representations to display their features in the same way as other properties and types, including rectangles used to display properties in structure compartments.

9.3.1.7 ProxyPort

Proxy ports can appear in block compartments labeled proxy ports. The keyword «proxy» before a property name can also indicate the property is stereotyped by ProxyPort. Nested ports on proxy ports can appear on the portion of the boundary of the owning port rectangle that is outside the rectangle the owning port overlaps.

9.3.1.8 TriggerOnNestedPort

The nested port path is notated following a trigger signature with a string “‘«from» (’ <port-name> [‘,’ <port-name>]+ ‘)’” in the name string of the icon for the trigger. It shows the values of the onNestedPort property in order, and the value of the port property at the end.

9.3.2 Stereotypes

Package PortsAndFlows

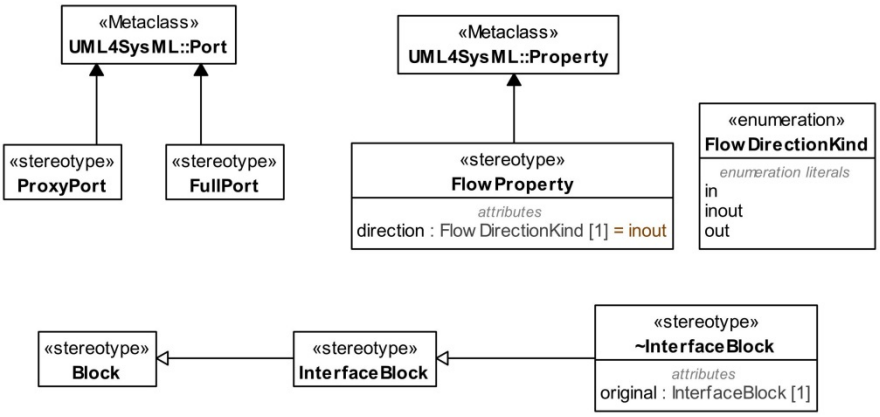


Figure 9-1: Port Stereotypes

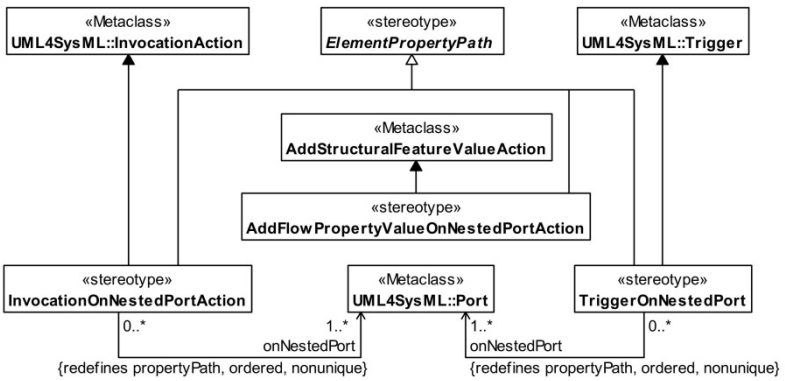


Figure 9-2: Stereotypes for Actions on Nested Ports

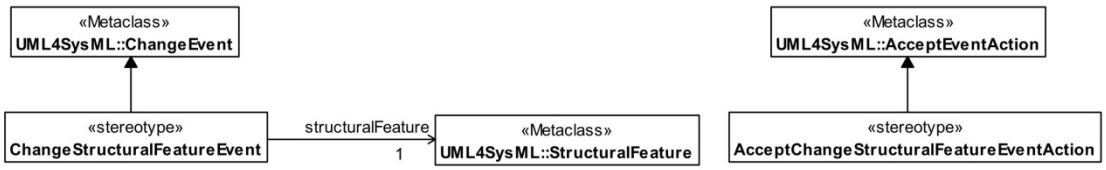


Figure 9-3: Stereotypes for Property Value Change Events

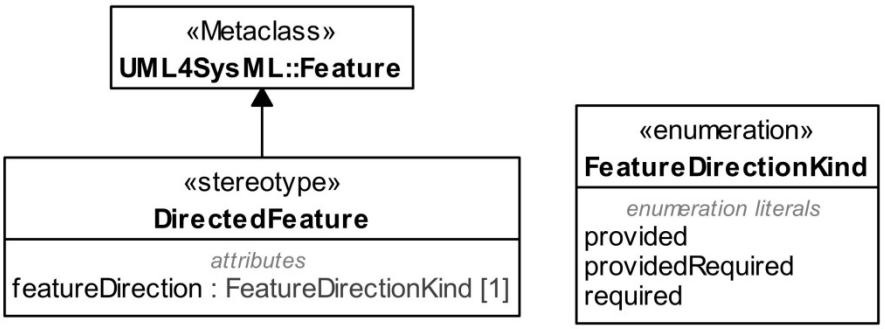


Figure 9-4: Provided and Required Features

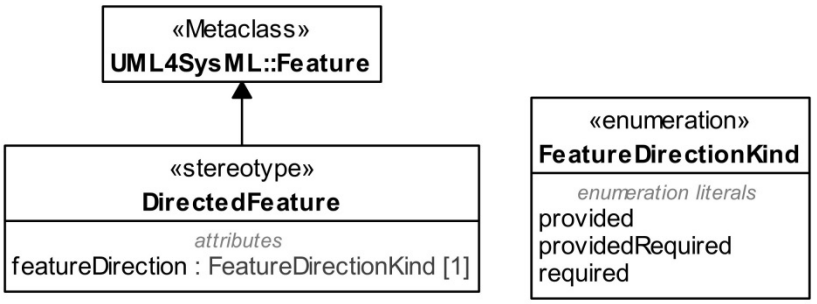


Figure 9-5: Item Flow Stereotype

9.3.2.1 AcceptChangeStructuralFeatureEventAction

Description

Accept change structural feature event actions handle change structural feature events (see clause 9.3.2.10. The actions have exactly two output pins. The first output pin holds the values of the structural feature just after the values changed, while the second pin holds the values just before the values changed. The action only accepts events for structural features on the blocks owning the behavior containing the action, or on the behavior itself, if the behavior is not owned by a block.

Association Ends

- base_AcceptEventAction : AcceptEventAction [1]

Constraints

- 1_one_trigger
  The action has exactly one trigger, the event of which shall be a change structural feature event.

```
self.base_AcceptEventAction.trigger->size()=1 and let trigger: UML::Trigger =
self.base_AcceptEventAction.trigger->any(true) in
ChangeStructuralFeatureEvent.allInstances().base_ChangeEvent->
includes(trigger.event)
```

- 2_two_resultpins
  The action has two result pins with type and ordering the same as the type and ordering of the structural feature of the trigger event, and multiplicity compatible with the multiplicity of the structural feature.

```
let event: ChangeStructuralFeatureEvent =
ChangeStructuralFeatureEvent.allInstances()->any(e | e.base_ChangeEvent =
self.base_AcceptEventAction.trigger->any(true).event) in
self.base_AcceptEventAction.result->size() = 2 and
self.base_AcceptEventAction.result->forAll(r | r.type =
event.structuralFeature.type and r.isOrdered = event.structuralFeature.isOrdered
and r.lower <= event.structuralFeature.lower and r.upper >=
event.structuralFeature.upper)
```

- 3_context_owns_structuralfeature
  The structural feature of the trigger event shall be owned by or inherited by the context of the behavior containing the action. (The context of a behavior is either its owning block or itself if it is not owned by a block. See definition in the UML 2 standard.)

```
let event: ChangeStructuralFeatureEvent =
ChangeStructuralFeatureEvent.allInstances()->any(e | e.base_ChangeEvent =
self.base_AcceptEventAction.trigger->any(true).event) in
self.base_AcceptEventAction._'context'->notEmpty() and
self.base_AcceptEventAction._'context'.allFeatures()
->includes(event.structuralFeature)
```

- 4_can_access_structuralfeature
  Visibility of the structural feature of the trigger event shall allow access to the object performing the action.

```
let event: ChangeStructuralFeatureEvent =
ChangeStructuralFeatureEvent.allInstances()->any(e | e.base_ChangeEvent =
```

```
self.base_AcceptEventAction.trigger->any(true).event) in if
event.structuralFeature.visibility = UML::VisibilityKind::private then
self.base_AcceptEventAction._'context'.feature->includes(event.structuralFeature)
else if event.structuralFeature.visibility = UML::VisibilityKind::protected then
self.base_AcceptEventAction._'context'.allFeatures()
->includes(event.structuralFeature) else if event.structuralFeature.visibility =
UML::VisibilityKind::_'package' then let thePackage: UML::Package =
event.structuralFeature.allNamespaces()->select(n | n.oclIsKindOf(UML::Package))
->first().oclAsType(UML::Package) in (not thePackage.oclIsUndefined()) and ( let
index: Integer = event.structuralFeature.allNamespaces()->indexOf(thePackage) in
event.structuralFeature.allNamespaces()->subOrderedSet(1, index) ->iterate(n; acc:
Boolean=true | acc and not (n.visibility=UML::VisibilityKind::private or
n.visibility=UML::VisibilityKind::protected)) ) else true endif endif endif
```

- `5_uml_constraint_removed`
  The constraint under 11.3.2, "AcceptEventAction" in the UML 2 standard, "[2] There are no output pins if the trigger events are only ChangeEvents," shall be removed for accept event actions that have AcceptChangeStructuralFeatureEventAction applied.

```
-- cannot be expressed in OCL
```

9.3.2.2 AddFlowPropertyValueOnNestedPortAction

Description

This enables values added to a flow property to propagate out through a specified behavioral port of an object executing the action, rather than all behavior ports exposing the flow property. It also enables values added to a flow property to propagate into objects. Values flowing out of an object are added to an out or inout flow property of the executing object. In this case, the applied stereotype specifies a (possibly nested) behavioral port at the end of a (possibly multi-level) path of behavioral ports from a block that supports the flow property. Values flowing into an object are added to an in or inout flow property of that object, specifying a (possibly nested) port of that object.

Generalizations

- ElementPropertyPath (from Blocks)

Attributes

- onNestedPort : Port [1..*]
  Gives a series of ports that end in one supporting the flow property to which a value is being added. The ordering of ports is from a port of the object of the stereotyped action, through a port of each intermediate block that types the preceding port, ending in a port with a type that owns or inherits the flow property. The same port might appear more than once because a block can own a port with the same block as a type, or another block that has the same property.
  (redefines: ElementPropertyPath::propertyPath)

Association Ends

- base_AddStructuralFeatureValueAction : AddStructuralFeatureValueAction [1]

Constraints

- 1_feature_flowproperty
  The structural feature referred by actions with this stereotype applied must have FlowProperty applied.

```
FlowProperty.allInstances().base_Property
->includes(self.base_AddStructuralFeatureValueAction.structuralFeature)
```

- 2_onnestedport_first_owned_by_target_type
  The port at the first position in the onNestedPort list shall be owned by the block that types the object pin of the stereotyped action, or one of that blocks generalizations.

```
self.base_AddStructuralFeatureValueAction.object.type.oclAsType(UML::Classifier) -
>allFeatures()->includes(self.onNestedPort->first()))
```

- 3_path_consistency
  The port at each successive position of the onNestedPort attribute, following the first position, shall be owned by the Block that types the port at the immediately preceding position, or a generalization of that Block

```
self.onNestedPort->size() >1 implies self.propertyPath->subSequence(2,
self.onNestedPort->size())->forAll(p |
let pp: UML::Property = self.onNestedPort->at(self.onNestedPort->indexOf(p)-1) in
let owners: Set(UML::Classifier) = pp.type.oclAsType(UML::Classifier)
->including(pp.type.oclAsType(UML::Classifier)) in
owners->includes(p.owner))
```

- 4_onnestedport_last_type_owns_invocation_onPort
  The type of the port at the last position of the onNestedPort list shall own or inherit the flow property that is the structural feature of the stereotyped action

```
self.onNestedPort->last().type.oclAsType(UML::Classifier).allFeatures()
->includes(self.base_AddStructuralFeatureValueAction.structuralFeature)
```

9.3.2.3 Block

Description

Blocks (including specializations of Block) can own ports, including but not limited to proxy ports and full ports. These blocks can be the type of ports (specifying nested ports), with some restrictions described in other stereotypes in this sub clause. All links and interactions with a behavioral port (in the UML sense of standing in for the owning object) are links and interactions with the owner, so the semantics of behavioral ports is the same as if the value of the port as a property were always the owning block instance (the owning block instance for behavioral ports on proxy ports is the value of the block usage the proxy port is standing in for, which might be an internal part). Blocks loosen UML constraints on connectors to support nested ports. See Clause 8, “Blocks” for further details of blocks.

9.3.2.4 ChangeStructuralFeatureEvent

Description

A ChangeStructuralFeatureEvent models changes in values of structural features.

Association Ends

- base_ChangeEvent : ChangeEvent [1]
- structuralFeature : StructuralFeature [1]
  he event models occurrences of changes to values of this structural feature.

Constraints

- 1_not_static
  The structural feature shall not be static
```
not self.structuralFeature.isStatic
```

- 2_one_featuringclassifier
  The structural feature shall have exactly one featuringClassifier
```
self.structuralFeature.featuringClassifier->size()=1
```

9.3.2.5 DirectedFeature

Description

A DirectedFeature indicates whether the feature is supported by the owning block (provided), or is to be supported by other blocks for the owning block to use (required), or both (the owning block for features on types of proxy ports is the type of the block usage the proxy port is standing in for, which might be an internal part). Using non-flow properties means to read or write them, and using behavioral features means to invoke them. Provided non-flow properties are read and written on the owning block, while required non-flow properties are read or written on an external block. Provided behavioral features are invoked with the owning block as target, while required behavioral features are invoked with an external block as target (required).

Blocks owning or inheriting required behavioral features can have behaviors invoking the behavioral features on instances of the block. This sends invocations out along connectors from usages of the block in internal structures of other blocks, provided the behavioral features match on the other end of the connectors.

Invocations of provided behavioral features due to required behavioral features can only occur when the features match. A single provided behavioral feature shall match each required one according to the following conditions:

- The kind of behavioral feature is the same (operation or reception).
- Names are the same, including parameter names, in the same order.
- Parameter directions are the same, in the same order.
- Provided parameter types for parameters with:
  - in direction are the same or more general than the required ones, in order.
  - out or return direction are the same or more specialized than the required ones, in order.
  - inout direction are the same as the required ones, in order.

Parameters without types are treated as if their type is more general than all other types.

- Provided parameter multiplicity has the same condition as type, where wider multiplicities are “more general” than narrower ones.
- Provided parameter order (of each parameter separately) has the same condition as type, where unordered parameters are “more general” than ordered ones.
- Provided parameter uniqueness (of each parameter separately) has the same condition as type, where non-unique parameters are “more general” than unique ones.
- Provided operation preconditions are the same as or more general than required ones.
- Provided operation body conditions and postconditions are the same or more specialized than required ones.

If corresponding parameters in provided and required behavioral features both have defaults, the default value specification of the required feature is used for in parameters, and the default value specification of the provided feature is used for out and return parameters.

Reading or writing provided non-flow properties due to required non-flow properties can only occur when the features match. Matching non-flow properties shall have the same name. For reading non-flow properties, the types, multiplicities, uniqueness, and ordering shall match in the same way as out parameters for behavioral features above. For writing non- flow properties, the types, multiplicities, uniqueness, and ordering shall match in the same way as in parameters for behavioral features above. For both reading and writing non-flow properties, the types, multiplicities, uniqueness, and ordering shall be the same. If provided and required non-flow properties both have defaults, the default value specification of the required feature is used for writing and the default specification of the provided feature is used for reading.

Attributes

featureDirection : FeatureDirectionKind [1]
Specifies whether the feature is supported by the owning block (featureDirection="provided"), or is to be supported by other blocks for the owning block to use (featureDirection="required"), or both (featureDirection="providedrequired").

Association Ends

base_Feature : Feature [1]

Constraints

1_behavioralfeature_or_not_flowproperty
DirectedFeature shall only be applied to behavioral features, or to properties that do not have FlowProperty applied, including on subsetted or redefined features.

```
self.base_Feature.oclIsKindOf(UML::BehavioralFeature) or
(self.base_Feature.oclIsKindOf(UML::Property) and let property: UML::Property =
self.base_Feature.oclAsType(UML::Property) in
FlowProperty.allInstances().base_Property->excludesAll(property.redefinedProperty
->union(property.subsettedProperty)->including(property)))
```

2_method_if_provided
A non-provided operation shall not be associated with a behavior as its method.

```
self.base_Feature.oclIsKindOf(UML::Operation) and
self.featureDirection=FeatureDirectionKind::required implies
self.base_Feature.oclAsType(UML::Operation).method->isEmpty()
```

9.3.2.6 FeatureDirectionKind

Description

FeatureDirectionKind is an enumeration type that defines literals used by directed features for specifying whether they are supported by the owning block, or is to be supported by other blocks for the owning block to use.

Literals

- provided
  Indicates that the feature shall be supported by the owning block.
- providedRequired
  Indicates that the feature shall be both provided and required.
- required
  Indicates that the feature shall be supported by other blocks.

Constraints

- `2_specializations_are_constraintblocks`
  Any classifier that specializes a ConstraintBlock shall also have the ConstraintBlock stereotype applied.

```
UML::Classifier.allInstances()->forAll(c | c.general->includes(self.base_Class)
implies ConstraintBlock.allInstances().base_Class->includes(c))
```

9.3.2.7 FlowDirectionKind

Description

FlowDirectionKind is an enumeration type that defines literals used for specifying the direction that items can flow to or from a block. FlowDirectionKind is used by flow properties to indicate the direction that its items can flow to or from its owner. (See 9.3.2.13 for definition of owning block of proxy ports in this case.)

Literals

- in
  Indicates that items of the flow property can flow into the owning block.
- inout
  Indicates that items of the flow property can flow into or out of the owning block.
- out
  Indicates that items of the flow property can flow out of the owning block.

9.3.2.8 FlowProperty

Description

A FlowProperty signifies a single kind of flow element that can flow to/from its owning instance that is specified by the block defining that flow property. A flow propertys values are either received from or transmitted to another instance. An "in" flow property value cannot be modified by the owning instance of that flow property, or by parts of that instance. An "out" flow property can only be modified by the owning instance of that flow property, or by parts of that instance. An "inout" flow property can be used as an "in" flow property or an "out" flow property, and there is no restriction regarding the way it can be modified. (The owning block of a proxy port in this case depends on how the port is nested in the internal structures of blocks, because the block directly owning the port might be used to type ports or parts at different levels of nesting in multiple blocks, or the same block. The owning block of a proxy port in the internal structure of a block is the block typing the innermost full port or part under which the port is nested.)

Flow due to flow properties can only occur when flow properties match. Matching flow properties shall have matching direction and types. Matching direction is defined below. Flow property types match when the target flow property type has the same, or a generalization of, the source flow property type. (See 9.3.2.11, ItemFlow for looser constraints on flow property types across connectors with item flows.) If multiple flow properties on either end of a connector match by direction and type, then the names of the flow properties shall also be the same for flow to occur. If multiple flow properties on either end match by direction, type, and name, which can happen for unnamed flow properties, then no flow will occur.

Flow properties enable item flows across connectors between usages typed by blocks having the properties. For Block and ValueType flow properties, setting an "out" or "inout" FlowProperty value of a block usage on one end of a connector will result in assigning the same value of an "in" or "inout" FlowProperty of a block usage at the other end of the connector, provided the flow properties are matched. It is not specified whether send/receive signal events are generated when values are written to out/in flow properties typed by Signal (implementations might choose to do this, but it is not required). This paragraph does not apply to internal connectors of proxy ports, see next paragraph.

Items going to or from behavioral ports (UML isBehavior = true) are actually going to or from the owning block. (See 9.3.2.8 for definition of owning block of proxy ports in this case.) Items going to or from non-behavioral ports (UML isBehavior = false) are actually going to the port itself (for full ports) or to internal parts connected to the port (for proxy ports). Because of this, flow properties of a proxy port are the same as flow properties on the owning block or internal parts, so the flow property directions shall be the same on the proxy port and owning block or internal parts for items to flow. See section 9.3.2.13 for the definition of internal connectors and the semantics of proxy ports.

The flow property semantics above applies to each connector of a block usage, including when the block usage has multiple connectors.

The binding of flow properties on ports to behavior parameters can be achieved in ways not dictated by SysML. One approach is to perform name and type matching. Another approach is to explicitly use binding relationships between the ports properties and behavior parameters or block properties.

Attributes

- direction : FlowDirectionKind [1]
  Specifies if the property value is received from an external block (direction="in"), transmitted to an external Block (direction="out") or both (direction="inout").

Association Ends

- base_Property : Property [1]

Constraints

- 1_restricted_types
  A FlowProperty shall be typed by a ValueType, Block, or Signal.
  ```
  Block.allInstances().base_Class->includes(self.base_Property.type) or
  ValueType.allInstances().base_DataType->includes(self.base_Property.type) or
  self.base_Property.oclIsKindOf(UML::Signal)
  ```

9.3.2.9 FullPort

Description

Full ports specify a separate element of the system from the owning block or its internal parts. They might have their own internal parts and behaviors to support interaction with the owning block, its internal parts, or external blocks. They cannot be behavioral ports, or linked to internal parts by binding connectors, because these constructs imply identity with the owning block or internal parts. However, full ports can be linked to non-full ports by binding connectors, because this does not necessarily imply identity with other parts of the system.

Association Ends

- base_Port : Port [1]

Constraints

- 1_not_proxy
  Full ports shall not also be proxy ports. This applies even if some of the stereotypes are on subsetted or redefined ports.
  ```
  ProxyPort.allInstances()->excludes(self.base_Port)
  ```

- 2_not_bound_to_fullport
  Binding connectors shall not link full ports (either directly or indirectly through other binding connectors) to other composite properties of the block owning the full port (or that blocks generalizations or specializations), unless the composite properties are non-full ports.
  ```
  let fullPorts: Set(UML::Port) = FullPort.allInstances().base_Port->asSet() in
  BindingConnector.allInstances().base_Connector->select(c | c.end.role
  ->includes(self.base_Port))->forAll(c | fullPorts->excludesAll(c.end.role->reject
  (r | r=self.base_Port)))
  ```

- 3_not_behavioral
  Full ports shall not be behavioral (isBehavior=false).
  ```
  not self.base_Port.isBehavior
  ```

9.3.2.10 InterfaceBlock

Description

Interface blocks cannot have behaviors, including classifier behaviors or methods, or internal parts.

Generalizations

- Block (from Blocks)

Operations

- getConjugated () : InterfaceBlock [0..*]

```
bodyCondition:
~InterfaceBlock.allInstances()->any(ib | ib.original = self)
```

Constraints

- 1_no_behavior
  Interface blocks shall not own or inherit behaviors, have classifier behaviors, or methods for their behavioral features.

```
self.base_Class.inheritedMember->select(m | m.oclIsKindOf(UML::Behavior))
->isEmpty() and self.base_Class.operation.method->flatten()->isEmpty()
```

- 2_no_part
  Interface blocks composite properties are either ports, value properties or flow properties.

```
self.base_Class.ownedAttribute->select(a|a.isComposite)->forAll(a |
a.oclIsKindOf(UML::Port) or a.oclIsKindOf(ValueType))
```

- 3_interfaceblock_typed_ports
  Ports owned by interface blocks shall only be typed by interface blocks.

```
self.base_Class.ownedPort->forAll(p|InterfaceBlock.allInstances().base_Class
->includes(p.type))
```

- isconjugated_not_used
  Any port typed by an InterfaceBlock shall have its isConjugated property set to false.

```
Port.allInstances()->forAll(p | p.type = self.base_Class implies
p.isConjugated=false)
```

9.3.2.11 InvocationOnNestedPortAction

Description

This extends the capabilities of UMLs onPort property of InvocationAction to support nested ports. It identifies a nested port by a multi-level path of ports from the block that executes the action. Like UMLs onPort property, this extends invocation actions to send invocations out of ports of objects executing the actions, or to ports of those objects or other objects. Invocations intended to go out of the object executing the action shall be sent to the executing object on a proxy port. Invocations intended to go directly to a target object are sent to that object on a port of that object.

Generalizations

- ElementPropertyPath (from Blocks)

Association Ends

- base_InvocationAction : InvocationAction [1]
  (redefines: ElementPropertyPath::base_Element)
- onNestedPort : Port [1..*]
  Gives a series of ports that identifies the port receiving the invocation in the context of the target object of the invocation. The ordering of ports is from a port of the target object, through a port of each intermediate block that types the preceding port, ending in a port with a type that owns or inherits the port given by the onPort property of the invocation action. The onPort port is not included in the onNestedPort list. The same port might appear more than once because a block can own a port with the same block as a type, or another block that has the same property.
  (redefines: ElementPropertyPath::propertyPath)

Constraints

- 1_onPort_defined
  The onPort property of an invocation action shall have a value when this stereotype is applied.

```
self.base_InvocationAction.onPort->notEmpty()
```

- 2_onnestedport_first_owned_by_target_type
  The port at the first position in the onNestedPort list shall be owned (directly or via inheritance) by a block that types the target pin of the invocation action, or one of the blocks generalizations.

```
let target: UML::InputPin = if
self.base_InvocationAction.oclIsKindOf(UML::CallOperationAction) then
self.base_InvocationAction.oclAsType(UML::CallOperationAction).target

else if self.base_InvocationAction.oclIsKindOf(UML::SendSignalAction) then
self.base_InvocationAction.oclAsType(UML::SendSignalAction).target
else if self.base_InvocationAction.oclIsKindOf(UML::SendObjectAction) then
self.base_InvocationAction.oclAsType(UML::SendObjectAction).target
else
invalid
endif endif endif in
not target.oclIsUndefined() and (
let target_type: UML::Class = Block.allInstances()->any(b | b.base_Class =
target.type).base_Class in
not target_type.oclIsUndefined() and target_type.allFeatures()
->includes(self.onNestedPort->first()))
```

- 3_path_consistency
  The port at each successive position of the onNestedPort attribute, following the first position, shall be owned by the Block that types the port at the immediately preceding position, or a generalization of that Block.

```
self.onNestedPort->size() >1 implies self.propertyPath->subSequence(2,
self.onNestedPort->size())->forAll(p |
let pp: UML::Property = self.onNestedPort->at(self.onNestedPort->indexOf(p)-1) in
let owners: Set(UML::Classifier) = pp.type.oclAsType(UML::Classifier)
```

```
->including(pp.type.oclAsType(UML::Classifier)) in
owners->includes(p.owner))
```

- 4_onnestedport_last_type_owns_invocation_onPort
  The type of the port at the last position of the onNestedPort list shall own or inherit the onPort port of the stereotyped invocation action.

```
self.onNestedPort->last().type.oclAsType(UML::Classifier).allFeatures()
->includes(self.base_InvocationAction.onPort)
```

9.3.2.12 ItemFlow

Description

An ItemFlow describes the flow of items across a connector or an association. It may constrain the item exchange between blocks, block usages, or ports as specified by their flow properties. For example, a pump connected to a tank: the pump has an "out" flow property of type Liquid and the tank has an "in" FlowProperty of type Liquid. To signify that only water flows between the pump and the tank, we can specify an ItemFlow of type Water on the connector.

One can label an ItemFlow with the classifiers of the items that may be conveyed. For example: a label Water would imply that instances of Water might be transmitted over this ItemFlow. In addition, if the item flow identifies an item property, then one can label the item flow with the item property. For example, a label of "liquid: Water" means Water items might flow and these items are the values of the property "liquid," i.e., the values of the "liquid" item property are the instances of Water flowing at any given time. Item properties are owned by the common (possibly indirect) owner of the source and target of the item flow, rather than by the source and target types, as flow properties are.

Item flows on connectors shall be compatible with flow properties of the blocks usages at each end of the connector, if any. The direction of the item flow shall be compatible with the direction of flow specified by the flow properties. (See 9.3.2.12 and 9.3.2.13 about flow property direction.) Each classifier of conveyed items on an item flow shall be the same as, a specialization of, or a generalization of at least one flow property type on each end of the connected block usages (or their accessible nested block usages recursively, see 9.3.2.8 about encapsulated blocks). The target flow property type shall be the same as, or a generalization of, a classifier of the item flow or the source flow property type, whichever is more specialized. (See 9.3.2.13, for tighter constraints on flow property types across connectors without item flows.)

Attributes

- itemProperty : Property [0..1]
  An optional property that relates the flowing item to the instances of the connectors enclosing block. This property is applicable only for item flows realized by connectors. The itemProperty attribute has no values if the item flow is realized by an Association.

Association Ends

- base_InformationFlow : InformationFlow [1]

Constraints

- 1_source_and_target_linked
  A Connector or an Association, or an inherited Association shall exist between the source and the target of the InformationFlow.

```
let target: UML::NamedElement = self.base_InformationFlow.informationTarget
->any(true) in let targets: Set(UML::NamedElement) = if
target.oclIsKindOf(UML::Classifier) then target.oclAsType(UML::Classifier)
->closure(general)->including(target) else target->asSet() endif in let source:
UML::NamedElement = self.base_InformationFlow.informationSource->any(true) in let
sources: Set(UML::NamedElement) = if source.oclIsKindOf(UML::Classifier) then
source.oclAsType(UML::Classifier)->closure(general)->including(source) else source
->asSet() endif in UML::Association.allInstances()->exists(a | a.memberEnd
->intersection(targets)->notEmpty() and a.memberEnd->intersection(sources)
->notEmpty()) or UML::Connector.allInstances()->exists(c | c.end
->intersection(targets)->notEmpty() and c.end->intersection(sources)->notEmpty())
```

- 2_type_restricted
  An ItemFlow itemProperty shall be typed by a ValueType, Block, or Signal.

```
ValueType.allInstances().base_DataType->includes(self.itemProperty.type) or
Block.allInstances().base_Class->includes(self.itemProperty.type) or
UML::Signal.allInstances()->includes(self.itemProperty.type)
```

- 3_itemproperty_common_owner
  If itemProperty has a value it shall be a property of the common (possibly indirect) owner of the source and the target.

```
self.itemProperty->notEmpty() implies (let target: UML::Element =
self.base_InformationFlow.informationTarget->any(true) in let source: UML::Element =
self.base_InformationFlow.informationSource->any(true) in
target.oclIsKindOf(UML::Property) and source.oclIsKindOf(UML::Property) and let
owners: Set(UML::Classifier) = target->closure(owner)->select(o1 |
o1.oclIsKindOf(UML::Classifier))->asSet() ->intersection(source->closure(owner)-
>select(o2 | o2.oclIsKindOf(UML::Classifier))).oclAsType(UML::Classifier)->asSet() in
owners.attribute->flatten()->includes(self.itemProperty))
```

- 4_association_xor_itemproperty
  itemProperty shall not have a value if the item flow is realized by an Association.

```
self.base_InformationFlow.realization->exists(r | r.oclIsKindOf(UML::Association))
implies self.itemProperty->isEmpty()
```

- 5_same_type
  If an ItemFlow has an itemProperty, one of the classifiers of conveyed items shall be the same as the type of the item property.

```
self.itemProperty->notEmpty() implies self.base_InformationFlow.conveyed
->includes(self.itemProperty.type)
```

- 6_same_name
  If an ItemFlow has an itemProperty, its name shall be the same as the name of the item flow.

```
self.itemProperty->notEmpty() implies self.itemProperty.name =
self.base_InformationFlow.name
```

9.3.2.13 ProxyPort

Description

Proxy ports identify features of the owning block or its internal parts that are available to external blocks through external connectors to the ports. They do not specify a separate element of the system from the owning block or internal parts. Actions on features of a proxy port have the same effect as if they were acting on features of the owning block or internal parts the port stands in for, and changes to features of the owning block or internal parts that the proxy port makes available to external blocks are visible to those blocks via connectors to the port. (This applies to provided features; for required features, see Section 9.3.2.10.) Proxy ports do not specify their own behaviors or internal parts, and shall be typed by interface blocks. Their nested ports shall also be proxy ports.

A completely specified proxy port shall describe how any interaction through the port is handled or initiated. This can be achieved in several ways. For instance by making it behavioral, by binding it to a fully specified internal part or by having all its properties individually bound to internal parts. However, blocks can be defined with non-behavioral proxy ports that do not have internal connectors, with the expectation that these will be added in specialized blocks. Internal connectors to ports are the ones inside the ports owner (specifically, they are the ones that do not have a UML partwithPort on the connector end linked to the port, assuming NestedConnectorEnd is not applied to that end, or if NestedConnectorEnd is applied to that end, they are the connectors that have only ports in the property path of that end). The rest of the connectors linked to a port are external.

Proxy ports can be connected to internal parts or ports on internal parts, identifying features on those parts or ports that are available to external blocks. When a proxy port is connected to a single internal part, the connector shall be a binding connector, or have the same semantics as a binding connector (the value of the proxy port and the connected internal part are the same; links of associations typing the connector are between all objects and themselves, and no others). When a proxy port is connected to multiple internal parts, the connectors have the same semantics as a single binding connector to an aggregate of those parts, supporting all their features, and treating flows and invocations from outside the aggregate as if they were to those parts, and flows and invocations it receives from those parts as if they were to the outside. This aggregate is not a separate element of the system, and only groups the internal parts for purposes of binding to the proxy port. Internal connectors to proxy ports can be typed by association blocks, including when the connector is binding.

Association Ends

- base_Port : Port [1]

Constraints

- `1_not_fullport`
  Proxy ports shall not also be full ports. This applies even if some of the stereotypes are on subsetted or redefined ports.

```
FullPort.allInstances()->excludes(self.base_Port)
```

- `2_interfaceblock`
  Proxy ports shall only be typed by interface blocks.

```
InterfaceBlock.allInstances().base_Class->includes(self.base_Port.type)
```

- `3_subports_are_proxyports`
  Ports owned by the type of a proxy port shall be proxy ports.

```
ProxyPort.allInstances().base_Port->includesAll(self.base_Port.class.ownedPort)
```

9.3.2.14 TriggerOnNestedPort

Description

This extends trigger to support nested ports. It identifies a nested port by a multi-level path of ports from the object receiving the triggering events. It is not applicable to full ports.

Generalizations

- ElementPropertyPath (from Blocks)

Association Ends

- base_Trigger : Trigger [1]
  (redefines: ElementPropertyPath::base_Element)
- onNestedPort : Port [1..*]
  Gives a series of ports that identifies a port on which the event is occurring, in the context of a block in which the trigger is used. The ordering of ports is from a port of the receiving object, through a port of each intermediate block that types the preceding port, ending in a property with a type that owns or inherits the port given by the port property of the trigger. The port property is not included in the onNestedPort list. The same port might appear more than once because a block can own a port with the same block as a type, or another block that has the same property.
  (redefines: ElementPropertyPath::propertyPath)

Constraints

- 1_single_proxyport
  The port property of the stereotyped trigger shall have exactly one value, and the value cannot be a full port.

```
self.base_Trigger.port->size()=1 and FullPort.allInstances().base_Port
->excludes(self.base_Trigger.port)
```

- 2_no_fullport
  The values of the onNestedPort property shall not be full ports.

```
FullPort.allInstances().base_Port->excludesAll(self.onNestedPort)
```

- 3_onnestedport_first_owned_by_context
  The port at the first position in the onNestedPort list shall be owned by a block in which the trigger is used, or one of the blocks generalizations.

```
let theContext: UML::Classifier = if self.base_Trigger.owner.oclIsKindOf(UML::Action)
then self.base_Trigger.owner.oclAsType(UML::Action)._'context'.oclAsType(UML::Class)
else
self.base_Trigger.owner.oclAsType(UML::Transition).containingStateMachine()._'context
'.oclAsType(UML::Class) endif in let owners: Set(UML::Classifier) = theContext
->closure(general)->including(theContext) in owners->includes(self.onNestedPort
->first().owner)
```

- 4_path_consistency
  The port at each successive position of the onNestedPort attribute, following the first position, shall be owned by the Block that types the port at the immediately preceding position, or a generalization of the Block.

```
self.onNestedPort->size() >1 implies self.onNestedPort->subSequence(2,
self.onNestedPort->size())->forAll(p |
let np: UML::Port = self.onNestedPort->at(self.onNestedPort->indexOf(p)-1) in
let owners: Set(UML::Classifier) = np.type.oclAsType(UML::Classifier)
->including(np.type.oclAsType(UML::Classifier)) in
owners->includes(p.owner))
```

- 5_onnestedport_last_type_owns_trigger_port
  The type of the port at the last position of the onNestedPort list must own or inherit the port of the stereotyped trigger.

```
self.onNestedPort->last().type.oclAsType(UML::Classifier).allFeatures()
->includes(self.base_Trigger.port)
```

9.3.2.15 ~InterfaceBlock

Description

The ~InterfaceBlock stereotype (shall be pronounced: "conjugated interface block") is a specialization of InterfaceBlock that has the same features as its original InterfaceBlock except that its DirectedFeatures and FlowProperties are reversed (conjugated), for example, in flow properties are conjugated as out flow properties and provided features are conjugated as required features. Conjugation is specified by a constraint giving the features of ~InterfaceBlocks according to those of their original InterfaceBlocks (see the Constraints subsection below). It is expected that tools conforming to this specification automatically create features of ~InterfaceBlocks.

Generalizations

- InterfaceBlock (from PortsAndFlows)

Attributes

- original : InterfaceBlock [1]
  The InterfaceBlock that this is a conjugation of.

Operations

- areConjugated (in df1 : DirectedFeature, in df2 : DirectedFeature) : Boolean [1]
  DirectedFeature overloaded version of the areConjugated query used for specifying the inverted_feature invariant that checks whether one feature definition is the conjugated definition of the other.

```
bodyCondition:
if (df1.oclIsUndefined()) then
(not df2.oclIsUndefined() and df2.featureDirection = FeatureDirectionKind::required)
else if (df2.oclIsUndefined()) then
(not df1.oclIsUndefined() and df1.featureDirection = FeatureDirectionKind::required)
else
(df1.featureDirection = FeatureDirectionKind::provided and df2.featureDirection =
FeatureDirectionKind::required)
or (df1.featureDirection = FeatureDirectionKind::required and df2.featureDirection =
```

```
FeatureDirectionKind::provided)
or (df1.featureDirection = FeatureDirectionKind::providedRequired and
df2.featureDirection = FeatureDirectionKind::providedRequired)
endif endif
```

- areConjugated (in fp1 : FlowProperty, in fp2 : FlowProperty) : Boolean [1]
  FlowProperty overloaded version of the areConjugated query used for specifying the inverted_feature invariant that check whether one feature definition is the conjugated definition of the other

```
bodyCondition:
(fp1.direction = FlowDirectionKind::_in and fp2.direction = FlowDirectionKind::out)
or (fp1.direction = FlowDirectionKind::out and fp2.direction =
FlowDirectionKind::_in)
or (fp1.direction = FlowDirectionKind::inout and fp2.direction =
FlowDirectionKind::inout)
```

- areConjugated (in r1 : Reception, in r2 : Reception) : Boolean [1]
  Reception overloaded version of the areConjugated query used for specifying the inverted_feature invariant that check whether one feature definition is the conjugated definition of the other.

```
bodyCondition:
let df1: DirectedFeature = DirectedFeature.allInstances()->any(base_Feature = r1) in
let df2: DirectedFeature = DirectedFeature.allInstances()->any(base_Feature = r2) in
r1.concurrency = r2.concurrency
and r1.isAbstract = r2.isAbstract
and r1.ownedParameterSet->forAll(ps1 | r2.ownedParameterSet->exists(ps2 |
areSameParameterSets(r1, ps1, r2, ps2)))
and haveSameSignatures(r1, r2)
and r1.signal = r2.signal
and areConjugated(df1, df2)
```

- areConjugated (in o1 : Operation, in o2 : Operation) : Boolean [1]
  Operation overloaded version of the areConjugated query used for specifying the inverted_feature invariant that check whether one feature definition is the conjugated definition of the other.

```
bodyCondition:
let df1: DirectedFeature = DirectedFeature .allInstances()->any(base_Feature = o1) in
let df2: DirectedFeature = DirectedFeature .allInstances()->any(base_Feature = o2) in
o1.concurrency = o2.concurrency
and o1.isAbstract = o2.isAbstract
and o1.ownedParameterSet->forAll(ps1 | o2.ownedParameterSet->exists(ps2 |
areSameParameterSets(o1, ps1, o2, ps2)))
and areSameConstraintSets(o1.bodyCondition->asSet(), o2.bodyCondition->asSet())
and areSameConstraintSets(o1.precondition, o2.precondition)
and areSameConstraintSets(o1.postcondition, o2.postcondition)
and haveSameSignatures(o1, o2)
and o1.raisedException->forAll(e1 | o2.raisedException->exists(e2 | e2 = e1))
and o1.isQuery = o2.isQuery
and areConjugated(df1, df2)
```

- areConjugated (in p1 : Property, in p2 : Property) : Boolean [1]
  Property overloaded version of the areConjugated query used for specifying the inverted_feature invariant that checks whether one feature definition is the conjugated definition of the other.

```
bodyCondition:
let fp1: FlowProperty = FlowProperty.allInstances()->any(base_Property = a1) in
let fp2: FlowProperty = FlowProperty.allInstances()->any(base_Property = a2) in
let df1: DirectedFeature = DirectedFeature .allInstances()->any(base_Feature = a1) in
let df2: DirectedFeature = DirectedFeature .allInstances()->any(base_Feature = a2) in
a1.name = a2.name
and a1.type = a2.type
and a1.isStatic = a2.isStatic
and a1.isOrdered = a2.isOrdered
and a1.isUnique = a2.isUnique
and a1.lower = a2.lower
and a1.upper = a2.upper
and a1.isReadOnly = a2.isReadOnly
and a1.aggregation = a2.aggregation
and a1.isDerived = a2.isDerived
and a1.isDerivedUnion = a2.isDerivedUnion
and a1.isID = a2.isID
and ((not fp1.oclIsUndefined() and not fp2.oclIsUndefined() and areConjugated(fp1,
fp2))
or
(fp1.oclIsUndefined() and fp2.oclIsUndefined()))
and ((not df1.oclIsUndefined() and not df2.oclIsUndefined() and areConjugated(df1,
df2))
or (df1.oclIsUndefined() and df2.oclIsUndefined()))
```

- areSameConstraintSets (in cs1 : Constraint, in cs2 : Constraint) : Boolean [1]
  The areSameConstraintSets query is used for specifying the inverted_feature invariant. It checks whether two sets of constraints are equivalent.

```
bodyCondition:
(cs1->isEmpty() and cs2->isEmpty())
or (cs1->size() = cs2->size()
and cs1->forAll(c1 | cs1->exists(c2 | c2.name = c1.name
and c2.specification.booleanValue()=true implies c1.specification.booleanValue()=true
and c2.specification.booleanValue()=false implies
c1.specification.booleanValue()=false)))
```

- areSameParameterSets (in ps1 : ParameterSet, in ps2 : ParameterSet) : Boolean [1]
  The areSameParameterSets query is used for specifying the inverted_feature invariant. It checks whether two sets of parameters are identical.

```
bodyCondition:
(ps1->isEmpty() and ps2->isEmpty())
or (ps1->size() = ps2->size()
and areSameConstraintSets(ps1.condition, ps2.condition
and ps1.parameter->forAll(p1 | ps2.parameter->exists(p2 |
bf1.ownedParameter->indexOf(p1) = bf2.ownedParameter->indexOf(p2)))))
```

- haveSameSignatures (in bf1 : BehavioralFeature, in bf2 : BehavioralFeature) : Boolean [1]
  The areSameConstraintSignatures query is used for specifying the inverted_feature invariant. It checks whether two behavioral features have the same signature.

```
bodyCondition:
bf1.name = bf2.name
```

```
and bf1.ownedParameter->size() = bf2.ownedParameter->size()
and bf1.ownedParameter->forAll(p1 | let p2: UML::Parameter = bf2.ownedParameter
->at(bf1.ownedParameter->indexOf(p1)) in
p1.name = p2.name
and p1.type = p2.type
and p1.direction = p2.direction
and p1.isOrdered = p2.isOrdered
and p1.isUnique = p2.isUnique
and p1.lower = p2.lower
and p1.upper = p2.upper
and p1.effect = p2.effect
and p1.isException = p2.isException
and p1.isStream = p2.isStream)
```

Constraints

- `enforced_name`
  The name of an ~InterfaceBlock shall be the name of its original InterfaceBlock with a tilde ("~") character prepended

```
self.base_Class.name = '~'+self.original.base_Class.name
```

- `inverted_features`
  An ~InterfaceBlock has same features and owned rules than its original InterfaceBlock except that – where applicable – both its DirectedFeatures and FlowProperties have inverted directions (i.e., are "conjugated").

```
let allAttributes: Set(UML::Property) = self.base_Class.allFeatures()
->select(oclIsKindOf(UML::Property)).oclAsType(UML::Property)->asSet() in
let allOperations: Set(UML::Operation) = self.base_Class.allFeatures()
->select(oclIsKindOf(UML::Operation)).oclAsType(UML::Operation)->asSet() in
let allReceptions: Set(UML::Reception) = self.base_Class.allFeatures()
->select(oclIsKindOf(UML::Reception)).oclAsType(UML::Reception)->asSet() in
let inheritedRules: Set(UML::Constraint) =
self.base_Class.inherit(self.base_Class.inheritedMember
->select(oclIsKindOf(UML::Constraint))).oclAsType(UML::Constraint)->asSet() in
let allRules: Set(UML::Constraint) = self.base_Class.ownedRule->union(inheritedRules)
in
let allOriginalAttributes: Set(UML::Property) =
self.original.base_Class.allFeatures()
->select(oclIsKindOf(UML::Property)).oclAsType(UML::Property)->asSet() in
let allOriginalOperations: Set(UML::Operation) =
self.original.base_Class.allFeatures()
->select(oclIsKindOf(UML::Operation)).oclAsType(UML::Operation)->asSet() in
let allOriginalReceptions: Set(UML::Reception) =
self.original.base_Class.allFeatures()
->select(oclIsKindOf(UML::Reception)).oclAsType(UML::Reception)->asSet() in
let originalInheritedRules: Set(UML::Constraint) =
self.original.base_Class.inherit(self.original.base_Class.inheritedMember
->select(oclIsKindOf(UML::Constraint))).oclAsType(UML::Constraint)->asSet() in
let allOrignalRules: Set(UML::Constraint) = self.original.base_Class.ownedRule
->union(originalInheritedRules) in

allAttributes->size() = allOriginalAttributes->size()
and allOperations->size() = allOriginalOperations->size()
and allReceptions->size() = allOriginalReceptions->size()
```

```
and (allAttributes->isEmpty() or allAttributes->forAll(a | allOriginalAttributes
->exists(oa | areConjugated(a, oa))))
and (allOperations->isEmpty() or allOperations->forAll(o | allOriginalOperations
->exists(oo | areConjugated(o, oo))))
and (allReceptions->isEmpty() or allReceptions->forAll(r | allOriginalReceptions
->exists(ro | areConjugated(r, ro))))
and areSameConstraintSets(allRules, allOrignalRules)
```

9.4 Usage Examples

9.4.1 Ports with Required and Provided Features

Figure 9-6 is a fragment of the ibd:PwrSys diagram used in the HybridSUV Sample Problem in Annex D. (The complete diagram is in Figure D.19.) The ecu:PowerControlUnit part has three ports with required and provided features, each connected to a port of another part. Each of the ports in this example is typed by a block specifying provided and required features available via connectors to the ports. For example, the ICE block specifies the provided operations setMixture and setThrottle, the provided properties RPM, temperature, and isKnocking, and required property isControlOn, as shown in Figure D.20. This block types the ctrl port of InternalCombustionEngine while its conjugation (~ICE) types the ice port of PowerControlUnit. This means the provided features of ICE are provided by the ctrl port of InternalCombustionEngine, and required by the ice port of PowerControlUnit, while the required features of ICE are required by the ctrl port of InternalCombustionEngine, and provided by the ice port of PowerControlUnit. Since the ecu:PowerControlUnit part and ice:InternalCombustionEngine part are connected via these ports, the ecu:PowerControlUnit part may invoke setThrottle and setMixture on the ice:InternalCombustionEngine part via its ice port, across the connector to the ctrl port of ice:InternalCombustionEngine. By invoking these operations, the PowerControlUnit can set the throttle and mixture of the InternalCombustionEngine. The PowerControlUnit can also read properties of the InternalCombustionEngine across the connector to find out the rpm, temperature, and whether it is knocking. Inversely, the InternalCombustionEngine can read the isControlOn property of the PowerControlUnit across the connector to determine if the unit is still operating, and possibly shut down if it is not.

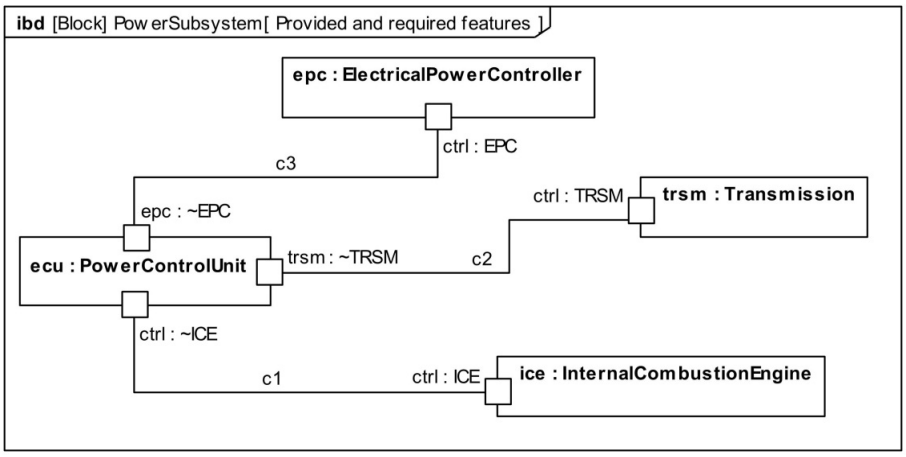


Figure 9-6: Usage example of ports with provided and required features

9.4.2 Ports and Item Flows

Figure D.25 shows the usage of ItemFlow. Here each of the item flows has an item property (fuelSupply:Fuel and fuelReturn:Fuel) that signify the actual flow of fuel across the fuel lines. We see how Fuel may flow between the FuelTankAssy and the InternalCombustionEngine. The FuelPump ejects Fuel via p1 port of FuelTankAssy, the Fuel flows across the fuelSupplyLine connector to the fuelFittingPort of InternalCombustionEngine and from there it is distributed via other ports to internal parts of the engine. Some of the fuel is returned to the FuelTankAssy from the fuelFitting port across the fuelReturnLine connector. Note that it is possible to connect a single port to multiple connectors: in this example the direction of the flow via the fuelFitting port on the external connectors is implied by the direction of the ports on the other side of the fuel lines as well as by the directions of the item flows on the fuel lines. The direction of the flow on the internal connectors is implied by the direction of the ports of the engine's internal parts.

9.4.3 Ports with Flow Properties

Figure D.22 shows a way to connect the PowerControlUnit to other parts over a CAN bus. Since connections over buses are characterized by broadcast asynchronous communications, ports with flow properties are used to connect the parts to the CAN bus. To specify the flow between the ports, we need to specify flow properties as done in Figure D.21. Here FS_ICE has three flow properties: an "out" flow property of type signal (ICEData) and two "in" flow properties of type Real. This allows the InternalCombustionEngine to transmit an ICEData signal via its fp port that will be transmitted over the CAN bus to the ice port of PowerControlUnit (a port typed by the conjugation ~FS_ICE). This single signal carries the temperature, rpm, and knockSensor information of the engine. In addition, the PowerControlUnit can set the mixture and throttle of the InternalCombustionEngine via the mixture and throttlePosition flow properties of FS_ICE.

9.4.4 Proxy and Full Ports

Modelers have the option of applying stereotypes for proxy and full ports to indicate whether ports are specifying features of their owners and internal parts (proxy), or for themselves separately (full). This is a concern when defining ports, rather than using existing blocks with ports already defined on them. Using existing blocks with ports only requires knowing the port types, because they define the features available for linking or communication with those ports via connectors. The stereotypes of proxy and full ports might be elided in these cases to simplify diagrams.

The ProxyPort and FullPort stereotypes can be applied at any level in a block taxonomy, whether on ports of the most general blocks, the most specialized, or at intermediate levels of generalization. Ports can be specialized through redefinition and subsetting if desired, as long they are not proxy and full at the same time, including the stereotypes they inherit. Figure 9-7 shows an example of a general block for an electrical plug specialized into two other blocks. The general block can be contained in its own package, for export to users of electrical plugs. The specialized blocks are for plug designers. This example has two designs, one using proxy ports and the other full. The proxy design adds internal parts exposed by the ports. The full design redefines the ports with specialized types. The same type is used for the internal parts of the proxy design and the redefined ports of the full design. The net result for the systems as-built are the same.

Modelers can apply stereotypes for proxy and full ports at any stage of model development, or not all if the stereotype constraints are not needed. Figure 9-7 happens to use unstereotyped ports on a general block distributed to users, and stereotyped ports on its specializations for implementation, but the modelers might have not used stereotypes at all, if they did not care whether the model met those constraints (such as no behaviors on proxy ports, or no internal binding connectors to full ports).

Unstereotyped ports do not commit to whether they are proxy or full, and do not prevent or dictate future application of the stereotypes, except for ports that violate constraints of the stereotypes. For example, if the port types on the general block in Figure 9-7 had behaviors defined, then the proxy specialization would be invalid. If the general ports had binding connectors to internal parts, then the full specialization would be invalid. If the general ports had both behaviors and internal binding connectors, then both specializations would be invalid. Unstereotyped ports have the basic functionality of stereotyped ones, including flow properties and nested ports, so they can be used as long as the modeler is not concerned with the distinction between proxy and full, and the constraints they impose.

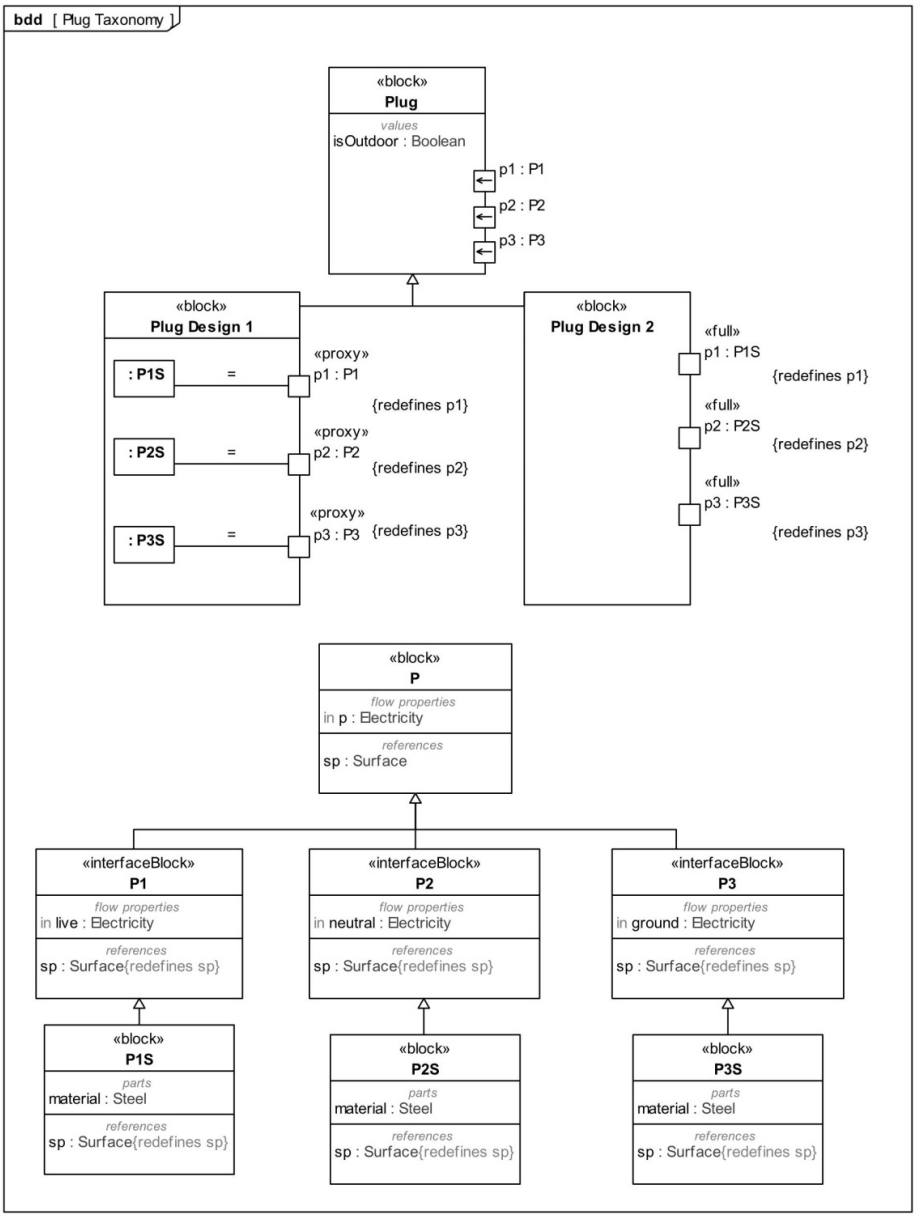


Figure 9-7: Usage example of proxy and full ports

9.4.5 Association and Port Decomposition

Figure 9-8 shows an association block Water Delivery between a bank of spigots and a faucet. The «port» keyword indicates which association ends are ports (associations use properties as ends, which can be ports). Figure 9-9 shows the internal structure of Water Delivery defining connectors between the spigots in the bank and inlets on the faucet. The participant properties identify the spigot bank and faucet being connected. The end property on the stereotype refers to the corresponding association end in Figure 9-8. The type of participant properties is shown for clarity, but is always the same as the association end type and can be elided. They are shown with dashed rectangles because they are reference properties. The internal structure connects hot and cold ports of the participants.

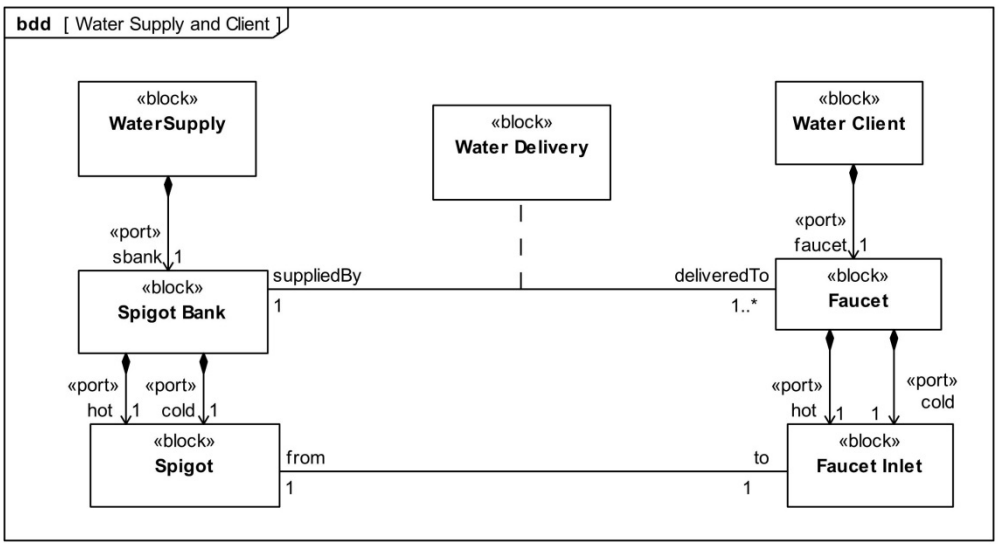


Figure 9-8: Water Delivery association block

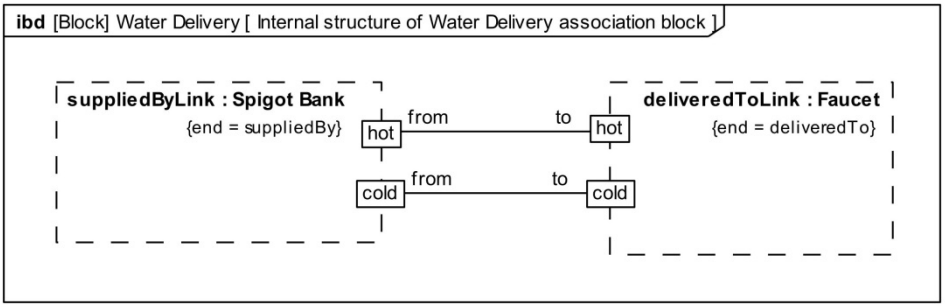


Figure 9-9: Internal structure of Water Delivery association block

Figure 9-10 shows two views of a block House with a connector of type Water Delivery. The connector in the top view "decomposes" into the subconnectors in the lower view according to the internal structure of Water Delivery. The subconnectors relate the nested ports of :WaterSupply to the nested ports of :WaterClient.

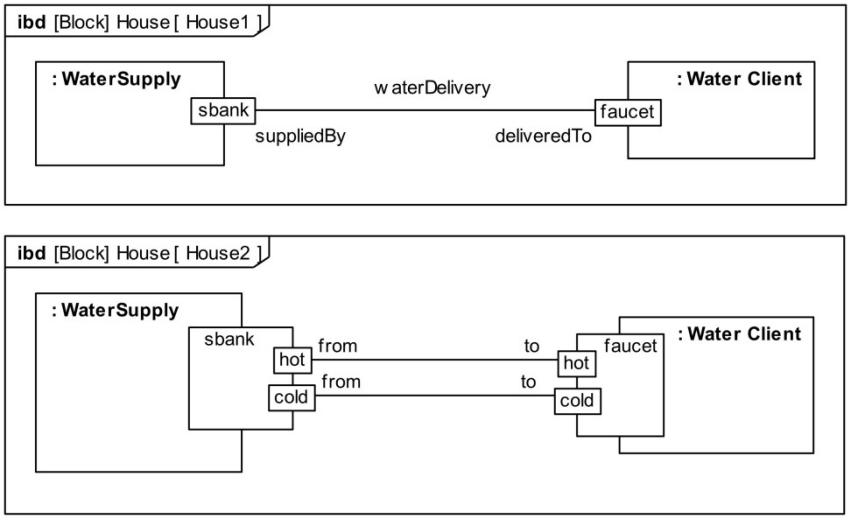


Figure 9-10: Two views of Water Delivery connector within House block

The top portion of Figure 9-11 shows specializations of the block WaterClient into Bath, Sink, and Shower. These are used as part types in the internal structure of the block House 2 shown in the lower portion of the figure. The composite connector for Water Delivery is reused three times to establish connections between spigots on the water supply and the inlets of faucets on the bath, sink, and shower.

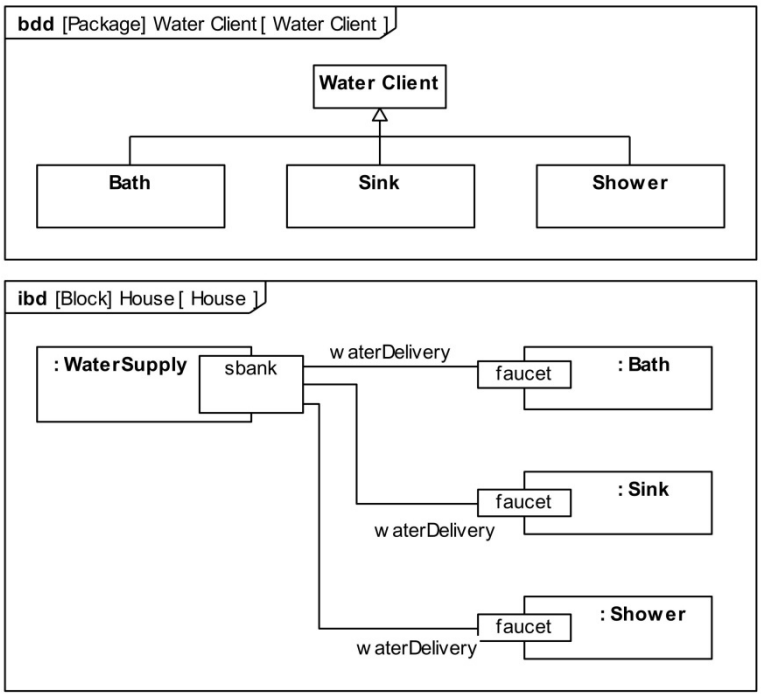


Figure 9-11: Specializations of Water Client in house example

Figure 9-12 adds a Plumbing association block for the association between Spigot and Faucet Inlet in Figure 9-11. Figure 9-13 shows the internal structure for the Plumbing association block, which includes a pipe and two fittings (the additional part and connector definitions are omitted for brevity).

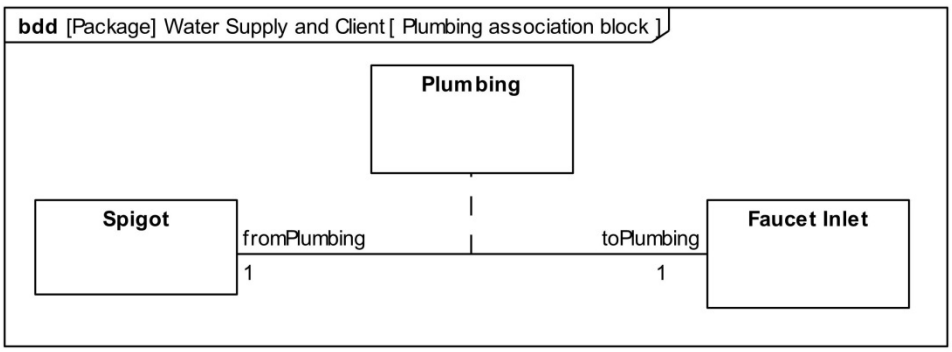


Figure 9-12: Plumbing association block

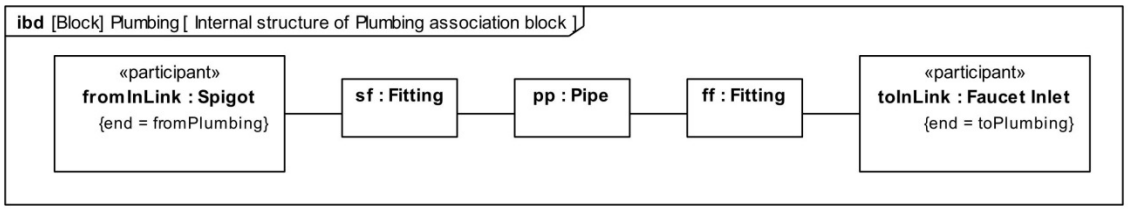


Figure 9-13: Internal structure of Plumbing association block

Figure 9-14 modifies Figure 9-9 to use Plumbing as a connector type within the Water Delivery association block. The lower connector shows its connector property explicitly, enabling the pipe it contains to be connected to a mounting bracket (the additional part and connector definitions are omitted for brevity).

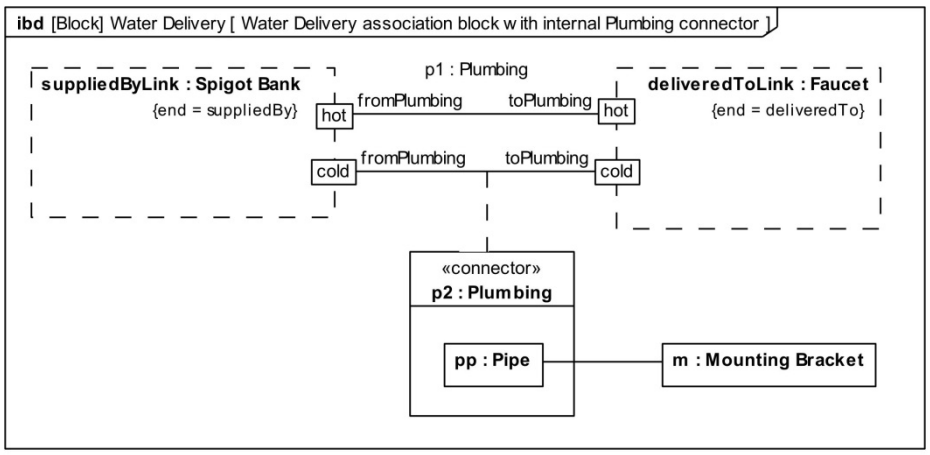


Figure 9-14: Water Delivery association block with internal Plumbing connector

9.4.6 Item Flow Decomposition

Item flows in internal block diagrams specify flows local to a block. For example, in Figure 9-15 the connector to the output of the water heater has an item flow indicating distilled water is flowing, even though the out flow property of the water heater indicates it produces water. The water heater is fed from a water distiller in this particular usage, so the modeler knows the output will always be distilled water, rather than other kinds of water. The radiator on the left requires distilled water, and its connection to the water heater is compatible because the item flow narrows the items to distilled water. Item flows can also be more general than the actual flow, as shown by the connector on the right. The water distiller produces distilled water, but the item flow is for any kind of fluid. The connection to the water heater is compatible because it accepts any kind of water, including distilled. The item flow does not require the heater to accept any kind of fluid, because the source of flow is still producing water, regardless of the generality of the item flow.

Connectors with item flows can be decomposed by association blocks that have additional item flows. The relationship between an item flow and those in the association block is determined by the modeler. Figures 9.16 and 9.17 are examples of item flow decomposition that modelers might choose, but they are not the only possible decompositions and are not required. In Figure 9-16, the item flow classifier (EnginePart) is a supertype of the classifiers of the item flows in the decomposition. The flow properties are all in the types of the nested ports, while the composing item flow summarizes the kinds of items flowing by generalization. In Figure 9-17, the item flow classifier (Engine) composes the

classifiers of the items flows in the decomposition from Figure 9-17. The port types have an additional flow property that is not in the nested ports. These are for the flow of the engine, as opposed to its parts. Constraints can be added between the flow properties for the engine and those for the parts, to indicate the flowing parts are inside the flowing engine, or are separate, for example as spare parts.

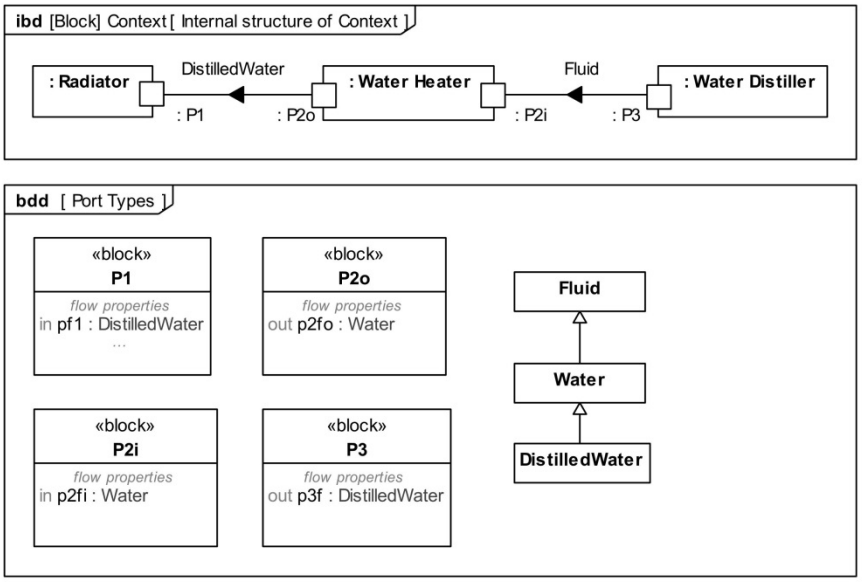


Figure 9-15: Usage example of item flows in internal block diagrams

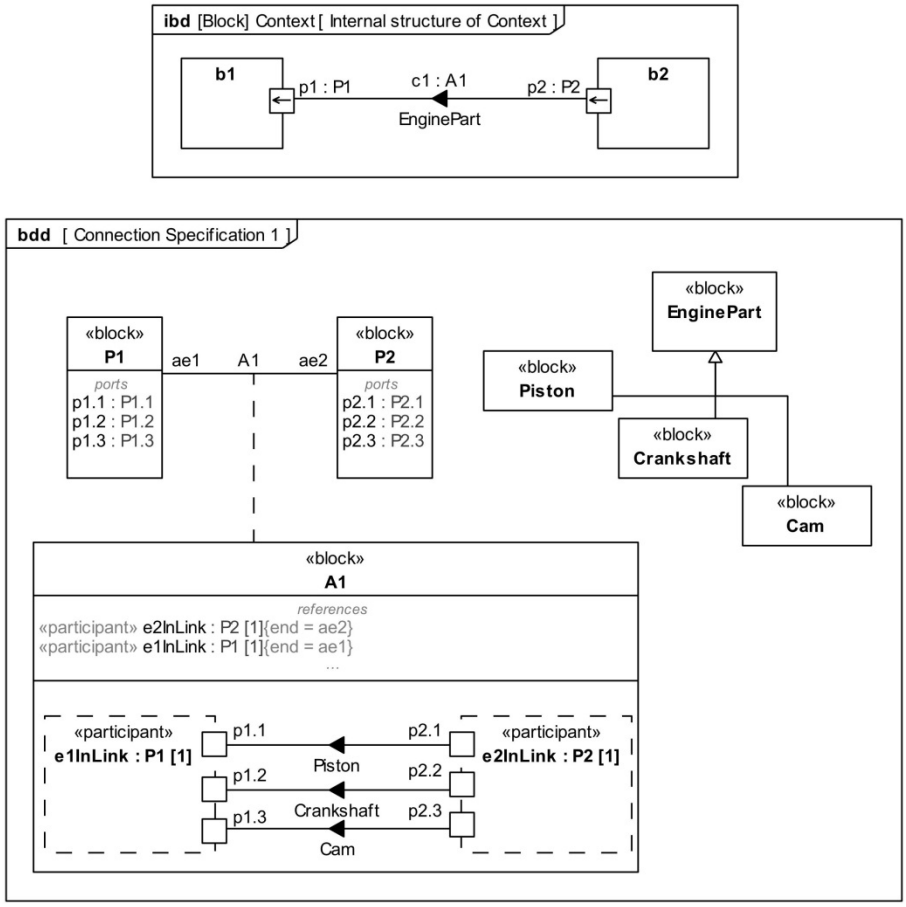


Figure 9-16: Usage example of item flow decomposition

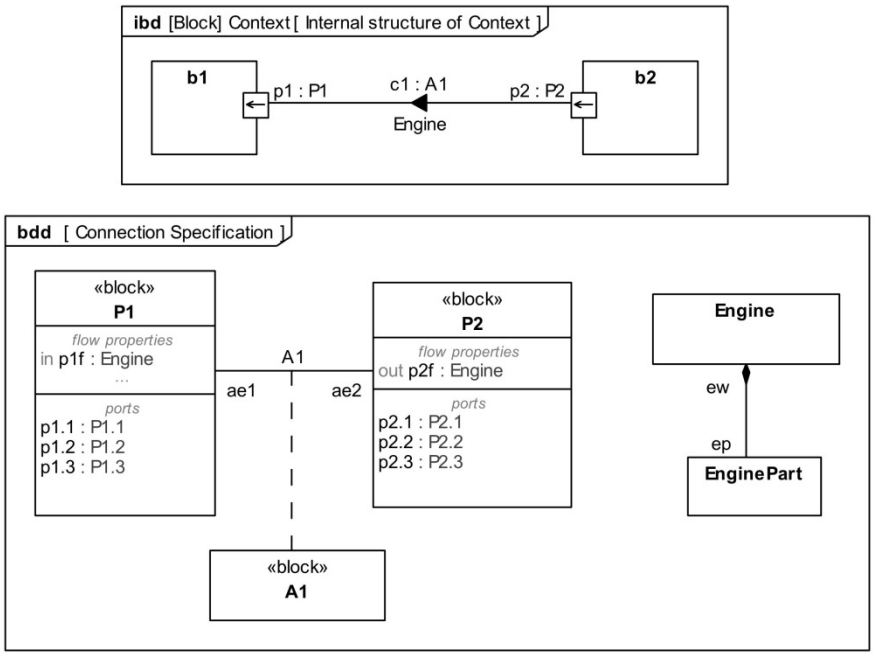


Figure 9-17: Usage example of item flow decomposition

10 Constraint Blocks

10.1 Overview

Constraint blocks provide a mechanism for integrating engineering analysis such as performance and reliability models with other SysML models. Constraint blocks can be used to specify a network of constraints that represent mathematical expressions such as {F=m*a} and {a=dv/dt}, which constrain the physical properties of a system. Such constraints can also be used to identify critical performance parameters and their relationships to other parameters, which can be tracked throughout the system life cycle.

A constraint block includes the constraint, such as {F=m*a}, and the parameters of the constraint such as F, m, and a. Constraint blocks define generic forms of constraints that can be used in multiple contexts. For example, a definition for Newton's Laws may be used to specify these constraints in many different contexts. Reusable constraint definitions may be specified on block definition diagrams and packaged into general-purpose or domain-specific model libraries. Such constraints can be arbitrarily complex mathematical or logical expressions. The constraints can be nested to enable a constraint to be defined in terms of more basic constraints such as primitive mathematical operators.

Parametric diagrams include usages of constraint blocks to constrain the properties of another block. The usage of a constraint binds the parameters of the constraint, such as F, m, and a, to specific properties of a block, such as a mass, that provide values for the parameters. The constrained properties, such as mass or response time, typically have simple value types that may also carry units, quantity kinds, or probability distributions. A pathname dot notation can be used to refer to nested properties within a block hierarchy. This allows a value property (such as an engine displacement) that may be deeply nested within a containing hierarchy (such as vehicle, power system, engine) to be referenced at the outer containing level (such as vehicle-level equations). The context for the usages of constraint blocks shall also be specified in a parametric diagram to maintain the proper namespace for the nested properties.

Time can be modeled as a property that other properties may be dependent on. A time reference can be established by a local or global clock that produces a continuous or discrete time value property. Other values of time can be derived from this clock, by introducing delays and/or skew into the value of time. Discrete values of time as well as calendar time can be derived from this global time property. SysML includes the time model from UML, but other UML specifications offer more specialized descriptions of time that may also apply to specific needs.

A state of the system can be specified in terms of the values of some of its properties. For example, when water temperature is below 0 degrees Celsius, it may change from liquid to solid state. In this example, the change in state results in a different set of constraint equations. This can be accommodated by specifying constraints that are conditioned on the value of the state property.

Parametric diagrams can be used to support trade-off analysis. A constraint block can define an objective function to compare alternative solutions. The objective function can constrain measures of effectiveness or merit and may include a weighting of utility functions associated with various criteria used to evaluate the alternatives. These criteria, for example, could be associated with system performance, cost, or desired physical characteristics. Properties bound to parameters of the objective function may have probability distributions associated with them that are used to compute expected or probabilistic measures of the system. The use of an objective function and measures of effectiveness in parametric diagrams are included in Annex E: "Non-normative Extensions."

SysML identifies and names constraint blocks, but does not specify a computer interpretable language for them. The interpretation of a given constraint block (e.g., a mathematical relation between its parameter values) shall be provided. An expression may rely on other mathematical description languages both to capture the detailed specification of

mathematical or logical relations, and to provide a computational engine for these relations. In addition, the block constraints are non-causal and do not specify the dependent or independent variables. The specific dependent and independent variables are often defined by the initial conditions, and left to the computational engine.

A constraint block is defined by a keyword of «constraint» applied to a block definition. Properties of this block define parameters of the constraint, with the exception of properties that hold internally nested usages of constraint blocks. The usage of a constraint block is distinguished from other parts by a box having rounded corners rather than the square corners of an ordinary part. A parametric diagram is a restricted form of internal block diagram that shows only the use of constraint blocks along with the properties they constrain within a context.

10.2 Diagram Elements

10.2.1 Block Definition Diagram

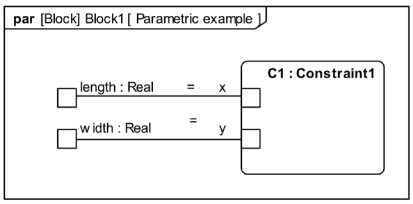
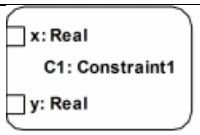
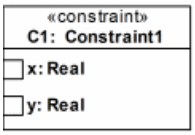
Table 10-1: Graphical nodes defined in Block Definition diagrams

Element Name	Concrete Syntax Example	Metamodel Reference
ConstraintBlock	«constraint» **ConstraintBlock1** *constraints* {{L1} x > y} nested: ConstraintBlock2 *parameters* x: Real y: Real	**SysML::ConstraintBlocks: ConstraintBlock**

10.2.2 Parametric Diagram

The diagram elements described in this sub clause are additions to the Internal Block Diagram described in Clause 8. The Parametric Diagram includes all of the notations of an Internal Block Diagram, subject only to the restrictions described in 10.3.1.2.

Table 10-2: Graphical nodes defined in Parametric diagrams

Element Name	Concrete Syntax Example	Metamodel Reference
ParametricDiagram		**SysML::ConstraintBlocks: ConstraintBlock** **SysML::Blocks::Block**
ConstraintProperty	 	**UML4SysML::Property typed by SysML::ConstraintBlocks::ConstraintBlock**

10.3 UML Extensions

10.3.1 Diagram Extensions

10.3.1.1 Block Definition Diagram

10.3.1.1.1 Constraint block definition

The «constraint» keyword on a block definition states that the block is a constraint block. An expression that specifies the constraint may appear in the constraints compartment of the block definition, using either formal statements in some language, or informal statements using text. This expression can include a formal reference to a language in braces as indicated in Table 10-1. Parameters of the constraint may be shown in a compartment with the predefined compartment label “parameters.”

10.3.1.1.2 Parameters compartment

Constraint blocks support a special form of compartment, with the label “parameters,” which may contain declarations for some or all of its constraint parameters. Properties of a constraint block should be shown either in the constraints compartment, for nested constraint properties, or within the parameters compartment.

10.3.1.2 Parametric Diagram

A parametric diagram is defined as a restricted form of internal block diagram. A parametric diagram may contain constraint properties and their parameters, along with other properties from within the internal block context. All properties that appear, other than the constraints themselves, shall either be bound directly to a constraint parameter, or contain a property that is bound to one (through any number of levels of containment).

10.3.1.2.1 Round-cornered rectangle notation for constraint property

A constraint property may be shown on a parametric diagram using a rectangle with rounded corners. This graphical shape distinguishes a constraint property from all other properties and avoids the need to show an explicit «constraint» keyword. Otherwise, this notation is equivalent to the standard form of an internal property with a «constraint» keyword shown. Compartments and internal properties may be shown within the shape just as for other types of internal properties.

10.3.1.2.2 «constraint» keyword notation for constraint property

A constraint property may be shown on a parametric diagram using a standard form of internal property rectangle with the «constraint» keyword preceding its name. Parameters are shown within a constraint property using the standard notations for internal properties.

10.3.1.2.3 Small square box notation for an internal property

A value property may optionally be shown by a small square box, with the name and other specifications appearing in a text string close to the square box. The text string for such a value property may include all the elements that could ordinarily be used to declare the property in a compartment of a block, including an optional default value. The box may optionally be shown with one edge flush with the boundary of a containing property. Placement of property boxes is purely for notational convenience, for example to enable simpler connection from the outside, and has no semantic significance. If a connector is drawn to a region where an internal property box is shown flush with the boundary of a containing property, the connector is always assumed to connect to the innermost property.

10.3.2 Stereotypes

Package Constraint Blocks

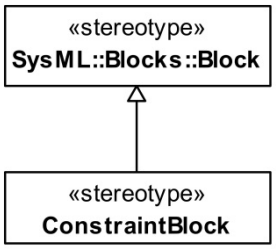


Figure 10-1: Stereotypes defined in SysML ConstraintBlocks package

10.3.2.1 ConstraintBlock

Description

A constraint block is a block that packages the statement of a constraint so it may be applied in a reusable way to constrain properties of other blocks. A constraint block typically defines one or more constraint parameters, which are

bound to properties of other blocks in a surrounding context where the constraint is used. Binding connectors, as defined in Clause 8 are used to bind each parameter of the constraint block to a property in the surrounding context. All properties of a constraint block are constraint parameters, with the exception of constraint properties that hold internally nested usages of constraint blocks.

A constraint property is a property of any block that is typed by a constraint block. It holds a localized usage of the constraint block. Binding connectors may be used to bind the parameters of this constraint block to other properties of the block that contains the usage.

Generalizations

- Block (from Blocks)

Constraints

- `1_constraintparameters_only`
  A constraint block shall not own any structural or behavioral elements beyond the properties that define its constraint parameters, constraint properties that hold internal usages of constraint blocks, binding connectors between its internally nested constraint parameters, constraint expressions that define an interpretation for the constraint block, and general-purpose model management and crosscutting elements.

```
-- Cannot be expressed in OCL
```

- `3_composite`
  Any property of a block that is typed by a ConstraintBlock shall have composite aggregation.

```
self.base_Class.ownedAttribute->forAll(p| p.isComposite)
```

10.4 Usage Examples

10.4.1 Definition of Constraint Blocks on a Block Definition Diagram

Constraint blocks can only be defined on a block definition diagram or a package diagram, where they shall have the «constraint» keyword shown. The strings in braces in the compartment labeled “constraints” are ordinary UML constraints, using a special compartment to hold the constraint. This is shown in Figure D.34. These particular constraints are specified only in an informal language, but a more formal language such as OCL or MathML could also be used. The compartment labeled “parameters” shows the parameters of this constraint, which are bound on the parametric diagram.

10.4.2 Usage of Constraint Blocks on a Parametric Diagram

Figure D.32 shows the use of constraint properties on a parametric diagram. This diagram shows the use of nested property references to the properties of the parts; parametric diagrams can make use of the nested property name notation to refer to multiple levels of nested property containment, as shown in this example. A parametric diagram is similar to an internal block diagram with the exception that the only connectors that may be shown are binding connectors. The Sample Problem in Annex D provides definitions of the containing EconomyContext block for which this parametric diagram is shown.

This page intentionally left blank.

BEHAVIORAL CONSTRUCTS

This page intentionally left blank.

11 Activities

11.1 Overview

Activity modeling emphasizes the inputs, outputs, sequences, and conditions for coordinating other behaviors. It provides a flexible link to blocks owning those behaviors. The following is a summary of the SysML extensions to UML Activity diagrams. For additional information, see extensions for Enhanced Functional Flow Block Diagrams in E.2, Activity Diagram Extensions.

11.1.1 Control as Data

SysML extends control in activity diagrams as follows:

- In UML Activities, control can only enable actions to start. SysML extends control to support disabling of actions that are already executing. This is accomplished by providing a model library with a type for control values that are treated like data (see ControlValueKind in Figure 11-9).
- A control value is an input or output of a control operator, which is how control acts as data. A control operator can represent a complex logical operation that transforms its inputs to produce an output that controls other actions (see ControlOperator in Figure 11-8).

11.1.2 Continuous Systems

SysML provides extensions that might be very loosely grouped under the term “continuous,” but are generally applicable to any sort of distributed flow of information and physical items through a system. These are:

- Restrictions on the rate at which entities flow along edges in an activity, or in and out of parameters of a behavior (see Rate in Figure 11-8). This includes both discrete and continuous flows, either of material, energy, or information. Discrete and continuous flows are unified under rate of flow, as is traditionally done in mathematical models of continuous change, where the discrete increment of time approaches zero.
- Extension of object nodes, including pins, with the option for newly arriving values to replace values that are already in the object nodes (see Overwrite in Figure 11-8). SysML also extends object nodes with the option to discard values if they do not immediately flow downstream (see NoBuffer in Figure 11-8). These two extensions are useful for ensuring that the most recent information is available to actions by indicating when old values should not be kept in object nodes, and for preventing fast or continuously flowing values from collecting in an object node, as well as modeling transient values, such as electrical signals.

11.1.3 Probability

SysML introduces probability into activities as follows (see Probability in Figure 11-8):

- Extension of edges with probabilities for the likelihood that a value leaving the decision node or object node will traverse an edge.
- Extension of output parameter sets with probabilities for the likelihood that values will be output on a parameter set.

11.1.4 Activities as Blocks

In UML, all behaviors including activities are classes, and their instances are executions. Behaviors can appear on block definition diagrams, and participate in generalization and associations. SysML clarifies the semantics of composition association between activities, and between activities and the type of object nodes in the activities, and defines consistency rules between these diagrams and activity diagrams. See 11.3.1.1, Activity.

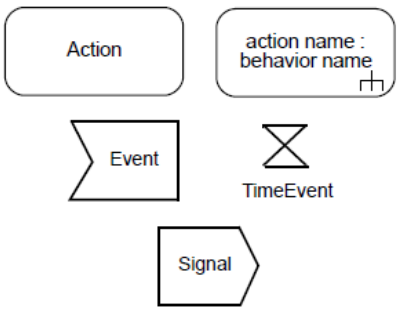
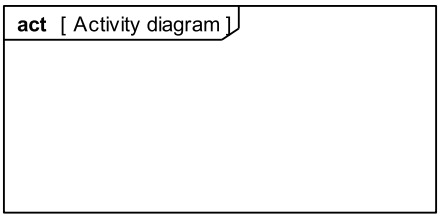
11.1.5 Timelines

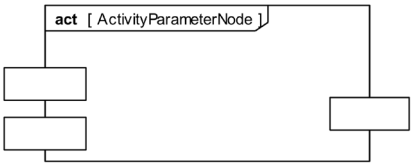
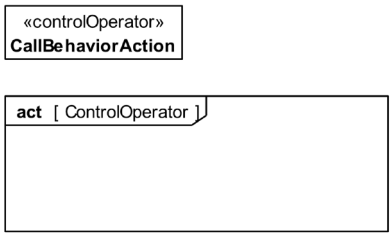
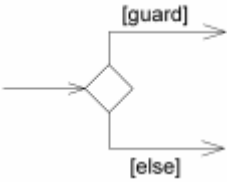
The simple time model in UML can be used to represent timing and duration constraints on actions in an activity model. These constraints can be notated as constraint notes in an activity diagram. Although the UML 2 timing diagram was not included in this version of SysML, it can complement SysML behavior diagrams to notate this information. More sophisticated SysML modeling techniques can incorporate constraint blocks from Clause 10, “Constraint Blocks” to specify resource and related constraints on the properties of the inputs, outputs, and other system properties. (Note: refer to 11.3.1.4, ObjectNode, Variables, and Parameters for constraining properties of object nodes).

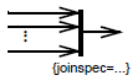
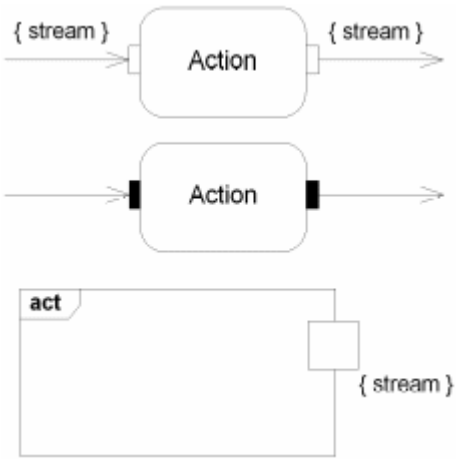
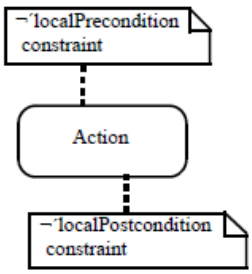
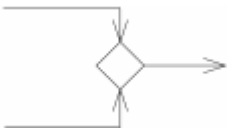
11.2 Diagram Elements

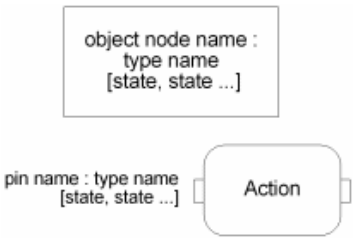
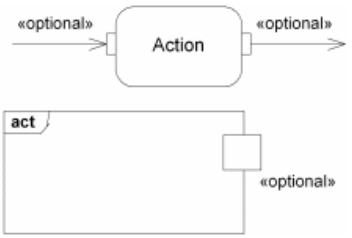
11.2.1 Activity Diagram

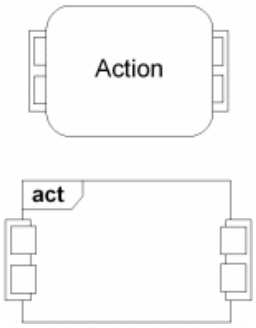
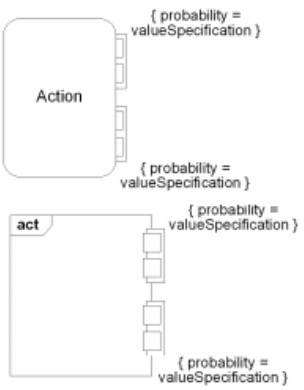
Table 11-1: Graphical notation of activity diagrams

Notation Name	Concrete Syntax	Abstract Syntax Reference
Action, CallBehaviorAction, AcceptEventAction, SendSignalAction		**UML4SysML::Action** **UML4SysML::CallBehaviorAction** **UML4SysML::AcceptEventAction** **UML4SysML::SendSignalAction**
Activity Frame and Heading		**UML4SysML::Activity**

Notation Name	Concrete Syntax	Abstract Syntax Reference
ActivityFinal		**UML4SysML::ActivityFinalNode**
ActivityNode	See ControlNode and ObjectNode	**UML4SysML::ActivityNode**
ActivityParameterNode		**UML4SysML::ActivityParameter Node**
ControlNode	See DecisionNode, FinalNode, ForkNode, InitialNode, JoinNode, and MergeNode	**UML4SysML::ControlNode**
ControlOperator		**SysML::Activities::ControlOperator**
DecisionNode		**UML4SysML::DecisionNode**
FinalNode	See ActivityFinal and FlowFinal	**UML4SysML::FinalNode**
FlowFinal		**UML4SysML::FlowFinalNode**
InitialNode		**UML4SysML::InitialNode**

Notation Name	Concrete Syntax	Abstract Syntax Reference
JoinNode		**UML4SysML::JoinNode**
isControl		**UML4SysML::Pin.isControl**
isStream		**UML4SysML::Parameter.isStream**
Local pre- and postconditions		**UML4SysML::Action.local Precondition,** **UML4SysML::Action.local Postcondition**
MergeNode		**UML4SysML::MergeNode**

Notation Name	Concrete Syntax	Abstract Syntax Reference
NoBuffer		**SysML::Activities::NoBuffer**
ObjectNode		**UML4SysML::ObjectNode and its children,** **SysML::Activities::ObjectNode**
Optional		**SysML::Activities::Optional**
OverWrite		**SysML::Activities::Overwrite**

Notation Name	Concrete Syntax	Abstract Syntax Reference
ParameterSet		**SysML::Activities::ParameterSet**
Portability		**SysML::Activities::Portability**

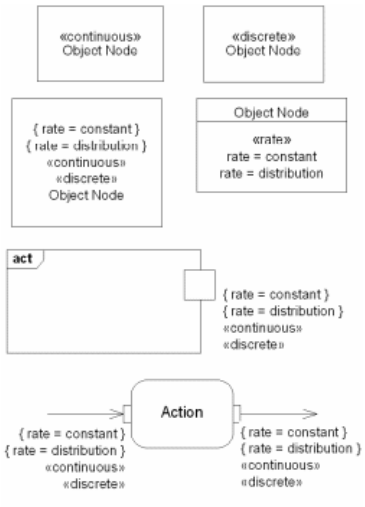
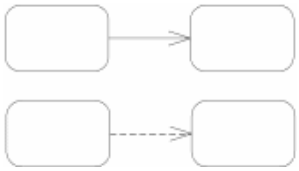
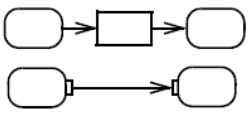
Notation Name	Concrete Syntax	Abstract Syntax Reference
Rate		**SysML::Activities::Rate** **SysML::Activities::Continuous,** **SysML::Activities::Discrete**

Table 11-2: Graphical paths included in activity diagrams

Path Name	Concrete Syntax	Abstract Syntax Reference
ActivityEdge	**See ControlFlow and ObjectFlow**	**UML4SysML::ActivityEdge**
ControlFlow		**UML4SysML::ControlFlow** **SysML::Activities::ControlFlow**
ObjectFlow		**UML4SysML::ObjectFlow**

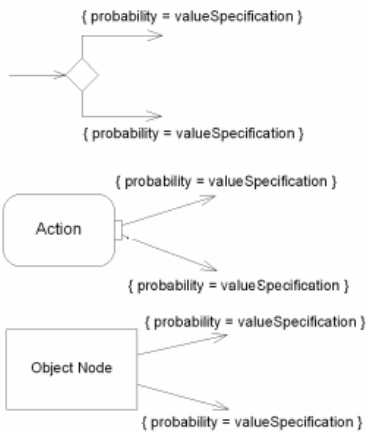
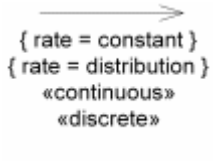
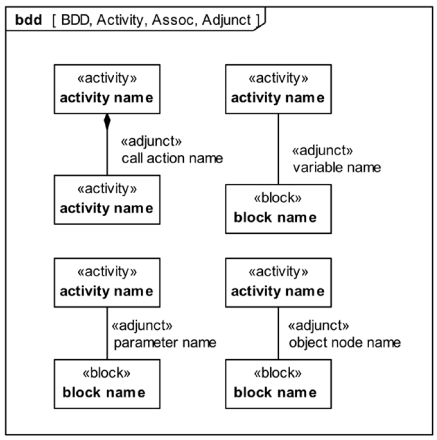
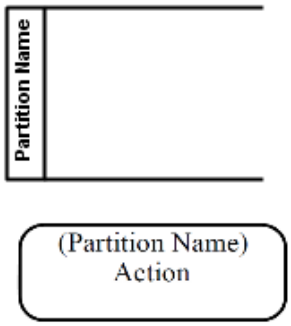
Path Name	Concrete Syntax	Abstract Syntax Reference
Probability		**SysML::Activities::Probability**
Rate		**SysML::Activities::Rate,** **SysML::Activities::Continuous,** **SysML::Activities::Discrete**

Table 11-3: Other graphical elements included in activity diagrams

Element Name	Concrete Syntax	Abstract Syntax Reference
In Block Definition Diagrams, Activity, Association, AdjunctProperty		**UML4SysML::Activity, UML4SysML::Association, SysML::Blocks::AdjunctProperty**
ActivityPartition		**UML4SysML::ActivityPartition**
InterruptibleActivity Region		**UML4SysML::InterruptibleActivity Region**
StructuredActivityNode		**UML4SysML::StructuredActivity Node**

11.3 UML Extensions

11.3.1 Diagram Extensions

The following specify diagram extensions to the notations defined in Clause 17, “Profiles & Model Libraries.”

11.3.1.1 Activity

11.3.1.1.1 Notation

In UML, all behaviors are classes, including activities, and their instances are executions of the activity. This follows the general practice that classes define the constraints under which the instances must operate. Creating an instance of an activity causes the activity to start executing, and vice versa. Destroying an instance of an activity terminates the corresponding execution, and vice versa. Terminating an execution also terminates the execution of any other activities that it invoked synchronously, that is, expecting a reply.

Activities as blocks can have associations between each other, including composition associations. Composition means that destroying an instance at the whole end destroys instances at the part end. When composition is used with activity blocks, the termination of execution of an activity on the whole end will terminate executions of activities on the part end of the links.

Combining the two aspects above, when an activity invokes other activities, they can be associated by a composition association, with the invoking activity on the whole end, and the invoked activity on the part end. If an execution of an activity on the whole end is terminated, then the executions of the activities on the part end are also terminated. The upper multiplicity on the part end restricts the number of concurrent synchronous executions of the behavior that can be invoked by the containing activity. See Constraints below.

Activities in block definition diagrams appear as regular blocks, except the «activity» keyword may be used to indicate the Block stereotype is applied to an activity, as shown in Figure 11-1. See example in 11.4, Usage Examples. This provides a means for representing activity decomposition in a way that is similar to classical functional decomposition hierarchies. Properties with AdjunctProperty applied, where the principal of the AdjunctProperties are call actions, including call behavior actions, can be used as the part end of the associations. See 8.3.2.2 for constraints when AdjunctProperty is used with call actions. Activities in block definition diagrams can also appear with the same notation as CallBehaviorAction, except the rake notation can be omitted, if desired. Also see use of activities in block definition diagrams that include ObjectNodes.

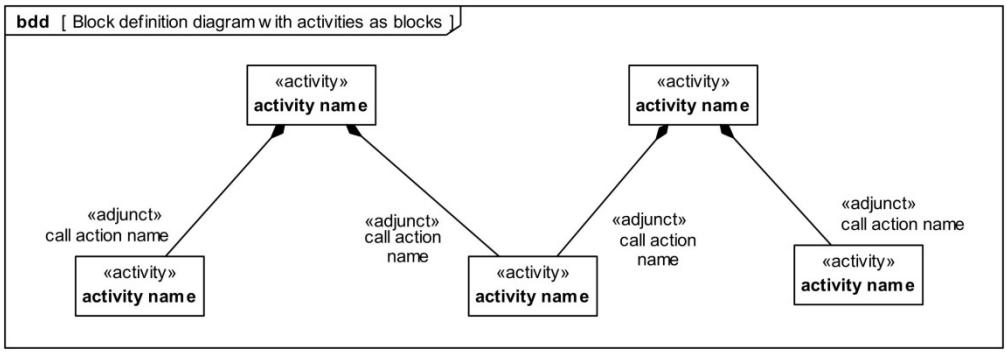


Figure 11-1: Block definition diagram with activities as blocks

Activities as blocks can have properties of any kind, including value properties. Activity block properties have all the capabilities of other properties, including that value properties can be bound to parameters in constraint blocks by binding connectors.

11.3.1.2 CallBehaviorAction

Stereotypes applied to behaviors may appear on the notation for CallBehaviorAction when invoking those behaviors, as shown in Figure 11-2.

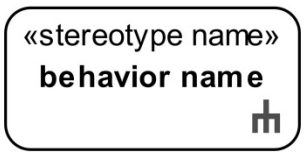


Figure 11-2: CallBehaviorAction notation.with behavior stereotype

CallBehaviorActions in activity diagrams may optionally show the action name with the name of the invoked behavior using the colon notation shown in Figure 11-3.

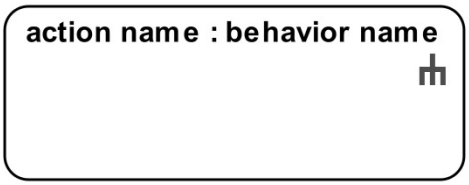


Figure 11-3: CallBehaviorAction notation.with action name ControlFlow

11.3.1.3 ControlFlow

11.3.1.3.1 Presentation Option

Control flow may be notated with a dashed line and stick arrowhead, as shown in Figure 11-4.

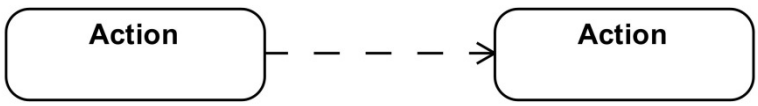


Figure 11-4: Control flow notation

11.3.1.4 ObjectNode, Variables, and Parameters

11.3.1.4.1 Notation

See 11.3.1.1, Activity with regard to activities appearing in block definition diagrams. Associations can be used between activities and classifiers (blocks or value types) that are the type of object nodes, variables, or parameters in the activity,

as shown in Figure 11-5. This supports linking the execution of the activity with items that are flowing through the activity or assigned to variables or parameters, and happen to be contained by an object node or assigned to a variable or parameter at the time the link exists. Properties with AdjunctProperty applied, where the principal of the AdjunctProperty is an object node, variable, or parameter, can be used as the end of the associations toward the object node, variable, or parameter type. Like any association end or property these can be the subject of parametric constraints, design values, units, and quantity kinds. The associations may be composition if the intention is to delete instances of the classifier flowing the activity when the activity is terminated. See example in 11.4, Usage Examples.

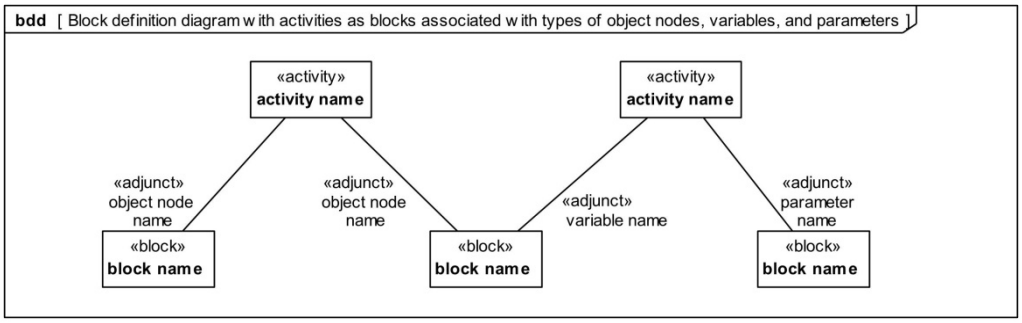


Figure 11-5: Block definition diagram with activities as blocks associated with types of object nodes, variables, and parameter

Object nodes in activity diagrams can optionally show the node name with the name of the type of the object node as shown in Figure 11-6.

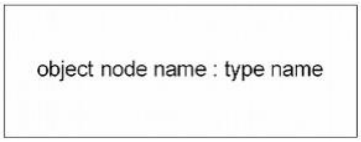


Figure 11-6: ObjectNode notation in activity diagrams

Stereotypes applying to parameters can appear on object nodes in activity diagrams, as shown in Figure 11-7, when the object node notation is used as a shorthand for pins. The stereotype applies to all parameters corresponding to the pins notated by the object node. Stereotype applying to object nodes can also appear in object nodes, and applies to all the pins notated by the object node.

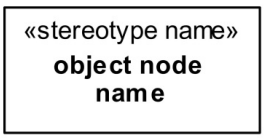


Figure 11-7: ObjectNode notation in activity diagrams

11.3.2 Stereotypes

The following abstract syntax defines the stereotypes in this clause and which metaclasses they extend. The descriptions, attributes, and constraints for each stereotype are specified below.

Package Activities

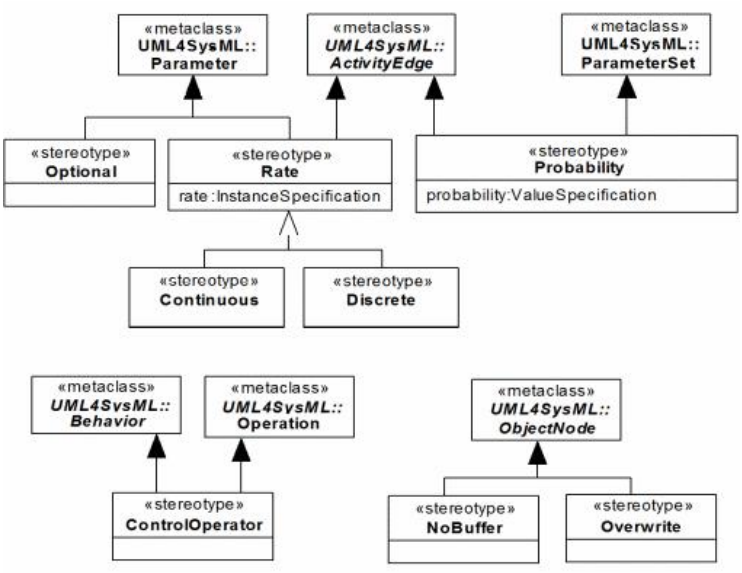


Figure 11-8: Abstract Syntax for SysML Activity Extensions

11.3.2.1 Continuous

Description

Continuous rate is a special case of rate of flow (see Rate) where the increment of time between items approaches zero. It is intended to represent continuous flows that may correspond to water flowing through a pipe, a time continuous signal, or continuous energy flow. It is independent from UML streaming, see clause 11.3.2.8. A streaming parameter may or may not apply to continuous flow, and a continuous flow may or may not apply to streaming parameters.

UML places no restriction on the rate at which tokens flow. In particular, the time between tokens can approach as close to zero as needed, for example to simulate continuous flow. There is also no restriction in UML on the kind of values that flow through an activity. In particular, the value may represent as small a number as needed, for example to simulate continuous material or energy flow. Finally, the exact timing of token flow is not completely prescribed in UML. In particular, token flow on different edges may be coordinated to occur in a clocked fashion, as in time march algorithms for numerical solvers of ordinary differential equations, such as Runge-Kutta.

Generalizations

- Rate (from Activities)

11.3.2.2 ControlOperator

Description

A control operator is a behavior that is intended to represent an arbitrarily complex logical operator that can be used to enable and disable other actions. When the «controlOperator» stereotype is applied to behaviors, the behavior takes control values as inputs or provides them as outputs, that is, it treats control as data (see 11.3.3.1.1). When the «controlOperator» stereotype is not applied, the behavior may not have a parameter typed by ControlValue. The «controlOperator» stereotype also applies to operations with the same semantics.

The control value inputs do not enable or disable the control operator execution based on their value, they only enable based on their presence as data. Pins for control parameters are regular pins, not UML control pins. This is so the control value can be passed into or out of the action and the invoked behavior, rather than control the starting of the action, or indicating the ending of it.

Association Ends

- base_Behavior : Behavior [0..1]
- base_Operation : Operation [0..1]

Constraints

- 1_one_parameter_controlvalue
  When the «controlOperator» stereotype is applied, the behavior or operation shall have at least one parameter typed by ControlValue. If the stereotype is not applied, the behavior or operation may not have any parameter typed by ControlValue.

```
UML::Behavior.allInstances()->forAll(b | not
(ControlOperator.allInstances().base_Behavior->includes(b) xor b.ownedParameter
->exists(p | p.type=SysML::Libraries::ControlValues::ControlValue))) and
UML::Operation.allInstances()->forAll(o | not
(ControlOperator.allInstances().base_Operation->includes(o) xor o.ownedParameter
->exists(p | p.type=SysML::Libraries::ControlValues::ControlValue)))
```

- 2_controloperator_operation_method
  A behavior shall have the «controlOperator» stereotype applied if it is a method of an operation that has the «controlOperator» stereotype applied.

```
(self.base_Operation->notEmpty() and self.base_Operation.method->notEmpty()) implies
self.base_Operation.method->forAll(b | ControlOperator.allInstances().base_Behavior
->includes(b))
```

11.3.2.3 Discrete

Description

Discrete rate is a special case of rate of flow (see clause 11.3.2.8) where the increment of time between items is a non-zero. Examples include the production of assemblies in a factory and signals set at periodic time intervals.

Generalizations

- Rate (from Activities)

Constraints

- 1_not_continuous
  The «discrete» and «continuous» stereotypes shall not be applied to the same element at the same time.

```
(self.base_ActivityEdge->notEmpty() implies
Continuous.allInstances().base_ActivityEdge->excludes(self.base_ActivityEdge)) and
(self.base_Parameter->notEmpty() implies Continuous.allInstances().base_Parameter
->excludes(self.base_Parameter))
```

11.3.2.4 NoBuffer

Description

When the «nobuffer» stereotype is applied to object nodes, tokens arriving at the node are discarded if they are refused by outgoing edges, or refused by actions for object nodes that are input pins. This is typically used with fast or continuously flowing data values, to prevent buffer overrun, or to model transient values, such as electrical signals. For object nodes that are the target of continuous flows, «nobuffer» and «overwrite» have the same effect. The stereotype does not override UML token offering semantics; it just indicates what happens to the token when it is accepted. When the stereotype is not applied, the semantics are as in UML, specifically, tokens arriving at an object node that are refused by outgoing edges, or action for input pins, are held until they can leave the object node.

Association Ends

- base_ObjectNode : ObjectNode [1]

Constraints

- 1_not_overwrite
  The «nobuffer» and «overwrite» stereotypes cannot be applied to the same element at the same time.

```
Overwrite.allInstances().base_ObjectNode->excludes(self.base_ObjectNode)
```

11.3.2.5 Overwrite

Description

When the «overwrite» stereotype is applied to object nodes, a token arriving at a full object node removes one that is already there before being added (a full object node has as many tokens as allowed by its upper bound). This is typically used on an input pin with an upper bound of 1 to ensure that stale data is overridden at an input pin. For upper bounds greater than one, the token removed is the one that has been in the object node the longest. For FIFO ordering, this is the token that is next to be selected, for LIFO it is the token that would be last to be selected. Tokens arriving at a full object node with the Overwrite stereotype applied take up their positions in the ordering as normal, if any. The arriving tokens do not take the positions of the removed tokens. A null token removes all the tokens already there. The number of tokens replaced is equal to the weight of the incoming edge, which defaults to 1. For object nodes that are the target of continuous flows, «overwrite» and «nobuffer» have the same effect. The stereotype does not override UML token offering semantics, just indicates what happens to the token when it is accepted. When the stereotype is not applied, the semantics is as in UML, specifically, tokens arriving at object nodes do not replace ones that are already there.

Association Ends

- base_ObjectNode : ObjectNode [1]

Constraints

- `1_not_nobuffer`
  The «overwrite» and «nobuffer» stereotypes cannot be applied to the same element at the same time.

```
NoBuffer.allInstances().base_ObjectNode->excludes(self.base_ObjectNode)
```

11.3.2.6 Optional

Description

When the «optional» stereotype is applied to parameters, the lower multiplicity shall be equal to zero. This means the parameter is not required to have a value for the activity or any behavior to begin or end execution. Otherwise, the lower multiplicity shall be greater than zero, which is called "required." The absence of this stereotype indicates a constraint, see below.

Association Ends

- base_Parameter : Parameter [1]

Constraints

- `1_lower_is_0`
  A parameter with the «optional» stereotypes applied shall have multiplicity.lower equal to zero, otherwise multiplicity.lower shall be greater than zero

```
UML::Parameter.allInstances()->forAll(p | Optional.allInstances().base_Parameter
->includes(p) xor p.lower > 0)
```

11.3.2.7 Probability

Description

When the «probability» stereotype is applied to edges coming out of decision nodes and object nodes, it provides an expression for the probability that the edge will be traversed. These shall be between zero and one inclusive, and add up to one for edges with same source at the time the probabilities are used.

When the «probability» stereotype is applied to output parameter sets, it gives the probability the parameter set will be given values at runtime. These shall be between zero and one inclusive, and add up to one for output parameter sets of the same behavior at the time the probabilities are used.

Attributes

- probability : ValueSpecification [1]
  Value of the probability

Association Ends

- base_ActivityEdge : ActivityEdge [0..1]
- base_ParameterSet : ParameterSet [0..1]

Constraints

- 1_source_decisionnode_or_objectnode
  The «probability» stereotype shall only be applied to activity edges that have decision nodes or object nodes as sources, or to output parameter sets.

```
(self.base_ActivityEdge->notEmpty() implies
self.base_ActivityEdge.source.oclIsKindOf(UML::DecisionNode)) and
(self.base_ParameterSet->notEmpty() implies self.base_ParameterSet.parameter
->forAll(p | p.direction=UML::ParameterDirectionKind::out))
```

- 2_all_outgoing_edges
  When the «probability» stereotype is applied to an activity edge, then it shall be applied to all edges coming out of the same source.

```
self.base_ActivityEdge->notEmpty() implies
Probability.allInstances().base_ActivityEdge
->includesAll(self.base_ActivityEdge.target.incoming)
```

- 3_all_parametersets
  When the «probability» stereotype is applied to an output parameter set, it shall be applied to all the parameter sets of the behavior or operation owning the original parameter set.

```
self.base_ParameterSet->notEmpty() implies
Probability.allInstances().base_ParameterSet
->includesAll(self.base_ParameterSet.namespace.ownedMember->select(m |
m.oclIsKindOf(UML::ParameterSet)))
```

- 4_all_outputparameter_in_parametersets
  When the «probability» stereotype is applied to an output parameter set, all the output parameters shall be in some parameter set.

```
(self.base_ActivityEdge->notEmpty() implies
Continuous.allInstances().base_ActivityEdge->excludes(self.base_ActivityEdge)) and
(self.base_Parameter->notEmpty() implies Continuous.allInstances().base_Parameter
->excludes(self.base_Parameter))
```

11.3.2.8 Rate

Description

When the «rate» stereotype is applied to an activity edge, it specifies the expected value of the number of objects and values that traverse the edge per time interval, that is, the expected value rate at which they leave the source node and arrive at the target node. It does not refer to the rate at which a value changes over time. When the stereotype is applied to a parameter, the parameter shall be streaming, and the stereotype gives the number of objects or values that flow in or out of the parameter per time interval while the behavior or operation is executing. Streaming is a characteristic of UML behavior parameters that supports the input and output of items while a behavior is executing, rather than only when the behavior starts and stops. The flow may be continuous or discrete, see the specialized rates in clause 11.3.2.1 and clause 11.3.2.3. The «rate» stereotype has a rate property of type InstanceSpecification. The values of this property shall be instances of classifiers stereotyped by «valueType» or «distributionDefinition», see Clause 8. In particular, the denominator for units used in the rate property shall be time units.

Attributes

- rate : InstanceSpecification [1]
  Value of the rate

Association Ends

- base_ActivityEdge : ActivityEdge [0..1]
- base_ObjectNode : ObjectNode [0..1]
- base_Parameter : Parameter [0..1]

Constraints

- 1_streaming
  When the «rate» stereotype is applied to a parameter, the parameter shall be streaming.

```
self.base_Parameter->notEmpty() implies self.base_Parameter.isStream
```

- 2_edges_rates
  The rate of a parameter shall be less than or equal to rates on edges that come into or go out from pins and parameters nodes corresponding to the parameter.

```
self.base_Parameter->notEmpty() implies (
let nodes: Set(UML::ObjectNode) =
if self.base_Parameter.owner.oclIsKindOf(UML::Behavior) then
let pOwner: UML::Behavior = self.base_Parameter.owner.oclAsType(UML::Behavior) in
UML::CallBehaviorAction.allInstances()->select(a | a.behavior = pOwner)
```

```
->collect(a | a.argument->at(pOwner.ownedParameter->indexOf(self.base_Parameter)))
->union(UML::StartObjectBehaviorAction.allInstances()->select(a | a.behavior() =
pOwner)
->collect(a | a.argument->at(pOwner.ownedParameter->indexOf(self.base_Parameter))))
->union(UML::ActivityParameterNode.allInstances()->select(n | n.parameter =
self.base_Parameter))->asSet()
else if self.base_Parameter.owner.oclIsKindOf(UML::Operation) then
let pOwner: UML::Operation = self.base_Parameter.owner.oclAsType(UML::Operation) in
UML::CallOperationAction.allInstances()->select(a | a.operation = pOwner)
->collect(a | a.argument->at(pOwner.ownedParameter->indexOf(self.base_Parameter)))
->asSet()
else
Set(UML::ObjectNode){}
endif endif in
nodes.incoming->flatten()->union(nodes.outgoing->flatten())
->forAll(e | let eRate: Rate = Rate.allInstances()->any(r | r.base_ActivityEdge=e) in
(not eRate.oclIsUndefined() and self.rate.specification.realValue() <=
eRate.rate.specification.realValue())) )
```

11.3.3 Model Libraries

11.3.3.1 Package ControlValues

The SysML model library for activities is shown in Figure 11-9.

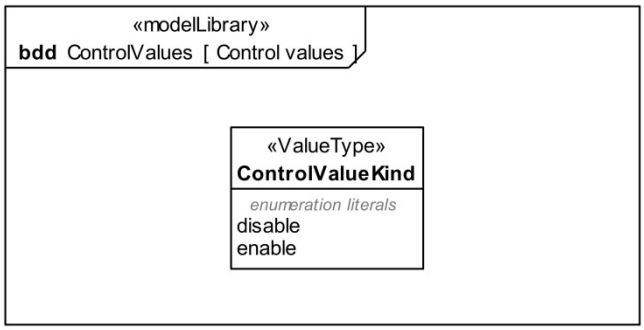


Figure 11-9: Control values

11.3.3.1.1 ControlValueKind

Description

The ControlValueKind enumeration is a type for treating control values as data (see 11.3.2.2) and for UML control pins. It can be used as the type of behavior and operation parameters, object nodes, and attributes, and so on. The possible runtime values are given as enumeration literals. Modelers can extend the enumeration with additional literals, such as suspend, resume, with their own semantics.

The disable literal means a termination of an executing behavior that can only be started again from the beginning (compare to suspend). The enable literal means to start a new execution of a behavior (compare to resume).

Literals

- disable
  The disable literal means a termination of an executing behavior that can only be started again from the beginning (compare to suspend).
- enable
  The enable literal means to start a new execution of a behavior (compare to resume).

Constraints

- `1_node_is_controltype`
  UML::ObjectNode::isControlType is true for object nodes with type ControlValue

11.4 Usage Examples

The following examples illustrate modeling continuous systems (see 11.1.2, Continuous Systems). Figure 11-10 shows a simplified model of driving and braking in a car that has an automatic braking system. Turning the key on has a duration constraint specifying that this action lasts no more than 0.1 seconds. Turning the key on starts two behaviors, Driving and Braking. These behaviors execute until the key is turned off, using streaming parameters to communicate with other behaviors. The Driving behavior outputs a brake pressure continuously to the Braking behavior while both are executing, as indicated by the «continuous» rate and streaming properties (streaming is a characteristic of UML behavior parameters that supports the input and output of items while a behavior is executing, rather than only when the behavior starts and stops). Brake pressure information also flows to a control operator that outputs a control value to enable or disable the Monitor Traction behavior. No pins are used on Monitor Traction, so once it is enabled, the continuously arriving enable control values from the control operator have no effect, per UML semantics. When the brake pressure goes to zero, disable control values are emitted from the control operator. The first one disables the monitor, and the rest have no effect. While the monitor is enabled, it outputs a modulation frequency for applying the brakes as determined by the ABS system. The rake notations on the control operator and Monitor Traction indicate they are further defined by activities, as shown in Figure 11-11 and Figure 11-12. An alternative notation for this activity decomposition is shown in Figure 11-13.

The duration constraint notation associated with the Turn Key To On action is supported by the UML Simple Time model. The Operate Car activity owns a duration constraint specifying that the “Turn Key To On” action lasts no more than 0.1 seconds. The concrete UML element used in this example is a DurationConstraint owned by Operate Car that constrains the Turn Key To On action. The DurationConstraint owns a DurationInterval, which specifies that the action is constrained to last between 0 seconds and 0.1 seconds (both being Duration expressions).

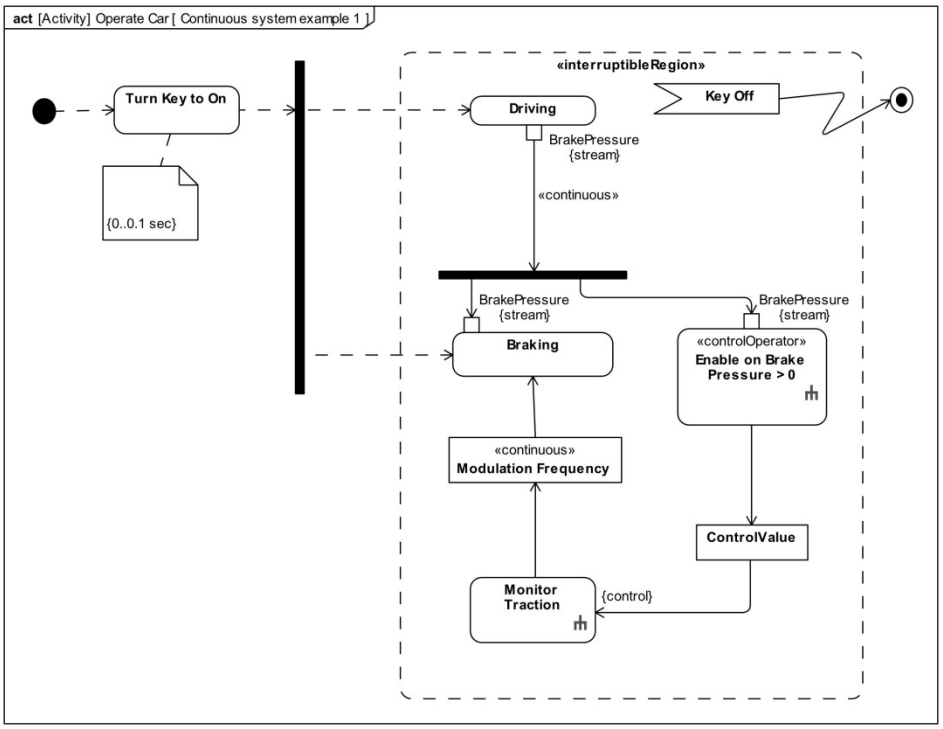


Figure 11-10: Continuous system example 1

The activity diagram for Monitor Traction is shown in Figure 11-11. When Monitor Traction is enabled, it begins listening for signals coming in from the wheel and accelerometer, as indicated by the signal receipt symbols on the left, which begin listening automatically when the activity is enabled. A traction index is calculated every 10 ms, which is the slower of the two signal rates. The accelerometer signals come in continuously, which means the input to Calculate Traction does not buffer values. The result of Calculate Traction is filtered by a decision node for a threshold value and Calculate Modulation Frequency determines the output of the activity.

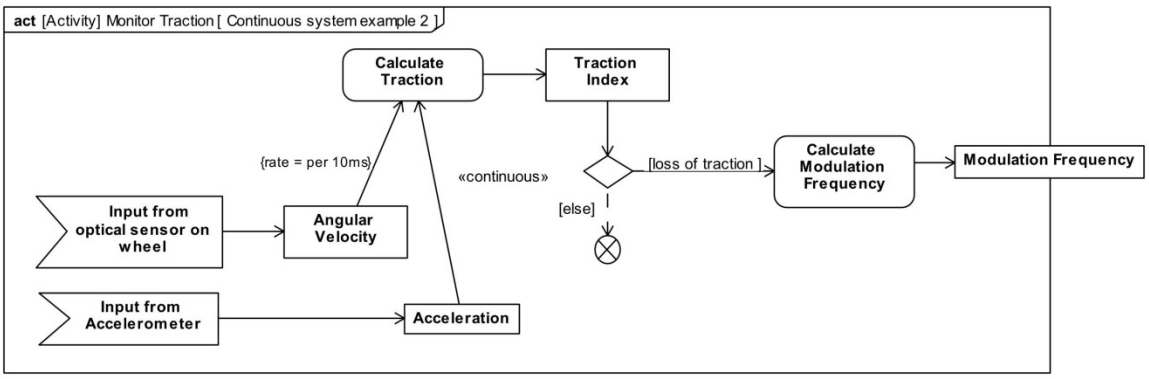


Figure 11-11: Continuous system example 2

The activity diagram for the control operator Enable on Brake Pressure > 0 is shown in Figure 11-12. The decision node and guards determine if the brake pressure is greater than zero, and flow is directed to value specification actions that

output an enabling or disabling control value from the activity. The edges coming out of the decision node indicate the probability of each branch being taken.

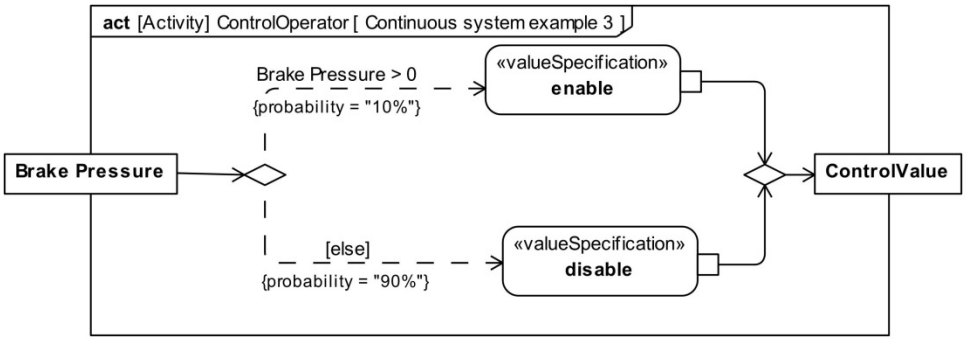


Figure 11-12: Continuous system example 3

Figure 11-13 shows a block definition diagram with composition associations between the activities and AdjunctProperty applied to the part ends in Figures 11.10, 11.11, and 11.12, as an alternative way to show the activity decomposition of Figures 11.10, 11.11, and 11.12. Each instance of Operating Car is an execution of that behavior. It owns the executions of the behaviors it invokes synchronously, such as Driving. Like all composition, if an instance of Operating Car is destroyed, terminating the execution, the executions it owns are also terminated.

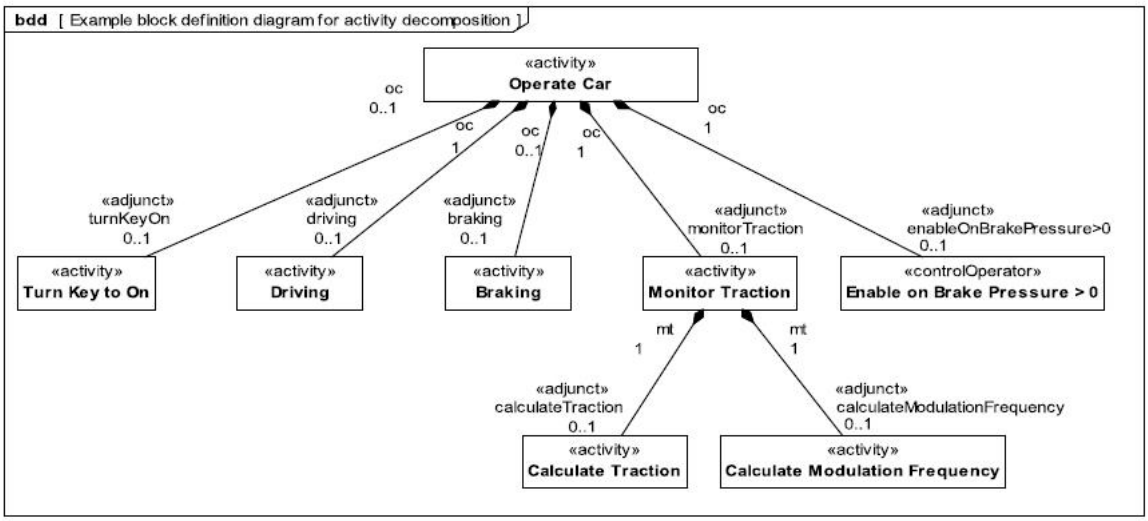


Figure 11-13: Example block definition diagram for activity decomposition

Figure 11-14 shows a block definition diagram with composition associations between the activity in Figure 11-10 and the types the object nodes in that activity, with AdjunctProperty applied to the object node type end. In an instance of Operating Car, which is one execution of it, instances of Brake Pressure and Modulation Frequency are linked to the execution instance when they are in the object nodes of the activity.

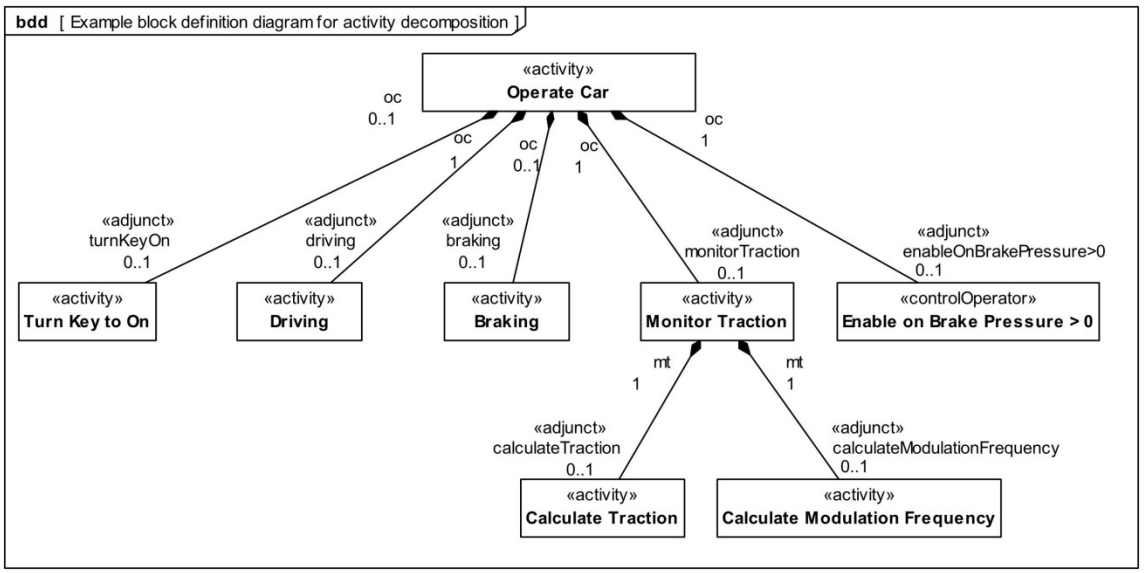


Figure 11-14: Example block definition diagram for object node types

This page intentionally left blank.

12 Interactions

12.1 Overview

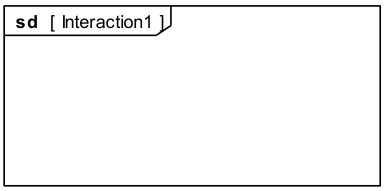
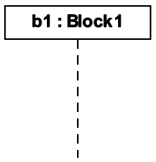
Interactions are used to describe interactions between entities. UML Interactions are supported by four diagram types including the Sequence diagram, Communications diagram, Interaction Overview diagram, and Timing diagram. The Sequence diagram is the most common of the Interaction diagrams. SysML includes the Sequence diagram only and excludes the Interaction Overview diagram and Communication diagram, which were considered to offer significantly overlapping functionality without adding significant capability for system modeling applications. The Timing diagram is also excluded due to concerns about its maturity and suitability for systems engineering needs.

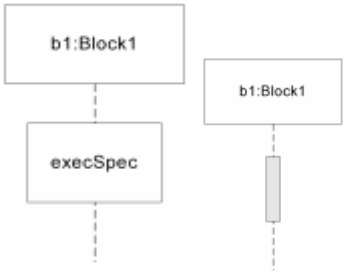
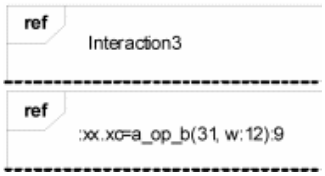
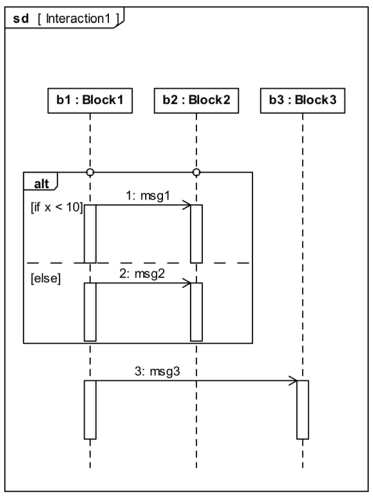
The Sequence diagram describes the flow of control between actors and systems (blocks) or between parts of a system. This diagram represents the sending and receiving of messages between the interacting entities called lifelines, where time is represented along the vertical axis. The sequence diagrams can represent highly complex interactions with special constructs to represent various types of control logic, reference interactions on other sequence diagrams, and decomposition of lifelines into their constituent parts.

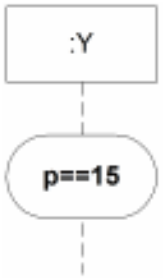
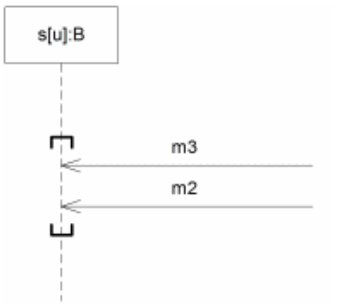
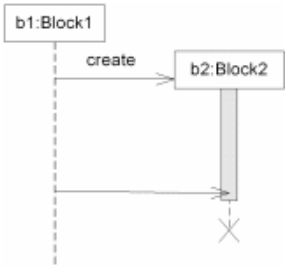
12.2 Diagram Elements

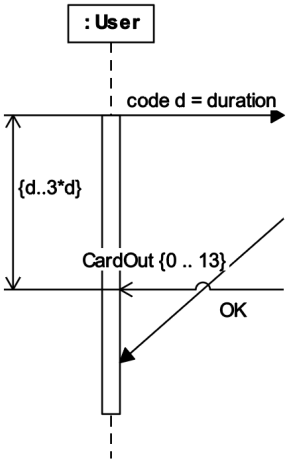
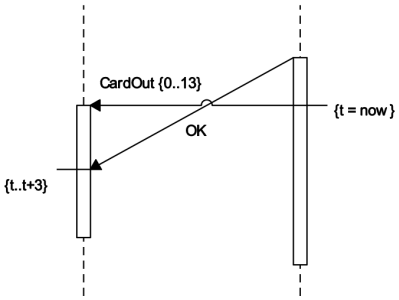
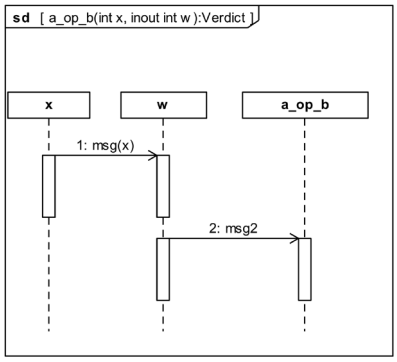
12.2.1 Sequence Diagram

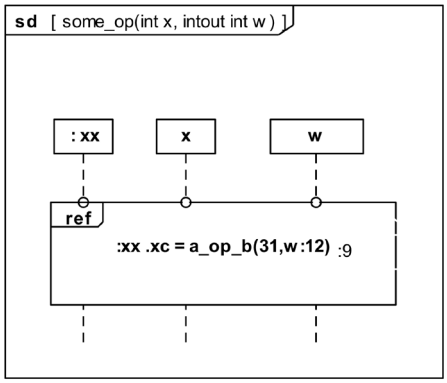
Table 12-1: Graphical notation of sequence diagrams

Notation Name	Concrete Syntax	Abstract Syntax Reference
SequenceDiagram Frame and Heading		**UML4SysML::Interaction**
Lifeline		**UML4SysML::Lifeline**

Notation Name	Concrete Syntax	Abstract Syntax Reference
Execution Specification		**UML4SysML::ExecutionSpecification**
InteractionUse		**UML4SysML::InteractionUse** An InteractionUse with just the <interaction-name>. An InteractionUse with <attribute - name>, the value of arguments, the <return-value>, etc.
CombinedFragment		**UML4SysML::CombinedFragment** A combined fragment is defined by an interaction operator and corresponding interaction operands. Interaction Operators include: seq - Weak Sequencing alt – Alternatives opt – Option break – Break par – Parallel strict - Strict Sequencing loop – Loop critical - Critical Region neg – Negative assert – Assertion ignore – Ignore consider – Consider

Notation Name	Concrete Syntax	Abstract Syntax Reference
StateInvariant / Continuations		**UML4SysML::Continuation** **UML4SysML::StateInvariant**
Coregion		**UML4SysML::CombineFragment** (under *parallel*)
CreationEvent **DestructionEvent**		**UML4SysML::CreationEvent** **UML4SysML::DestructionEvent**

Notation Name	Concrete Syntax	Abstract Syntax Reference
DurationConstraint Duration Observation		**UML4SysML::Interactions**
TimeConstraint TimeObservation		**UML4SysML::Interactions**
SequenceDiagram (advanced)		**UML4SysML::Interaction** Shows usage of arguments and assignment to return value.

Notation Name	Concrete Syntax	Abstract Syntax Reference
InteractionUse (advanced)		**UML4SysML::InteractionUse** Shows usage of arguments and assignment to attribute value upon return.

a. Table is compliant with UML 2.1 document.

Table 12-2: Graphical paths included in sequence diagram

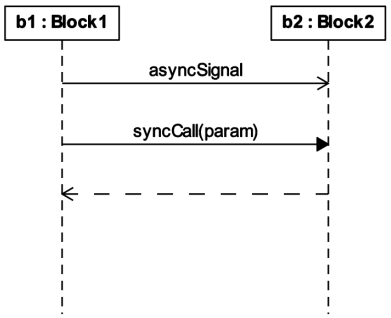
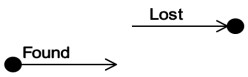
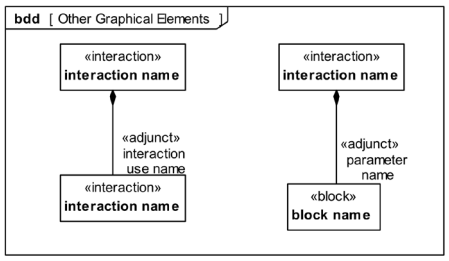
Path Name	Concrete Syntax	Abstract Syntax Reference
Message		**UML4SysML::Message**
Lost Message **Found Message**		**UML4SysML::Message**
GeneralOrdering		**UML4SysML::GeneralOrdering**

Table 12-3: Other graphical elements included in sequence diagram

Element Name	Concrete Syntax	Abstract Syntax Reference
In Block Definition Diagrams, Interaction, Association, AdjunctProperty		**UML4SysML::Interactions, UML4SysML::Association, SysML::Blocks::AdjunctProperty**

12.3 UML Extensions

12.3.1 Diagram Extensions

The following specify diagram extensions to the notations defined in Clause 17, “Profiles & Model Libraries.”

12.3.1.1 Exclusion of Communication Diagram, Interaction Overview Diagram, and Timing Diagram

Communication diagrams and Interaction Overview diagrams are excluded from SysML. The other behavioral diagram representations were considered to provide sufficient coverage without introducing these diagram kinds. Timing diagrams are also excluded due to concerns about their maturity and suitability for systems engineering needs.

12.3.1.2 Interactions and Parameters

12.3.1.2.1 Notation

In UML, all behaviors are classes, including interactions, and their instances are executions of the interaction. Interactions as blocks and associations between interactions corresponding to interaction uses have an analogous semantics to activities as blocks and associations between activities corresponding to call actions, see 11.3.1.1.1, Notation. Similarly, associations between interactions and classifiers (blocks or value types) have an analogous semantics to associations between activities and blocks or value types, see 11.3.1.4.1, Notation.

Interactions in block definition diagrams appear as regular blocks, except the «interaction» keyword may be used to indicate the Block stereotype is applied to an interaction, as shown in Figure 12-1 Properties with AdjunctProperty applied, where the principal of the AdjunctProperty is an interaction use, can be used as the end of the associations towards the interaction being used. Properties with AdjunctProperty applied, where the principal of the AdjunctProperty is a parameter of the interaction, can be used as the end of the associations towards the parameter type. See 8.3.2.2, AdjunctProperty for constraints when AdjunctProperty is used with interaction uses and parameters. Interactions in block definition diagrams can also appear with the same notation as InteractionUses.

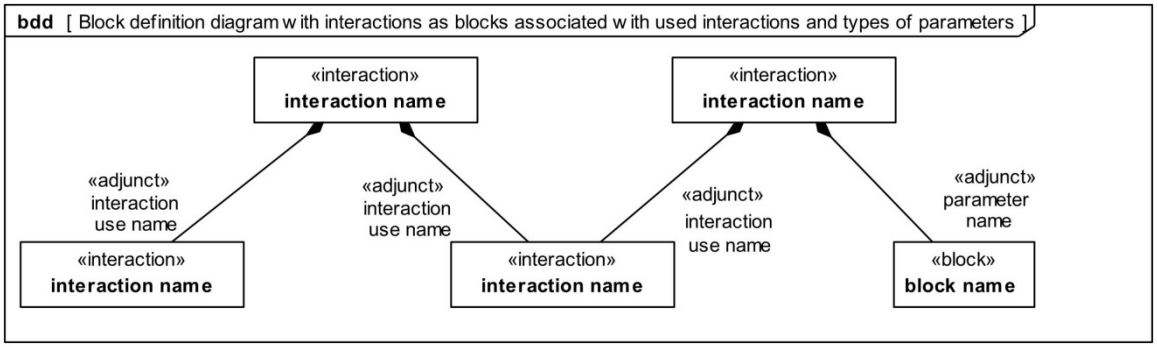


Figure 12-1: Block definition diagram with interactions as blocks associated with used interactions and types of parameters

12.4 Usage Examples

12.4.1 Sequence Diagrams

Figure D.7 illustrates the overall system behavior for operating the vehicle in Sequence diagram format. To manage the complexity, a hierarchical sequence diagram is used which refers to other interactions that further elaborate the system behavior ("ref StartVehicleBlackBox"). CombinedFragments are used to illustrate that steering can take place at the same time as controlling the speed and that controlling speed can be either idling, accelerating/cruising, or braking.

Figure D.9 shows an interaction that includes events and messages communicated between the driver and vehicle during the starting of the vehicle. The "hybridSUV" lifeline represents another interaction which further elaborates what happens inside the "hybridSUV" when the vehicle is started.

Figure D.10 shows the sequence of communication that occurs inside the HybridSUV when the vehicle is started successfully.

This page intentionally left blank.

13 State Machines

13.1 Overview

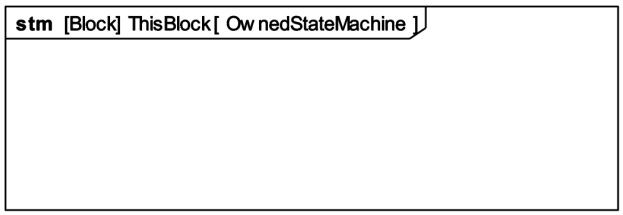
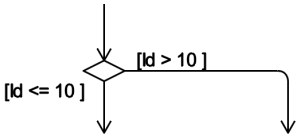
The StateMachine package defines a set of concepts that can be used for modeling discrete behavior through finite state transition systems. The state machine represents behavior as the state history of an object in terms of its transitions and states. The activities that are invoked during the transition, entry, and exit of the states are specified along with the associated event and guard conditions. Activities that are invoked while in the state are specified as “do Activities,” and can be either continuous or discrete. A composite state has nested states that can be sequential or concurrent.

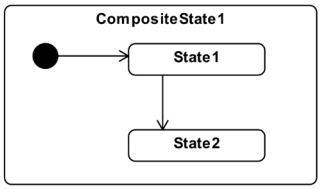
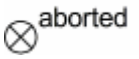
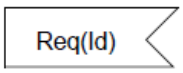
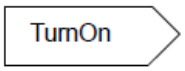
The UML concept of protocol state machines is excluded from SysML to reduce the complexity of the language. The standard UML state machine concept, called behavior state machines in UML, is thought to be sufficient for expressing protocols.

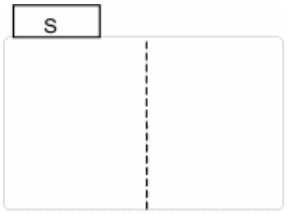
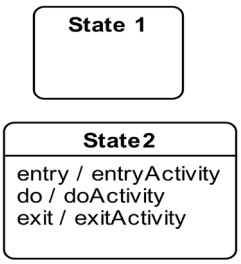
13.2 Diagram Elements

13.2.1 State Machine Diagram

Table 13-1: Graphical notation of state machine diagrams

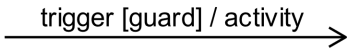
Notation Name	Concrete Syntax	Abstract Syntax Reference
StateMachineDiagram Frame and Heading		**UML4SysML::StateMachine**
Choice pseudo state		**UML4SysML::PseudoState**

Notation Name	Concrete Syntax	Abstract Syntax Reference
Composite state		**UML4SysML::State**
Entry point		**UML4SysML::PseudoState**
Exit point		**UML4SysML::PseudoState**
Final state		**UML4SysML::FinalState**
History, Deep Pseudo state		**UML4SysML::PseudoState**
History, Shallow pseudo state		**UML4SysML::PseudoState**
Initial Pseudo state		**UML4SysML::PseudoState**
Junction Pseudo state		**UML4SysML::PseudoState**
Receive signal action		**UML4SysML::Transition**
Send signal action		**UML4SysML::Transition**

Notation Name	Concrete Syntax	Abstract Syntax Reference
Action		**UML4SysML::Transition**
Region		**UML4SysML::Region**
Simple state		**UML4SysML::State**
State list		**UML4SysML::State**
State Machine		**UML4SysML::StateMachine**
Terminate node		**UML4SysML::PseudoState**

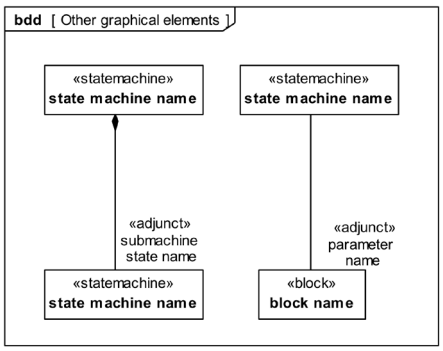
Notation Name	Concrete Syntax	Abstract Syntax Reference
Submachine state		**UML4SysML::State**
Composite State with a hidden decomposition indicator icon		**UML4SysML::State**

Table 13-2: Graphical paths included in state machine diagrams

Path Name	Concrete Syntax	Abstract Syntax Reference
Transition		**UML4SysML::Transition**
Alternative entry point Connection-PointReference notation		**UML4SysML::ConnectionPoint Reference**

Path Name	Concrete Syntax	Abstract Syntax Reference
Alternative exit point ConnectionPointReference notation		**UML4SysML::ConnectionPointReference**

Table 13.3: Other graphical elements included in state machine diagram

Element Name	Concrete Syntax	Abstract Syntax Reference
In Block Definition Diagrams, Interaction, Association, AdjunctProperty		**UML4SysML::StateMachines, UML4SysML::Association, SysML::Blocks::AdjunctProperty**

13.3 UML Extensions

13.3.1 Diagram Extensions

13.3.1.1 State Machines and Parameters

13.3.1.1.1 Notation

In UML, all behaviors are classes, including state machines, and their instances are executions of the state machine. State machines as blocks and associations between state machines corresponding to submachine states have an analogous semantics to activities as blocks and associations between activities corresponding to call actions, see 11.3.1.1.1, Notation. Similarly, associations between state machines and classifiers (blocks or value types) have an analogous semantics to associations between activities and blocks or value types, see 11.3.1.4.1, Notation.

State machines in block definition diagrams appear as regular blocks, except the «stateMachine» keyword may be used to indicate the Block stereotype is applied to a state machine, as shown in Figure 13-1. Properties with AdjunctProperty applied, where the principal of the AdjunctProperty is a submachine state, can be used as the end of the associations towards the sub state machine. Properties with AdjunctProperty applied, where the principal of the AdjunctProperty is a parameter of the state machine, can be used as the end of the associations towards the parameter type. See 8.3.2.2, AdjunctProperty for constraints when AdjunctProperty is used with submachine states and parameters. State machines in block definition diagrams can also appear with the same notation as submachine states.

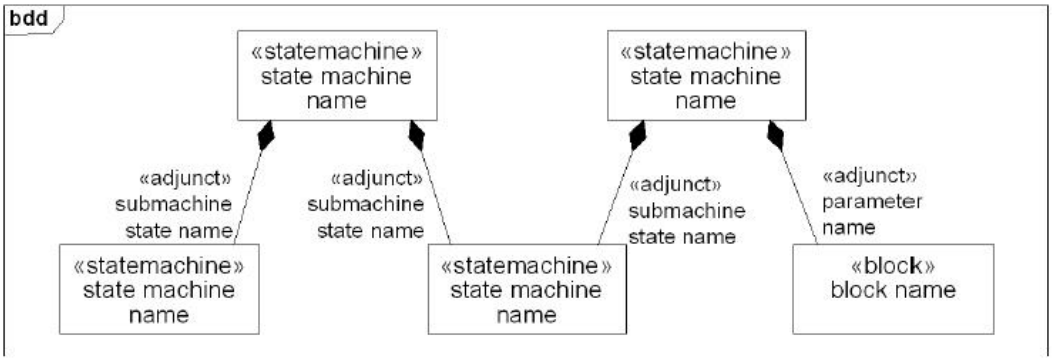


Figure 13-1: Block definition diagram with state machines as blocks associated with submachines and types of parameters

13.4 Usage Examples

13.4.1 State Machine Diagram

The high level states or modes of the HybridSUV including the events that trigger changes of state are illustrated in the state machine diagram in Figure D.8.

14 Use Cases

14.1 Overview

The use case diagram describes the usage of a system (subject) by its actors (environment) to achieve a goal, that is realized by the subject providing a set of services to selected actors. The use case can also be viewed as functionality and/or capabilities that are accomplished through the interaction between the subject and its actors. Use case diagrams include the use case and actors and the associated communications between them. Actors represent classifier roles that are external to the system that may correspond to users, systems, and or other environmental entities. They may interact either directly or indirectly with the system. The actors are often specialized to represent a taxonomy of user types or external systems.

The use case diagram is a method for describing the usages of the system. The association between the actors and the use case represent the communications that occur between the actors and the subject to accomplish the functionality associated with the use case. The subject of the use case can be represented via a system boundary. The use cases that are enclosed in the system boundary represent functionality that is realized by behaviors such as activity diagrams, sequence diagrams, and state machine diagrams.

The use case relationships are “communication,” “include,” “extend,” and “generalization.” Actors are connected to use cases via communication paths, which are represented by an association relationship. The “include” relationship provides a mechanism for factoring out common functionality that is shared among multiple use cases, and is required for the goals of the actor of the base use case to be met. The “extend” relationship provides optional functionality (optional in the sense of not being required to meet the goals), which extends the base use case at defined extension points under specified conditions. The “generalization” relationship provides a mechanism to specify variants of the base use case.

The use cases are often organized into packages with the corresponding dependencies between the use cases in the packages.

14.2 Diagram Elements

14.2.1 Use Case Diagram

Table 14-1: Graphical nodes included in Use Case diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Use Case		**UML4SysML::UseCase**

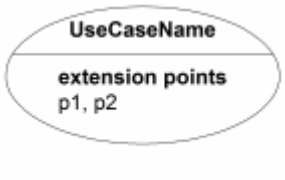
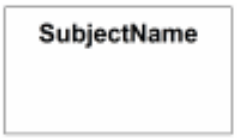
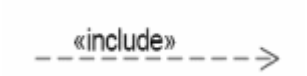
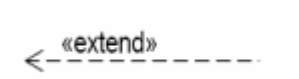
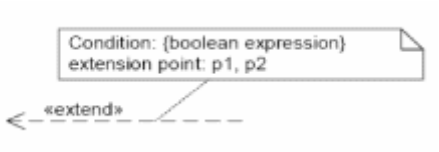
Node Name	Concrete Syntax	Abstract Syntax Reference
Use Case with Extension Points		**UML4SysML::UseCase**
Actor		**UML4SysML::Actor**
Subject		**Association end name on** UML4SysML::Classifier

Table 14-2: Graphical paths included in Use Case diagrams

Path Name	Concrete Syntax	Abstract Syntax Reference
Communication path		**UML4SysML::Association**
Include		**UML4SysML::include**
Extend		**UML4SysML::Extend**

Path Name	Concrete Syntax	Abstract Syntax Reference
Extend with Condition		**UML4SysML::Extend**
Generalization		**UML4SysML::Kernel**

14.3 UML Extensions

None.

14.4 Usage Examples

Figure D.5 is a top-level set of use cases for the Hybrid SUV System. Figure D.6 shows the decomposition of the Operate the Vehicle use case. In this diagram, the frame represents the package that contains the lower level use cases. The convention of naming the package with the same name as the top level use case has been employed. This practice offers an implicit tracing mechanism that complements the explicit trace relationships in SysML.

In Figure D.6 the Extend relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case. Note, however, that the extended use case is defined independently of the extending use case and is meaningful independently of the extending use case. On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions. The “Start the Vehicle” use case is modeled as an extension of “Drive the Vehicle.” This means that there are conditions that may exist that require the execution of an instance of “Start the Vehicle” before an instance of “Drive the Vehicle” is executed.

The use cases “Accelerate,” “Steer,” and “Brake” are modeled using the include relationship. Include is a DirectedRelationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case. This means that “Accelerate,” “Steer,” and “Brake” are all part of the normal process of executing an instance of “Drive the Car.”

In many situations, the use of the Include and Extend relationships is subjective and may be reversed, based on the approach of an individual modeler.

This page intentionally left blank.

CROSSCUTTING CONSTRUCTS

This page intentionally left blank.

15 Allocations

15.1 Overview

Allocation is the term used by systems engineers to denote the organized cross-association (mapping) of elements within the various structures or hierarchies of a user model. The concept of “allocation” requires flexibility suitable for abstract system specification, rather than a particular constrained method of system or software design. System modelers often associate various elements in a user model in abstract, preliminary, and sometimes tentative ways. Allocations can be used early in the design as a precursor to more detailed rigorous specifications and implementations. The allocation relationship can provide an effective means for navigating the model by establishing cross relationships, and ensuring the various parts of the model are properly integrated.

This clause does not try to limit the use of the term “allocation,” but provides a basic capability to support allocation in the broadest sense. It does include some specific subclasses of allocation for allocating behavior, structure, and flows. A typical example is the allocation of activities to blocks (e.g., functions to components). This clause specifies an extension for an allocation relationship and selected subclasses of allocation, along with the notation to represent allocations in a SysML model.

15.2 Diagram Elements

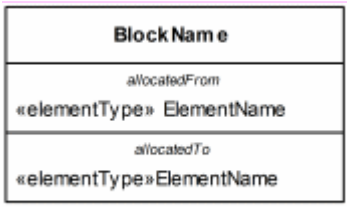
The diagram elements defined in this clause may be shown on some or all SysML diagram types, in addition to the diagram elements that are specific for each diagram type.

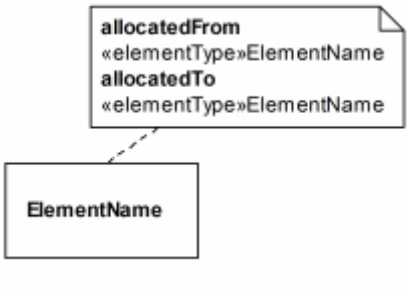
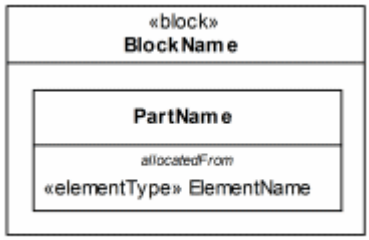
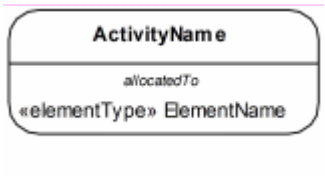
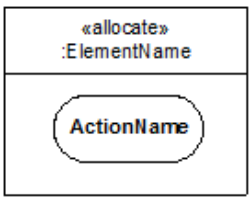
In the following table, «elementType» is a placeholder for a keyword used to specify the kind of element it prefixes. For uniformity, the «elementType» displayed for the allocated-to or allocated-from elements should be from the following list, as applicable: «activity», «action», «objectFlow», «controlFlow», «objectNode», «operation», «block», «property», «itemFlow», «connector», «port», «value».

Other «elementType» designations may be used, if none of the above apply. Note that it is important to use fully qualified names to avoid ambiguity when required. An example of a fully qualified name is the form: (PackageName::ElementName.PropertyName).

15.2.1 Representing Allocation on Diagrams

Table 15-1: Extension to graphical nodes included in diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Allocation derived properties displayed in compartment of a Block.		**SysML::Allocation:Allocate**

Node Name	Concrete Syntax	Abstract Syntax Reference
Allocation derived properties displayed in Comment.		**SysML::Allocation:Allocate**
Allocation derived properties displayed in compartment of Part on Internal Block Diagram.		**SysML::Allocation:Allocate**
Allocation derived properties displayed in compartment of Action on Activity Diagram.		**SysML::Allocation:Allocate**
Allocation Activity Partition		**SysML::Allocation:Allocate ActivityPartition**

Node Name	Concrete Syntax	Abstract Syntax Reference
Allocation (general)		**SysML::Allocation:Allocate**

15.3 UML Extensions

15.3.1 Diagram Extensions

15.3.1.1 Tables

Allocation relationships may be depicted in tables. A separate row is provided for each «allocate» dependency. “from” is the client of the «allocate» dependency, and “to” is the supplier. Both ElementType and ElementName for client and supplier appear in this table.

15.3.1.2 Allocate Relationship Rendering

The “allocate” relationship is a dashed line with an open arrow head. The arrow points in the direction of the allocation. In other words, the directed line points “from: the element being allocated “to” the element that is the target of the allocation.

15.3.1.3 Allocation Compartment Format

When the allocations of a model element are displayed in a compartment, a shorthand notation is used as shown in Table 15-1. This shorthand groups and lists the elements allocated to that element together (in the “allocated from” compartment), then the elements allocated from that element (in the “allocated to” compartment), per the result of Allocate::getAllocatedFrom() and getAllocatedTo() respectively, called with that element as parameter.

15.3.1.4 Allocation Callout Format

When the allocation compartment is not used, a callout notation may be used. An allocation callout notation uses the same shorthand notation as the allocation compartment. This notation is also shown in Table 15-1. For brevity, the «elementType» portion of allocated-from or allocated-to elements may be elided from the diagram.

15.3.1.5 AllocatedActivityPartition Label

For brevity, the keyword used on an AllocatedActivityPartition is «allocate», rather than the stereotype name («allocateActivityPartition»). For brevity, the «elementType» portion of the allocatedFrom or allocatedTo property may be elided from the diagram.

15.3.2 Stereotypes

Package Allocations

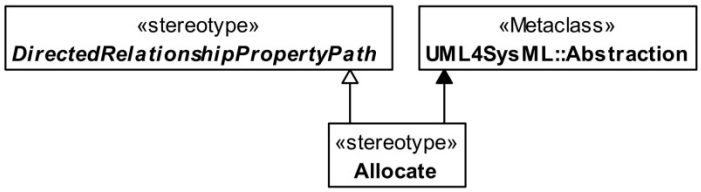


Figure 15-1: Abstract syntax extensions for SysML Allocation

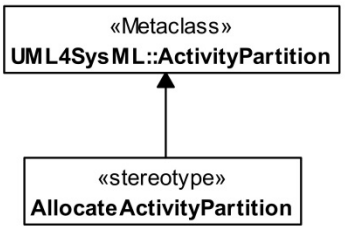


Figure 15-2: Abstract syntax expression for AllocatedActivityPartition

15.3.2.1 Allocate

Description

Allocate is a dependency based on UML::Abstraction. It is a mechanism for associating elements of different types, or in different hierarchies, at an abstract level. Allocate is used for assessing user model consistency and directing future design activity. It is expected that an «allocate» relationship between model elements is a precursor to a more concrete relationship between the elements, their properties, operations, attributes, or sub-classes.

Allocate is a stereotype of a UML4SysML::Abstraction that is permissible between any two NamedElements. It is depicted as a dependency with the "allocate" keyword attached to it. Allocate is directional in that one NamedElement is the "from" end (no arrow), and one NamedElement is the "to" end (the end with the arrow). The Allocate stereotype specializes DirectedRelationshipPropertyPath to enable allocations to identify their sources and targets by a multi-level path of accessible properties from context blocks for the sources and targets.

The following paragraphs describe types of allocation that are typical in systems engineering.

Behavior allocation relates to the systems engineering concept segregating form from function. This concept requires independent models of "function" (behavior) and "form" (structure), and a separate, deliberate mapping between elements in each of these models. It is acknowledged that this concept does not support a standard object-oriented paradigm, not is this always even desirable. Experience on large scale, complex systems engineering problems have proven, however, that segregation of form and function is a valuable approach. In addition, behavior allocation may also include the allocation of Behaviors to BehavioralFeatures of Blocks (e.g., Operations).

Flow allocation specifically maps flows in functional system representations to flows in structural system representations.

Flow between activities can either be control or object flow. The figures in the Usage Examples show concrete syntax for how object flow is mapped to connectors on Activity Diagrams. Allocation of control flow is not specifically addressed in SysML, but may be represented by relating an ItemFlow to the Control Flow using the UML relationship InformationalFlow.realizingActivityEdge.

Note that allocation of ObjectFlow to Connector is an Allocation of Usage, and does NOT imply any relation between any defining Blocks of ObjectFlows and any defining associations of connectors.

The figures in the Usage Examples illustrate an available mechanism for relating the objectNode from an activity diagram to the ItemFlow on an internal block diagram. ItemFlow is discussed in 9, "Ports and Flows."

Pin to Port allocation is not addressed in this release of SysML.

Structure allocation is associated with the concept of separate "logical" and "physical" representations of a system. It is often necessary to construct separate depictions of a system and define mappings between them. For example, a complete system hierarchy may be built and maintained at an abstract level. In turn, it shall then be mapped to another complete assembly hierarchy at a more concrete level. The set of models supporting complex systems development may include many of these levels of abstraction. This International Standard will not define "logical" or "physical" in this context, except to acknowledge the stated need to capture allocation relationships between separate system representations.

Generalizations

- DirectedRelationshipPropertyPath (from Blocks)

Association Ends

- base_Abstraction : Abstraction [1]
  (redefines: DirectedRelationshipPropertyPath::base_DirectedRelationship)

Operations

- getAllocatedFrom (in ref : NamedElement) : NamedElement [0..*]

```
bodyCondition:
getAllocatedFrom = Allocate.allInstances()->select(to = ref).from
```

- getAllocatedTo (in ref : NamedElement) : NamedElement [0..*]

```
bodyCondition:
getAllocatedFrom = Allocate.allInstances()->select(from = ref).to
```

Constraints

- 2_binary
  A single «allocate» dependency shall have only one client (from) and one supplier (to).

```
self.base_Abstraction.source->size() = 1 and self.base_Abstraction.target->size() = 1
```

15.3.2.2 AllocateActivityPartition

Description

AllocateActivityPartition is used to depict an «allocate» relationship on an Activity diagram. The AllocateActivityPartition is a standard UML::ActivityPartition, with modified constraints as stated below.

Association Ends

- base_ActivityPartition : ActivityPartition [1]

Constraints

- 1_actions_on_client_ends
  An Action appearing in an "AllocateActivityPartition" shall be the /client (from) end of an "allocate" dependency. The element that represents the "AllocateActivityPartition" shall be the /supplier (to) end of the same "allocate" dependency. In the «AllocateActivityPartition» name field, Properties are designated by the use of a fully qualified name (including colon, e.g., "part_name:Block_Name"), and Classifiers are designated by a simple name (no colons, e.g., "Block_Name").

```
self.base_ActivityPartition.node->select(n|n.oclIsKindOf(UML::Action)) ->forAll(a |
let allocs: Set(UML::Abstraction) = Allocate.allInstances().base_Abstraction
->select(x |x.client->includes(a))->asSet() in allocs->exists(x | x.supplier
->includes(self.base_ActivityPartition.represents)))
```

- 2_not_uml_semantics
  The «AllocateActivityPartition» shall maintain the constraints, but not the semantics, of the UML::ActivityPartition. Classifiers or Properties represented by an «AllocateActivityPartition» do not have any direct responsibility for invoking behavior depicted within the partition boundaries. To depict this kind of direct responsibility, the modeler is directed to the UML 2 standard, sub clause 12.3.10, "ActivityPartition," Semantics topic.

```
-- Cannot be expressed in OCL
```

15.4 Usage Examples

The following examples depict allocation relationships as property callout boxes (basic), property compartment of a Block (basic), and property compartments of Activities and Parts (advanced). Figure 15-3 shows generic allocation for Blocks.

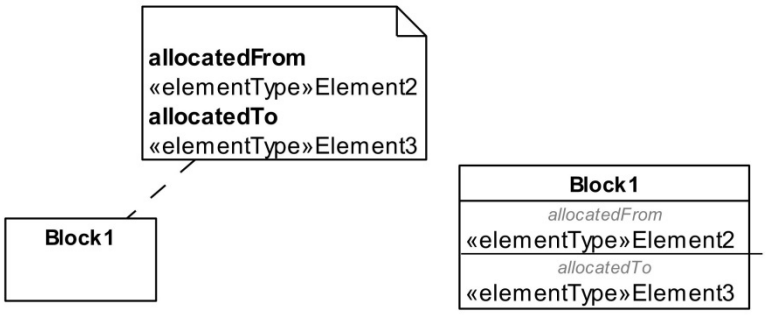


Figure 15-3: Generic Allocation, including /from and /to association ends

15.4.1 Behavior Allocation of Actions to Parts and Activities to Blocks

Specific behavior allocation of Actions to Parts are depicted in Figure 15-4.

Note that the AllocateActivityPartition, if used in this manner, is unambiguously associated with behavior allocation. The allocation to Activity6 comes from a nested part, and uses the attributes of DirectedRelationshipPropertyPath to specify the path of properties to reach that part. The sourceContext of the allocation is Block4 and the sourcePropertyPath is (Part5).

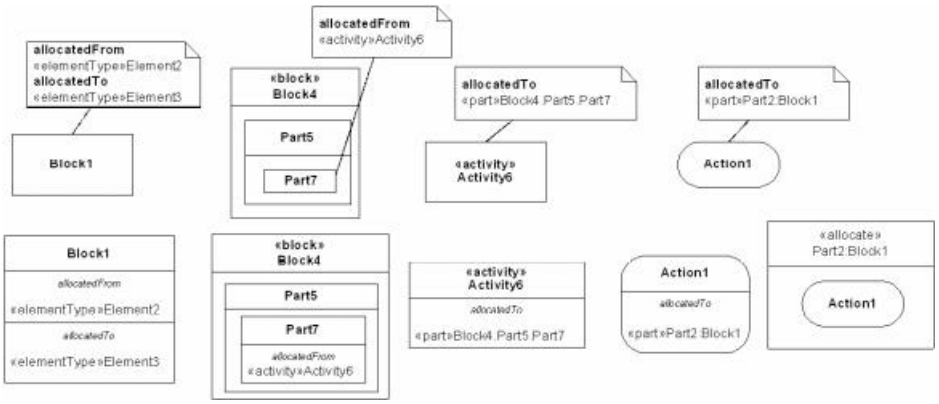


Figure 15-4: Behavior Allocation

15.4.2 Allocate Flow

Figure 15-5 shows flow allocation of ObjectFlow to a Connector, or alternatively to an ItemFlow. Allocation of ControlFlow is not shown as an example, but it is not prohibited in SysML.

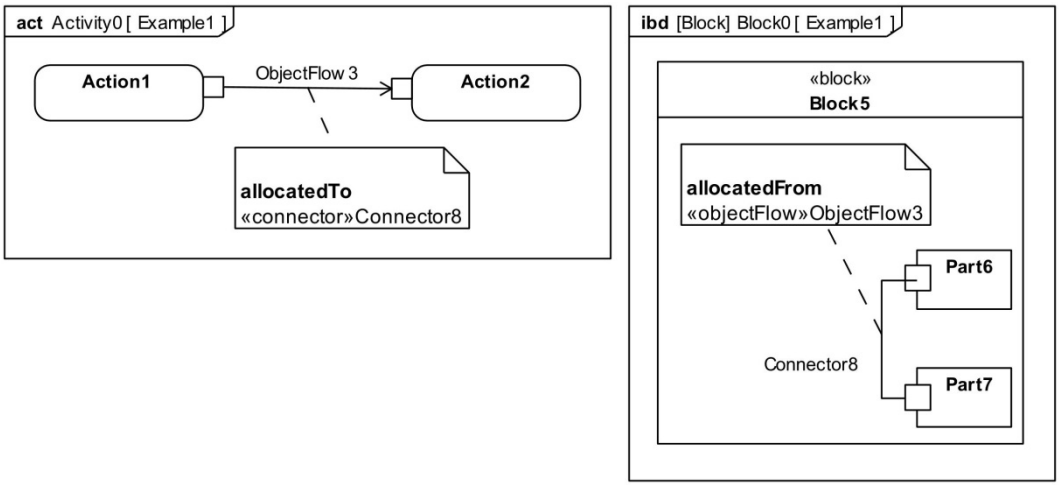


Figure 15-5: Example of flow allocation from ObjectFlow to Connector

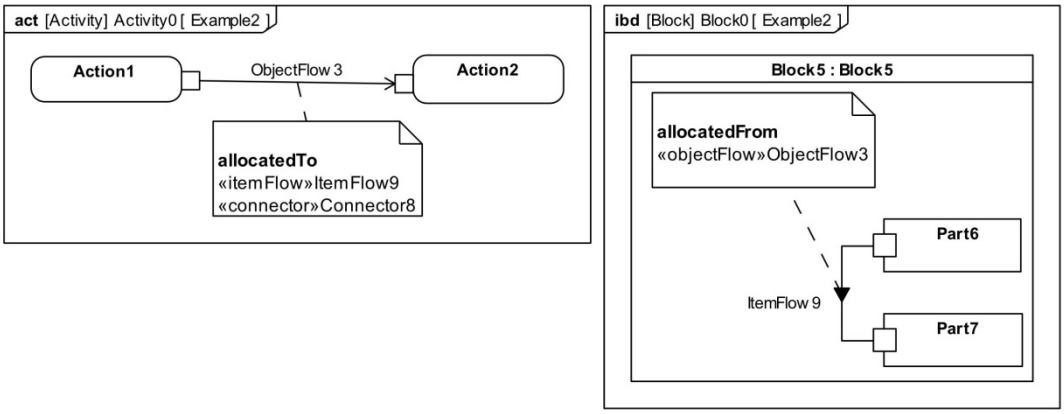


Figure 15-6: Example of flow allocation from ObjectFlow to ItemFlow

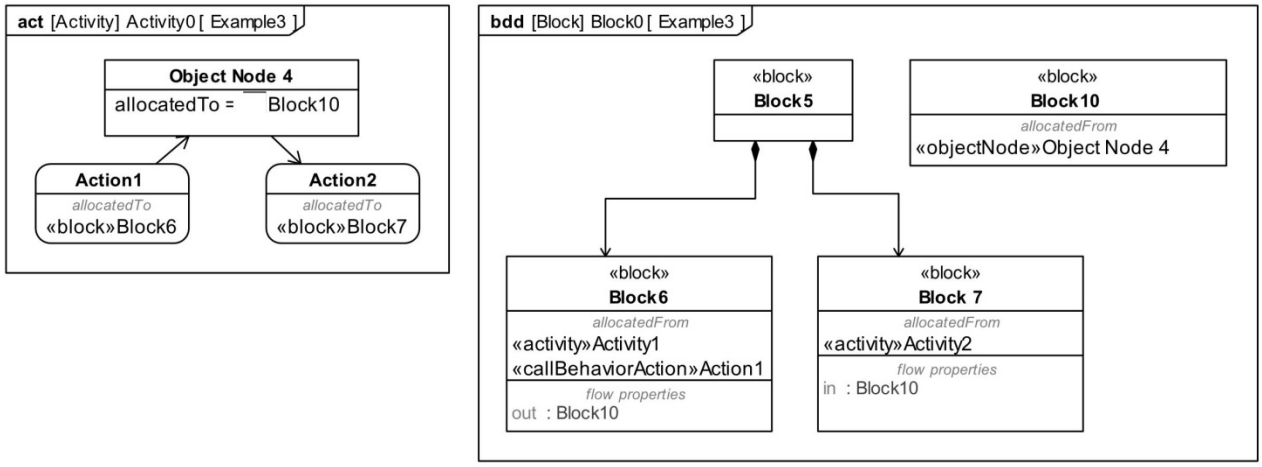


Figure 15-7: Example of flow allocation from ObjectNode to FlowProperty

15.4.2.1 Allocating Structure

Systems engineers have frequent need to allocate structural model elements (e.g., blocks, parts, or connectors) to other structural elements. For example, if a particular user model includes an abstract logical structure, it may be important to show how these model elements are allocated to a more concrete physical structure. The need also arises, when adding detail to a structural model, to allocate a connector (at a more abstract level) to a part (at a more concrete level).

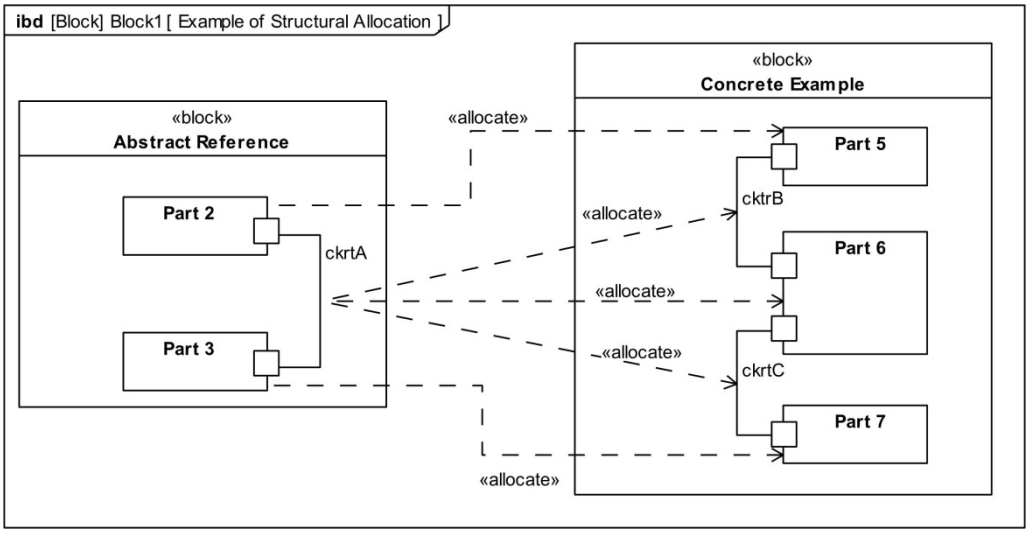


Figure 15-8: Example of Structural Allocation

15.4.2.2 Automotive Example

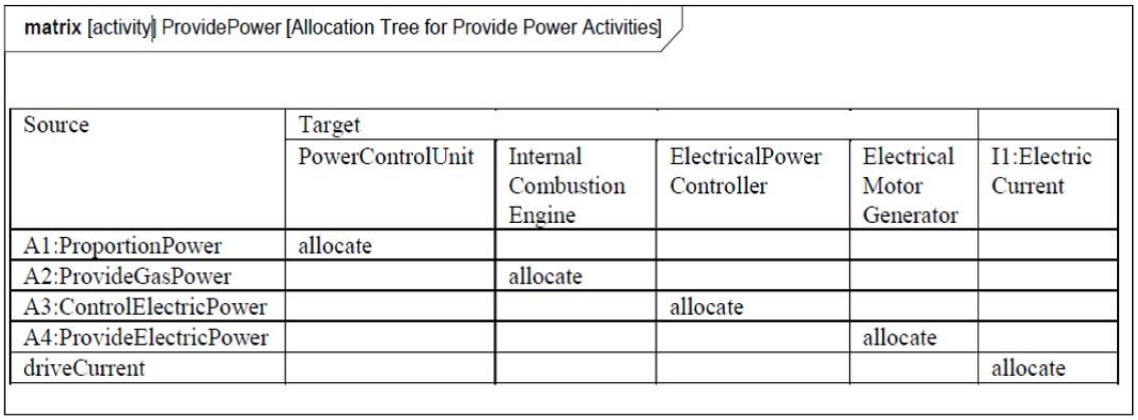
Example: consider the functions required to portion and deliver power for a hybrid SUV. The activities for providing power are allocated to blocks within the Hybrid SUV, as shown in Figure D.38.

Figure D.39 shows an internal block diagram showing allocation for the HybridSUV Accelerate example.

15.4.3 Tabular Representation

The table shown in Figure D.40 is provided as a specific example of how the «allocate» dependency may be depicted in tabular form, consistent with the automotive example above.

The allocation table can also be shown using a sparse matrix style as in the following example shown in Figure 15-9.



matrix [activity] ProvidePower [Allocation Tree for Provide Power Activities]

Source	Target				
	PowerControlUnit	Internal Combustion Engine	ElectricalPower Controller	Electrical Motor Generator	I1:Electric Current
A1:ProportionPower	allocate				
A2:ProvideGasPower		allocate			
A3:ControlElectricPower			allocate		
A4:ProvideElectricPower				allocate	
driveCurrent					allocate

Figure 15-9: Allocation Matrix showing Allocation for Hybrid SUV Accelerate Example

16 Requirements

16.1 Overview

A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition a system must achieve. SysML provides modeling constructs to represent text-based requirements and relate them to other modeling elements. The requirements diagram described in this clause can depict the requirements in graphical, tabular, or tree structure format. A requirement can also appear on other diagrams to show its relationship to other modeling elements. The requirements modeling constructs are intended to provide a bridge between traditional requirements management tools and the other SysML models.

A requirement is defined as a stereotype of UML Class subject to a set of constraints. A standard requirement includes properties to specify its unique identifier and text requirement. Additional properties such as verification status, can be specified by the user.

Several requirements relationships are specified that enable the modeler to relate requirements to other requirements as well as to other model elements. These include relationships for defining a requirements hierarchy, deriving requirements, satisfying requirements, verifying requirements, and refining requirements.

A composite requirement can contain subrequirements in terms of a requirements hierarchy, specified using the UML namespace containment mechanism. This relationship enables a complex requirement to be decomposed into its containing child requirements. A composite requirement may state that the system shall do A and B and C, which can be decomposed into the child requirements that the system shall do A, the system shall do B, and the system shall do C. An entire specification can be decomposed into children requirements, which can be further decomposed into their children to define the requirements hierarchy.

There is a real need for requirement reuse across product families and projects. Typical scenarios are regulatory, statutory, or contractual requirements that are applicable across products and/or projects and requirements that are reused across product families (versions/variants). In these cases, one would like to be able to reference a requirement, or requirement set in multiple contexts with updates to the original requirements propagated to the reused requirement(s).

The use of namespace containment to specify requirements hierarchies precludes reusing requirements in different contexts since a given model element can only exist in one namespace. Since the concept of requirements reuse is very important in many applications, SysML introduces the concept of a slave requirement. A slave requirement is a requirement whose text property is a read-only copy of the text property of a master requirement. The text property of the slave requirement is constrained to be the same as the text property of the related master requirement. The master/slave relationship is indicated by the use of the copy relationship.

The “derive requirement” relationship relates a derived requirement to its source requirement. This typically involves analysis to determine the multiple derived requirements that support a source requirement. The derived requirements generally correspond to requirements at the next level of the system hierarchy. A simple example may be a vehicle acceleration requirement that is analyzed to derive requirements for engine power, vehicle weight, and body drag.

The satisfy relationship describes how a design or implementation model satisfies one or more requirements. A system modeler specifies the system design elements that are intended to satisfy the requirement. In the example above, the engine design satisfies the engine power requirement.

The verify relationship defines how a test case or other model element verifies a requirement. In SysML, a test case or other named element can be used as a general mechanism to represent any of the standard verification methods for inspection, analysis, demonstration, or test. Additional subclasses can be defined by the user if required to represent the

different verification methods. A verdict property of a test case can be used to represent the verification result. The SysML test case is defined consistent with the UML testing profile to facilitate integration between the two profiles.

The refine requirement relationship can be used to describe how a model element or set of elements can be used to further refine a requirement. For example, a use case or activity diagram may be used to refine a text-based functional requirement. Alternatively, it may be used to show how a text-based requirement refines a model element. In this case, some elaborated text could be used to refine a less fine-grained model element.

A generic trace requirement relationship provides a general-purpose relationship between a requirement and any other model element. The semantics of trace include no real constraints and therefore are quite weak. As a result, it is recommended that the trace relationship not be used in conjunction with the other requirements relationships described above.

The rationale construct that is defined in Clause 7, “Model Elements” is quite useful in support of requirements. It enables the modeler to attach a rationale to any requirements relationship or to the requirement itself. For example, a rationale can be attached to a satisfy relationship that refers to an analysis report or trade study that provides the supporting rationale for why the particular design satisfies the requirement. Similarly, this can be used with the other relationships such as the derive relationship. It also provides an alternative mechanism to capture the verify relationship by attaching a rationale to a satisfy relationship that references a test case.

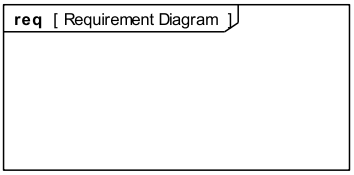
Modelers can customize requirements taxonomies by defining additional subclasses of the Requirement stereotype. For example, a modeler may want to define requirements categories to represent operational, functional, interface, performance, physical, storage, activation/deactivation, design constraints, and other specialized requirements such as reliability and maintainability, or to represent a high level stakeholder need. The stereotype enables the modeler to add constraints that restrict the types of model elements that may be assigned to satisfy the requirement. For example, a functional requirement may be constrained so that it can only be satisfied by a SysML behavior such as an activity, state machine, or interaction. Some potential Requirement subclasses are defined in Annex E.3.

Some users may want a more explicit way to model numerical values and equations as expressed in requirements. Annex E.8 provides examples of non-normative extensions to SysML that meet this need.

16.2 Diagram Elements

16.2.1 Requirement Diagram

Table 16-1: Graphical nodes included in Requirement diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Requirement Diagram		**SysML::Requirements::Requirement, SysML::ModelElements::Package**

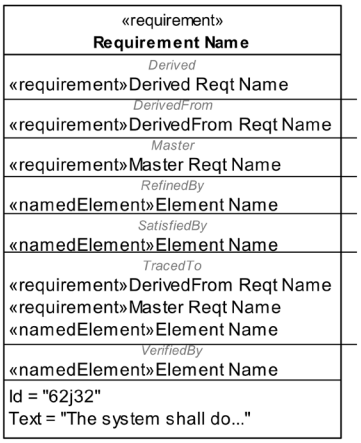
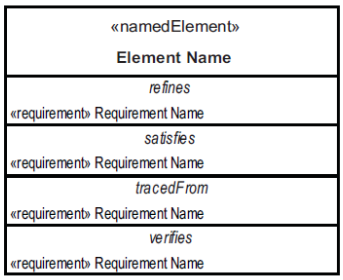
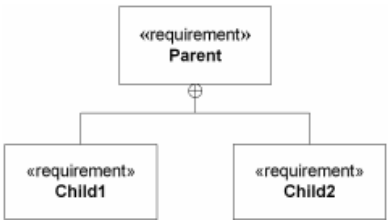
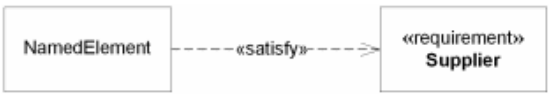
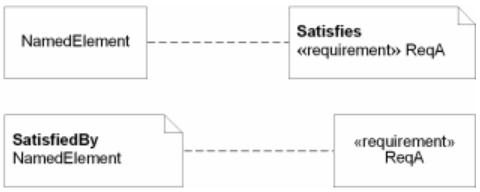
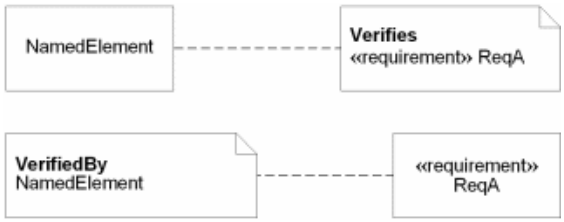
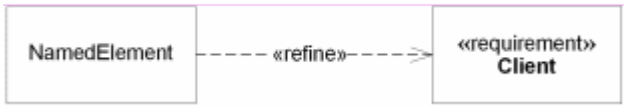
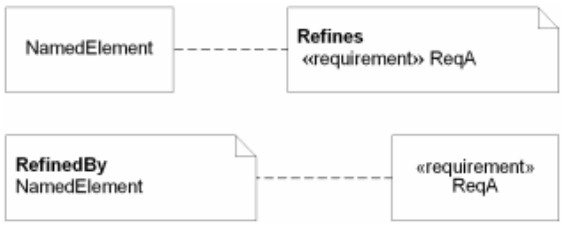
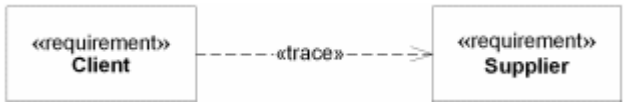
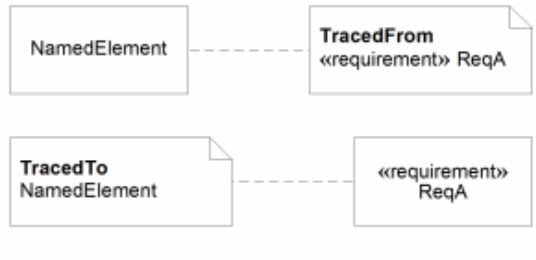
Node Name	Concrete Syntax	Abstract Syntax Reference
Requirement		**SysML::Requirements::Requirement**
NamedElement		**UML4SysML::NamedElement**

Table 16-2: Graphical paths included in Requirement diagrams

Path Type	Concrete Syntax	Abstract Syntax Reference
Requirement containment relationship		**UML4SysML::Nested Classifier**

Path Type	Concrete Syntax	Abstract Syntax Reference
Copy Dependency		**SysML::Requirements::Copy**
MasterCallout		**SysML::Requirements::Copy**
DeriveDependency		**SysML::Requirements:: DeriveReqt**
DeriveCallout		**SysML::Requirements:: DeriveReqt**
SatisfyDependency		**SysML::Requirements::Satisfy**
SatisfyCallout		**SysML::Requirements::Satisfy**
VerifyDependency		**SysML::Requirements::Verify**

Path Type	Concrete Syntax	Abstract Syntax Reference
VerifyCallout		**SysML::Requirements::Verify**
RefineDependency		**UML4SysML::Refine**
RefineCallout		**UML4SysML::Refine**
TraceDependency		**UML4SysML::Trace**
TraceCallout		**UML4SysML::Trace**

16.3 UML Extensions

16.3.1 Diagram Extensions

16.3.1.1 Requirement Diagram

The Requirement Diagram can only display requirements, packages, other classifiers, test cases, and rationale. The relationships for containment, deriveReqt, satisfy, verify, refine, copy, and trace can be shown on a requirement diagram. The callout notation can also be used to reflect the relationship of other model elements to a requirement.

16.3.1.2 Requirement Notation

The requirement is represented as shown in Table 16-1. The «requirement» compartment label for the stereotype properties compartment (e.g., id and text) can be elided.

16.3.1.3 Requirement Property Callout Format

A callout notation can be used to represent derive, satisfy, verify, refine, copy, and trace relationships as indicated in Table 16-2. For brevity, the «elementType» may be elided.

16.3.1.4 Requirements on Other Diagrams

Requirements can also be represented on other diagrams to show their relationship to other model elements. The compartment and callout notation described in 16.3.1.2, Requirement Notation and 16.3.1.3, Requirement Property Callout Format can be used. The callouts represent the requirement that is attached to another model element such as a design element.

16.3.1.5 Requirements Table

The tabular format is used to represent the requirements, their properties and relationships, and may include:

- Requirements with their properties in columns.
- A column that includes the supplier for any of the dependency relationships (Derive, Verify, Refine, Trace).
- A column that includes the model elements that satisfy the requirement.
- A column that represents the rationale for any of the above relationships, including reference to analysis reports for trace rationale, trade studies for design rationale, or test procedures for verification rationale.

The relationships between requirements and other objects can also be shown using a sparse matrix style that is similar to the table used for allocations (15.4.3, Tabular Representation). The table should include the source and target elements names (and optionally kinds) and the requirement dependency kind.

16.3.2 Stereotypes

Package Requirements

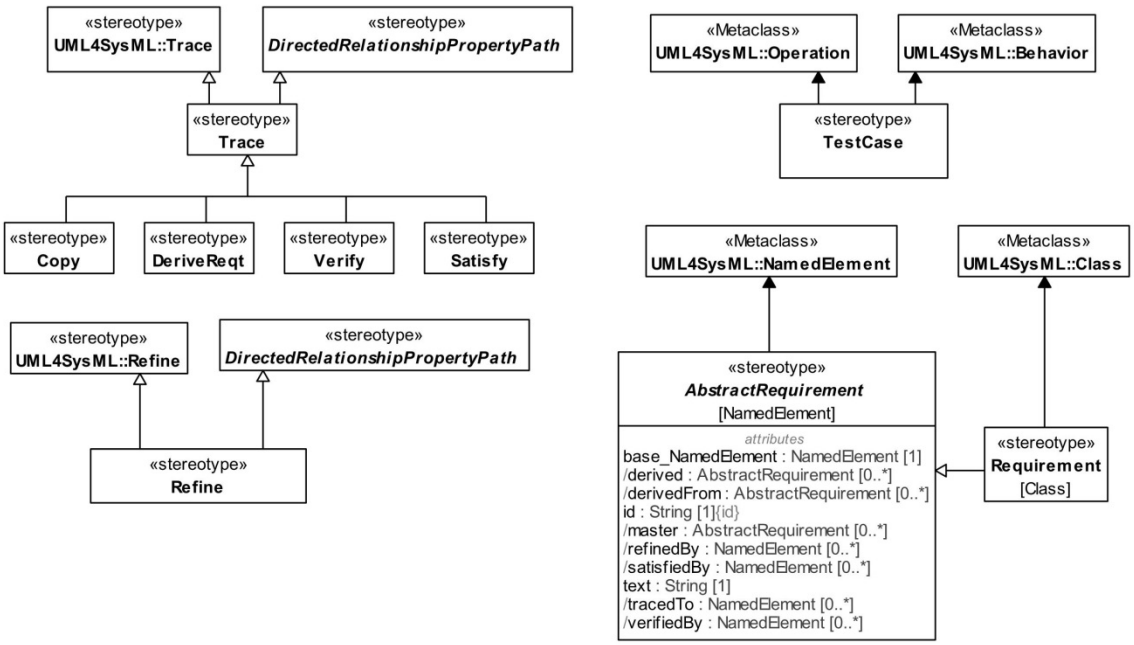


Figure 16-1: Abstract Syntax for Requirements Stereotypes

16.3.2.1 AbstractRequirement

Description

An AbstractRequirement establishes the attributes and relationships essential to any potential kind of requirement. Any intended requirement kind should subclass AbstractRequirement. The only normative stereotype based on AbstractRequirement is the Requirement stereotype, described in 16.3.2.5. Examples of additional non-normative stereotypes based on AbstractRequirement are included in E.8.

Attributes

- base_NamedElement : NamedElement [1]
- /derived : AbstractRequirement [0..*]
  Derived from all requirements that are the client of a «deriveReqt» relationship for which this requirement is a supplier.
  (derived)

- /derivedFrom : AbstractRequirement [0..*]
  Derived from all requirements that are the supplier of a «deriveReqt» relationship for which this requirement is a client.
  (derived)
- id : String [1]
  The unique id of the requirement.
- /master : AbstractRequirement [0..*]
  This is a derived property that lists the master requirement for this slave requirement. The master attribute is derived from the supplier of the Copy dependency that has this requirement as the slave.
  (derived)
- /refinedBy : NamedElement [0..*]
  Derived from all elements that are the client of a «refine» relationship for which this requirement is a supplier.
  (derived)
- /satisfiedBy : NamedElement [0..*]
  Derived from all elements that are the client of a «satisfy» relationship for which this requirement is a supplier.
  (derived)
- text : String [1]
  The textual representation or a reference to the textual representation of the requirement.
- /tracedTo : NamedElement [0..*]
  Derived from all elements that are the client of a «trace» relationship for which this requirement is a supplier.
  (derived)
- /verifiedBy : NamedElement [0..*]
  Derived from all elements that are the client of a «verify» relationship for which this requirement is a supplier.
  (derived)

Operations

- getDerived () : AbstractRequirement [0..*]

```
bodyCondition:
DeriveReqt.allInstances()
->select(base_Abstraction.supplier=self).base_Abstraction.client
```

- getDerivedFrom () : AbstractRequirement [0..*]

```
bodyCondition:
DeriveReqt.allInstances()
->select(base_Abstraction.client=self).base_Abstraction.supplier
```

- getMaster () : AbstractRequirement [0..*]

```
bodyCondition:
Copy.allInstances()->select(base_Abstraction.client=self).base_Abstraction.supplier
```

- getRefinedBy () : NamedElement [0..*]

```
bodyCondition:
Refine.allInstances()->select(base_Abstraction.supplier=self).base_Abstraction.client
```

- getSatisfiedBy () : NamedElement [0..*]

```
bodyCondition:
Satisfy.allInstances()
->select(base_Abstraction.supplier=self).base_Abstraction.client
```

- getTracedTo () : NamedElement [0..*]

```
bodyCondition:
Trace.allInstances()->select(base_Abstraction.client=self).base_Abstraction.supplier
```

- getVerifiedBy () : NamedElement [0..*]

```
bodyCondition:
Verify.allInstances()->select(base_Abstraction.supplier=self).base_Abstraction.client
```

16.3.2.2 Copy

Description

A Copy relationship is a dependency between a supplier requirement and a client requirement that specifies that the text of the client requirement is a read-only copy of the text of the supplier requirement.

A Copy dependency created between two requirements maintains a master/slave relationship between the two elements for the purpose of requirements re-use in different contexts. When a Copy dependency exists between two requirements, the requirement text of the client requirement is a read-only copy of the requirement text of the requirement at the supplier end of the dependency.

Generalizations

- Trace (from Requirements)

Operations

- isCopy (in req1 : AbstractRequirement, in req2 : AbstractRequirement) : Boolean [1]

```
bodyCondition:
let subReq1: Set(AbstractRequirement) = AbstractRequirement.allInstances()
->select(r | req1.base_NamedElement.ownedElement->includes(r.base_NamedElement)) in
let subReq2: Set(AbstractRequirement) = AbstractRequirement.allInstances()
->select(r | req2.base_NamedElement.ownedElement->includes(r.base_NamedElement)) in
req1.text = req2.text and subReq1->size() = subReq2->size() and
subReq1->forAll(r1 | subReq2->exists(r2 | self.isCopy(r1, r2) ))
```

Constraints

- 1_source_and_taget_are_requirements
  A Copy dependency may only be created between two NamedElements that have a subtype of the abstractRequirement stereotype applied

```
AbstractRequirement.allInstances().base_NamedElement
->includesAll(self.base_Abstraction.client) and
AbstractRequirement.allInstances().base_NamedElement
->includesAll(self.base_Abstraction.supplier)
```

- 2_same_text
  The text property of the client requirement is constrained to be a read-only copy of the text property of the supplier requirement and this applies recursively to all subrequirements

```
let cltReq: AbstractRequirement = AbstractRequirement.allInstances()->any(r |
self.base_Abstraction.client->includes(r.base_NamedElement)) in let supReq:
AbstractRequirement = AbstractRequirement.allInstances()->any(r |
self.base_Abstraction.supplier->includes(r.base_NamedElement)) in self.isCopy(cltReq,
supReq)
```

16.3.2.3 DeriveReqt

Description

A DeriveReqt relationship is a dependency between two requirements in which a client requirement can be derived from the supplier requirement. For example, a system requirement may be derived from a business need, or lower-level requirements may be derived from a system requirement. As with other dependencies, the arrow direction points from the derived (client) requirement to the (supplier) requirement from which it is derived.

Generalizations

- Trace (from Requirements)

Constraints

- 1_supplier_is_requirement
  The supplier shall be an element stereotyped by a subtype of AbstractRequirement.

```
AbstractRequirement.allInstances().base_NamedElement
->includesAll(self.base_Abstraction.client)
```

- 2_client_is_requirement
  The client shall be an element stereotyped by a subtype of AbstractRequirement.

```
AbstractRequirement.allInstances().base_NamedElement
->includesAll(self.base_Abstraction.supplier)
```

16.3.2.4 Refine

Description

The Refine stereotype specializes UML4SysML Refine and DirectedRelationshipPropertyPath to enable refinements to identify their sources and targets by a multi-level path of accessible properties from context blocks for the sources and targets.

Generalizations

- DirectedRelationshipPropertyPath (from Blocks)

Association Ends

- base_Abstraction : Abstraction [1]
  (redefines: DirectedRelationshipPropertyPath::base_DirectedRelationship)

Operations

- getRefines (in ref : NamedElement) : AbstractRequirement [0..*]
  The query getRefines() gives all the requirements that are suppliers ("to"end of the concrete syntax) of a «Refine» relationships whose client is the element in parameter. This is a static query.

```
bodyCondition:
Refine.allInstances()->select(base_Abstraction.client=ref).base_Abstraction.supplier
```

Constraints

- 2_binary
  Abstractions with a Refine stereotype or one of its specializations applied shall have exactly one client and one supplier.

```
self.base_Abstraction.client->size()=1 and self.base_Abstraction.supplier->size()=1
```

16.3.2.5 Requirement

Description

A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition that a system must satisfy. Requirements are used to establish a contract between the customer (or other stakeholder) and those responsible for designing and implementing the system.

A requirement is a stereotype of both Class and Abstract Requirement. Compound requirements can be created by using the nesting capability of the class definition mechanism. The default interpretation of a compound requirement, unless stated differently by the compound requirement itself, is that all its subrequirements shall be satisfied for the compound requirement to be satisfied. Subrequirements shall be accessed through the "nestedClassifier" property of a class. When a requirement has nested requirements, all the nested requirements apply as part of the container requirement. Deleting the container requirement deleted the nested requirements, a functionality inherited from UML.

Generalizations

- AbstractRequirement (from Requirements)

Association Ends

- base_Class : Class [1]
  (redefines: AbstractRequirement::base_NamedElement)

Constraints

- 1_no_operation
  The property "ownedOperation" shall be empty.
  `self.base_Class.ownedOperation->isEmpty()`

- 2_no_attribute
  The property "ownedAttribute" shall be empty.
  `self.base_Class.ownedAttribute->isEmpty()`

- 3_no_association
  Classes stereotyped by «requirement» shall not participate in associations.
  `UML::Association.allInstances().memberEnd->flatten().type->excludes(self.base_Class)`

- 4_no_generalization
  Classes stereotyped by «requirement» shall not participate in generalizations.
  `UML::Classifier.allInstances().general->flatten()->excludes(self.base_Class)`

- 5_nestedclassifiers_are_requirements
  A nested classifier of a class stereotyped by Requirement or one of its specializations shall also be stereotyped by Requirement or one of its specializations
  ```
  self.base_Class.nestedClassifier->forAll(c | Requirement.allInstances().base_Class
  ->includes(c))
  ```

- 6_not_a_type
  Classes stereotyped by «requirement» shall not be used to type any other model element.
  `UML::TypedElement.allInstances().type->excludes(self.base_Class)`

16.3.2.6 TestCase

Description

A test case is a method for verifying a requirement is satisfied.

Association Ends

- base_Behavior : Behavior [0..1]
- base_Operation : Operation [0..1]

Constraints

- 1_return_verdictkind
  The type of return parameter of the stereotyped model element shall be VerdictKind. (note this is consistent with the UML Testing Profile).
  ```
  (self.base_Behavior->notEmpty() implies self.base_Behavior.ownedParameter->exists(p |
  p.direction=UML::ParameterDirectionKind::return and p.type = VerdictKind )) and
  ```

```
(self.base_Operation->notEmpty() implies self.base_Operation.ownedParameter->exists(p
| p.direction=UML::ParameterDirectionKind::return and p.type = VerdictKind ))
```

16.3.2.7 Satisfy

Description

A Satisfy relationship is a dependency between a requirement and a model element that fulfills the requirement. As with other dependencies, the arrow direction points from the satisfying (client) model element to the (supplier) requirement that is satisfied.

Generalizations

- Trace (from Requirements)

Operations

- getSatisfies (in ref : NamedElement) : AbstractRequirement [0..*]

```
bodyCondition:
Satisfy.allInstances()->select(base_Abstraction.client=ref).base_Abstraction.supplier
```

Constraints

- `1_supplier_is_requirement`
  The supplier shall be an element stereotyped by any subtype of «AbstractRequirement».

```
AbstractRequirement.allInstances().base_NamedElement
->includes(self.base_Abstraction.supplier)
```

16.3.2.8 Trace

Description

The Trace stereotype specializes UML4SysML Trace and DirectedRelationshipPropertyPath to enable traces to identify their sources and targets by a multi-level path of accessible properties from context blocks for the sources and targets.

Generalizations

- DirectedRelationshipPropertyPath (from Blocks)

Association Ends

- base_Abstraction : Abstraction [1]
  (redefines: DirectedRelationshipPropertyPath::base_DirectedRelationship)

Operations

- getTracedFrom (in ref : NamedElement) : AbstractRequirement [0..*]
  The query getTracedFrom() gives all the requirements that are clients ("from" end of the concrete syntax) of a «Trace» relationship whose supplier is the element in parameter. This is a static query.

```
bodyCondition:
AbstractRequirement.allInstances()->select(tracedTo->includes(ref))
```

Constraints

- 2_binary
  Abstractions with a Trace stereotype or one of its specializations applied shall have exactly one client and one supplier.

```
self.base_Abstraction.client->size()=1 and self.base_Abstraction.supplier->size()=1
```

16.3.2.9 Verify

Description

A Verify relationship is a dependency between a requirement and a test case or other model element that can determine whether a system fulfills the requirement. As with other dependencies, the arrow direction points from the (client) element to the (supplier) requirement.

Generalizations

- Trace (from Requirements)

Operations

- getVerifies (in ref : NamedElement) : AbstractRequirement [0..*]
  The query getVerifies() gives all the requirements that are suppliers ("to" end of the concrete syntax) of a «Verify» relationships whose client is the element in parameter. This is a static query.

```
bodyCondition:
Verify.allInstances()->select(base_Abstraction.client=ref).base_Abstraction.supplier
```

Constraints

- 1_supplier_is_requirement
  The supplier shall be an element stereotyped by any subtype of «AbstractRequirement».

```
AbstractRequirement.allInstances().base_NamedElement
->includes(self.base_Abstraction.supplier)
```

16.4 Usage Examples

The examples in this clause show the use of the normative Requirement stereotypes. Examples showing the definition and use of non-normative requirement stereotypes based on AbstractRequirement are shown in Annex E.8. All the examples in this clause are based on a set of publicly available (on-line) requirement specifications from the *National*

Highway Traffic Safety Administration (NHTSA). Excerpts of the original requirement text used to create the models are shown in Figure 16.2. The name and ID of these requirements are referred to in the SysML usage examples that follow. See *NHTSA specification 49CFR571.135* for the complete text from which these examples are taken.

16.4.1 Requirement Decomposition and Traceability

The diagram in Figure 16-2 shows an example of a compound requirement decomposed into multiple subrequirements.

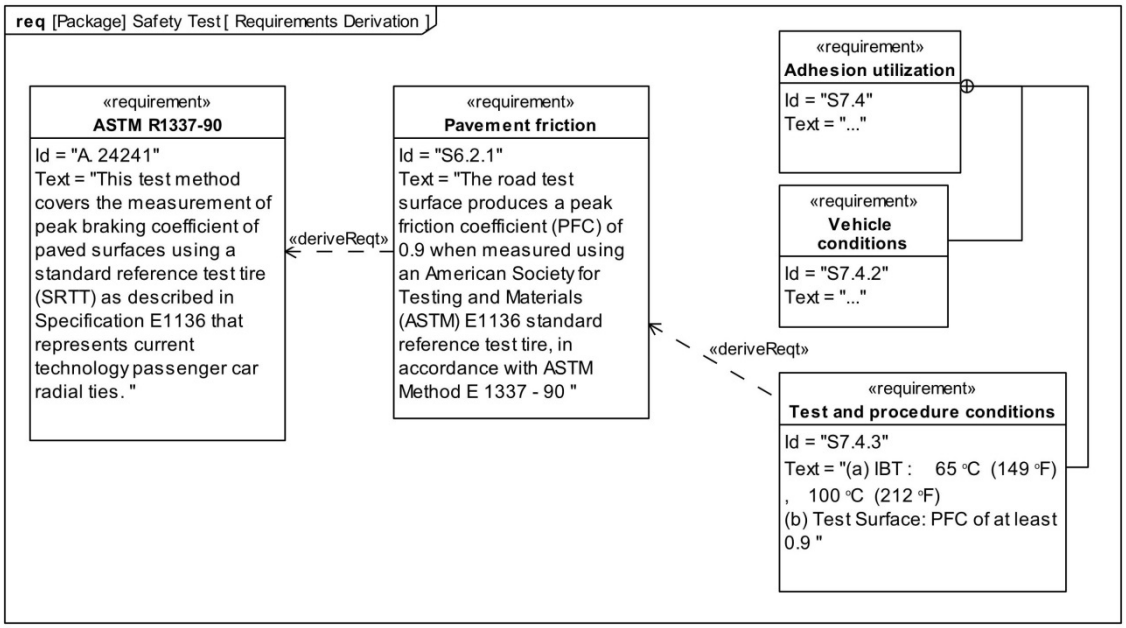


Figure 16-2: Requirements Derivation

16.4.2 Requirements and Design Elements

The diagram in Figure 16-3 shows derived requirements and refers to the design elements that satisfy them. The rationale is also shown as a basis for the design solution.

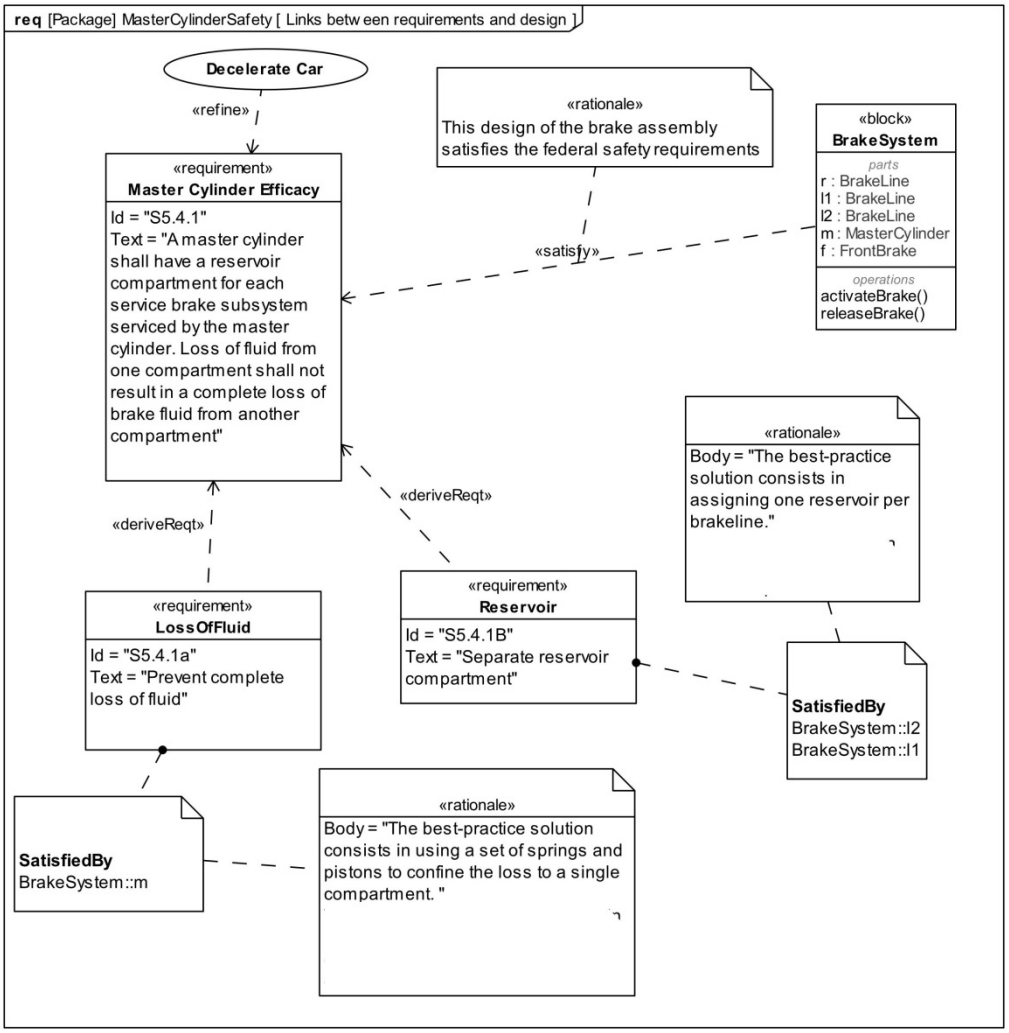


Figure 16-3: Links between requirements and design

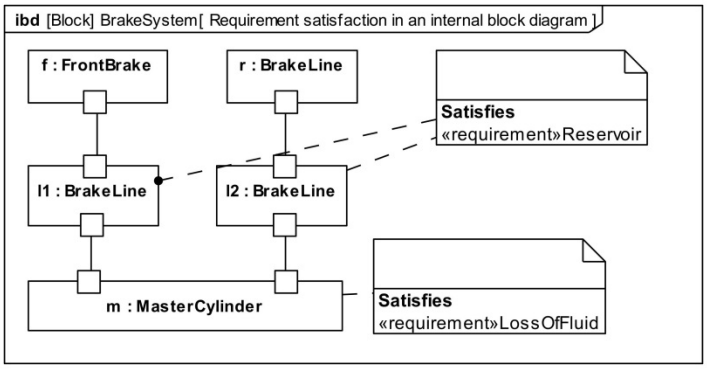


Figure 16-4: Requirement satisfaction in an internal block diagram

16.4.3 Requirements Reuse

Figure 16-5 illustrates the use of the Copy dependency to allow a single requirement to be reused in several requirements hierarchies. The master tag provides a textual reference to the reused requirement.

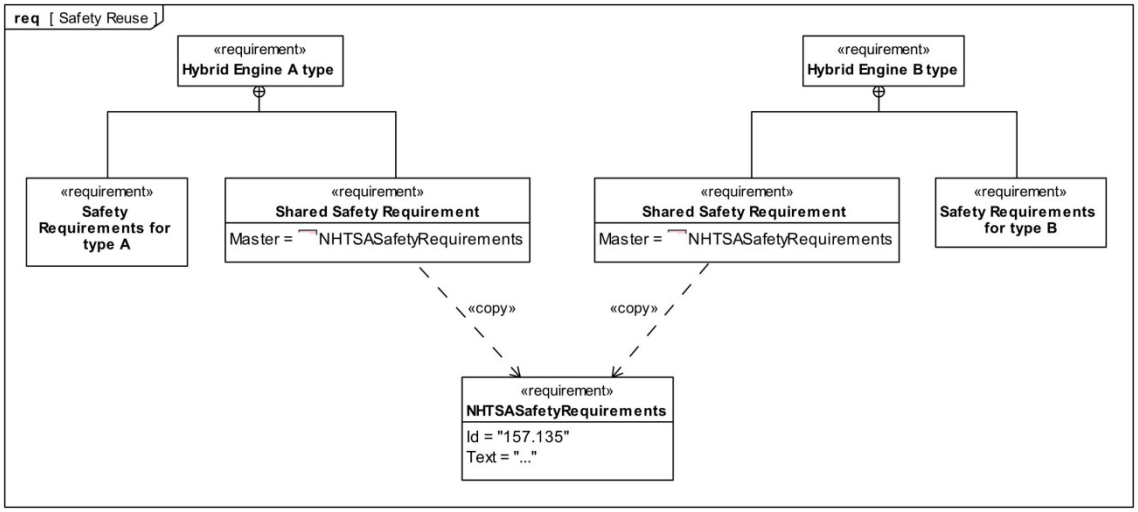


Figure 16-5: Use of the copy dependency to facilitate reuse

16.4.4 Verification Procedure (Test Case)

The example in Figure 16-6 is taken from the automotive safety domain, and shows a Burnish requirement contained in the NHTSASafetyRequirements requirement. Note that the text of the Burnish requirement indicates a specific sequence of steps and transition criteria. The Burnish requirement is shown as having a Verify relationship to the BurnishTest test case using callout notation on the diagram, indicating that the Burnish requirement is verified by the BurnishTest test case.

Figure 17-1 is a state machine diagram of the BurnishTest test case, which expresses the textual sequence and criteria of the Burnish requirement in state machine form. The Verify relationship is shown on Figure 16-7 using callout notation anchored to the diagram frame, which indicates that the BurnishTest test case verifies the Burnish requirement.

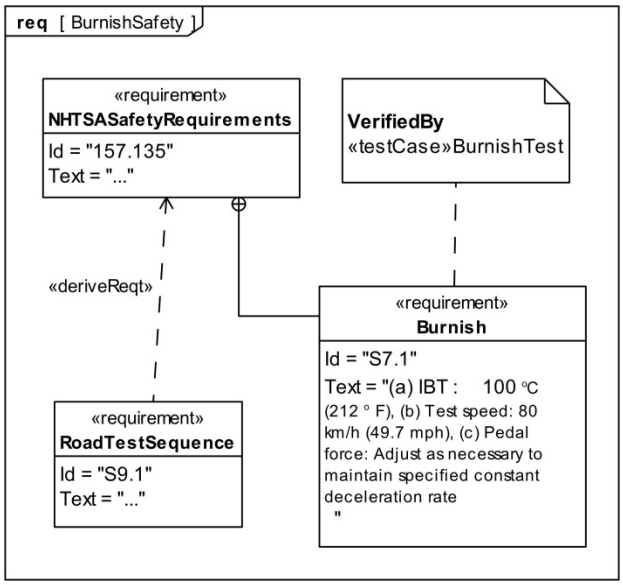


Figure 16-6: Linkage of a Test Case to a requirement: This figure shows the Requirement Diagram

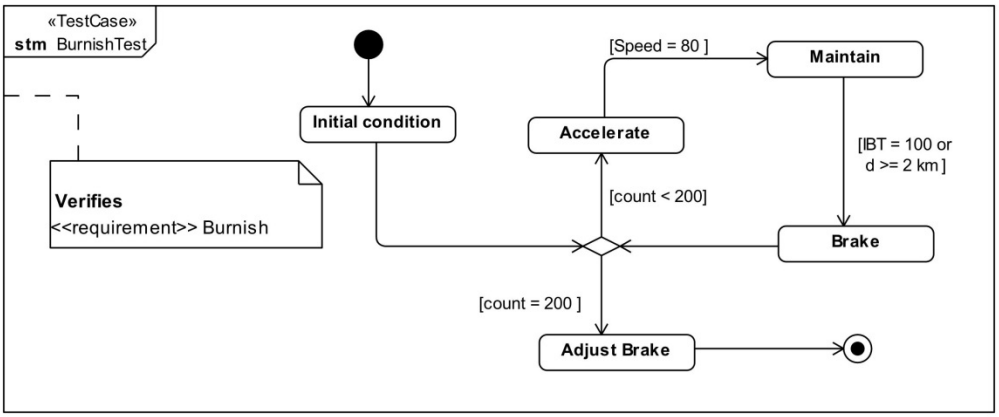


Figure 16-7: Linkage of a Test Case to a requirement: This figure shows the Test Case as a State Diagram

17 Profiles & Model Libraries

17.1 Overview

The Profiles package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different domains. The profiles mechanism is consistent with the OMG Meta Object Facility (MOF). SysML has added some notational extensions to represent stereotype properties in compartments as well as notes.

The stereotype is the primary mechanism used to create profiles to extend the metamodel. Stereotypes are defined by extending a metaclass, and then have them applied to the applicable model elements in the user model. A stereotype of a requirement could be extended to create a «functionalRequirement» as described in Annex E: “Non-normative Extensions.” This would allow specific properties and constraints to be created for a functional requirement. For example, a functional requirement may be constrained such that it must be satisfied by an operation or behavior. When the stereotype is applied to a requirement, then the requirement would include the notation «functionalRequirement» in addition to the name of the particular functional requirement. Extending the metaclass requirement is different from creating a subclass of requirement called functionalRequirement.

The Usage Examples sub clause provides guidance both on how to use existing profiles and how to create new profiles. In addition, the examples provide guidance on the use of model libraries. A model library is a library of model elements including class and other type definitions that are considered reusable for a given domain. These guidelines can be applied to further customize SysML for domain specific applications such as automotive, military, or space systems.

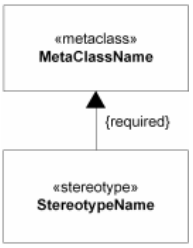
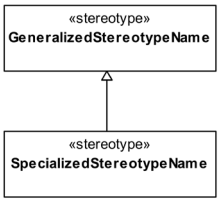
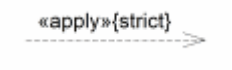
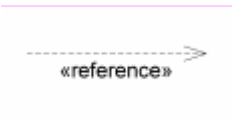
17.2 Diagram Elements

Table 17-1: Graphical nodes used in profile definition

Node Name	Concrete Syntax	Abstract Syntax Reference
Stereotype	«stereotype» **StereotypeName**	**UML4SysML::Stereotype**
Metaclass	«metaclass» **MetaClassName**	**UML4SysML::Class**
Profile	«profile» **ProfileName**	**UML4SysML::Profile**

Node Name	Concrete Syntax	Abstract Syntax Reference
Model Library		UML::StandardProfile

Table 17-2: Graphical paths used in profile definition

Path Name	Concrete Syntax	Abstract Syntax Reference
Extension		UML4SysML::Extension
Generalization		UML4SysML::Generalization
ProfileApplication		UML4SysML::Profile Application
MetamodelReference		UML4SysML::PackageImport; UML4SysML::ElementImport
Unidirectional Association		UML4SysML::Association

NOTE: In the above table, boolean properties can be displayed alternatively as BooleanPropertyName=[True|False].

17.2.1.1 Extension

In Figure 17-1, a simple stereotype Clock is defined to be applicable at will (dynamically) to instances of the metaclass Class and describes a clock software component for an embedded software system. It has description of the operating system version supported, an indication of whether it is compliant to the POSIX operating system standard and a reference to the operation that starts the clock.

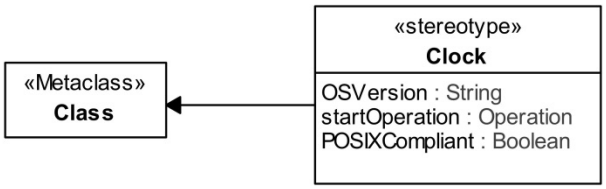
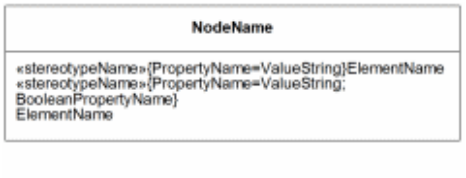
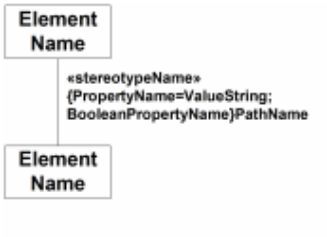
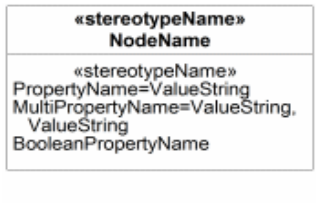


Figure 17-1: Defining a stereotype

17.2.2 Stereotypes Used On Diagrams

Table 17-3: Notations for Stereotype Use

Node Name	Concrete Syntax	Abstract Syntax Reference
StereotypeNode	«stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName Element Name — PathName — Element Name	**UML4SysML::Element**
StereotypeNode	«stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName Element Name	**UML4SysML::Element**
StereotypeInNode	«stereotypeName» {PropertyName=ValueString; BooleanPropertyName} **NodeName**	**UML4SysML::Element**

StereotypeInCompartmentElement		**UML4SysML::Element**
StereotypeOnEdge		**UML4SysML::Element**
StereotypeCompartment		**UML4SysML::Element**

17.2.2.1 StereotypeInNode

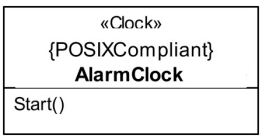


Figure 17-2: Using a stereotype

Figure 17-2 shows how the stereotype Clock, as defined in Figure 17-1, is applied to a class called AlarmClock.

17.2.2.2 StereotypeInComment

When two stereotypes, Clock and Creator, are applied to the same model element, as is shown in Figure 17-3, the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

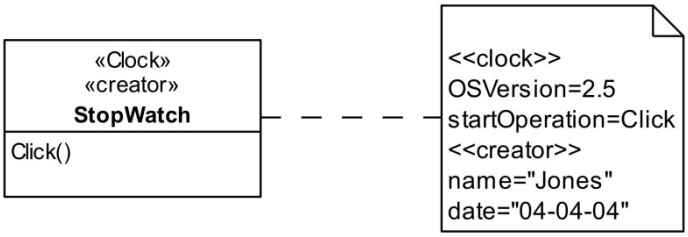


Figure 17-3: Other notational forms for showing values

17.2.2.3 StereotypeInCompartment

Finally, the compartment form is shown.

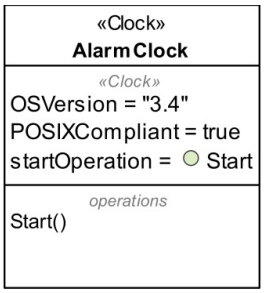


Figure 17-4: Other notational forms for showing values

In this case, AlarmClock is valid for OS version 3.4, is POSIX-compliant and has a starting operation called Start. Note that multiple stereotypes can be shown using multiple compartments.

17.3 UML Extensions

None.

17.4 Usage Examples

17.4.1 Defining a Profile

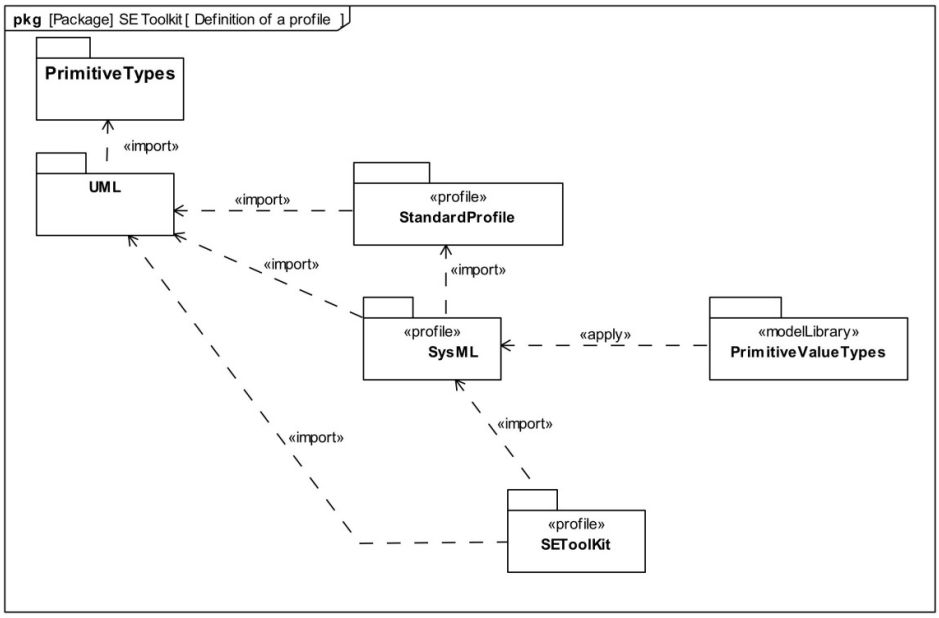


Figure 17-5: Definition of a profile

In this example, the modeler has created a new profile called SE Toolkit, which imports the SysML profile, so that it can build upon the stereotypes it contains. The set of metaclasses available to users of the SysML profile is identified by a reference to a metamodel, in this case a subset of UML specific to SysML. The SE Toolkit can extend those metaclasses from UML that the SysML profile references.

17.4.2 Adding Stereotypes to a Profile

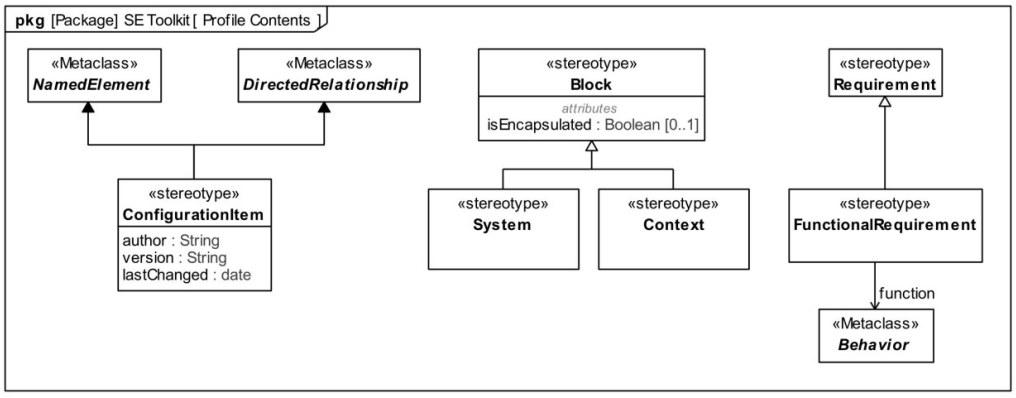


Figure 17-6: Profile Contents

In SE Toolkit, both the mechanisms for adding new stereotypes are used. The first, exemplified by configurationItem, is called an extension, shown by a line with a filled triangle; this relates a stereotype to a reference (called base) class or classes, in this case NamedElement and DirectedRelationship from UML and adds new properties that every NamedElement or DirectedRelationship stereotyped by configurationItem must have. NamedElement and DirectedRelationship are abstract classes in UML so it is their subclasses that can have the stereotype applied. The second mechanism is demonstrated by the system and context stereotypes which are sub-stereotypes of an existing SysML stereotype, Block; sub-stereotypes inherit any properties of their super-stereotype and also extend the same base class or classes. Note that TypedElements whose type is extended by «system» do not display the «system» stereotype; this also applies to InstanceSpecifications. Any notational conventions of this have to be explicitly specified in a diagram extension.

There is also an example of how stereotypes (in this case FunctionalRequirement) can have unidirectional associations to metaclasses in the reference metamodel (in this case Behavior).

17.4.3 Defining a Model Library that Uses a Profile

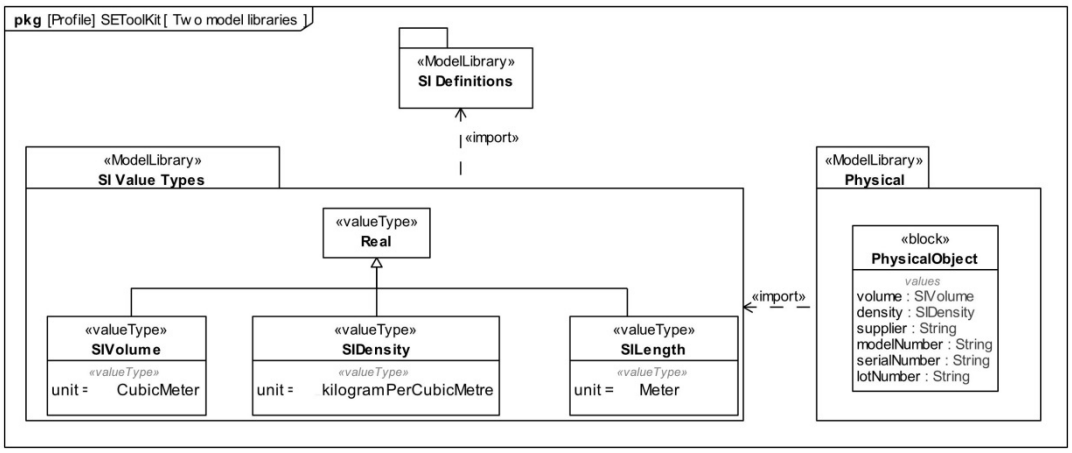


Figure 17-7: Two model libraries

The model library SI Value Types imports a model library called SI Definitions, so it can use model elements from them in its own definition. It defines value types having specific units which can be used when property values are measured in SI units. SI Definitions is a separately published model library, containing definitions of standard SI units and quantity kinds such as shown in Annex D, sub clause D.4. A further model library, Physical, imports SI Value Types so it can define properties that have those types. One model element, PhysicalObject, is shown, a block that can be used as a supertype for a physical object.

17.4.4 Guidance on Whether to Use a Stereotype or Class

This sub clause provides guidance on when to use stereotypes. Stereotypes can be applied to any model element. Stereotyping a model element allows the model element to be identified with the «guillemet» notation. In addition, the stereotyped model element can have stereotype properties, and the stereotype can specify constraints on the model element.

The modeler must decide when to create a stereotype of a class versus when to specialize (subclass) the class. One reason is to be able to identify the class with the «guillemet» notation. In addition, the stereotype properties are different from

properties of classes. Stereotype properties represent properties of the class that are not instantiated and therefore do not have a unique value for each instance of the class, although a class thus stereotyped can have a separate value for the property.

SE Toolkit::functionalRequirement, which extends Class through its superstereotype, Requirement, is an example where a stereotype is appropriate because every modeling element stereotyped by SE Toolkit::functionalRequirement has a reference to another modeling element. In another example, SE Toolkit::configurationItem defined above, which applies to classes among other concepts, is a stereotype because its properties characterize the author, version, and last changed date of the modeling element themselves. One test of this is whether the new properties are inheritable; in this case author, version, and last-changed date are not, because it is only those classes under configuration control that need the properties. To summarize, in the following circumstances a stereotype is appropriate:

- Where the model concept to be extended is not a class or class-based.
- Where the extensions include properties that reference other model elements.
- Where the extensions include properties that describe modeling data, not system data.

An example where a class is more appropriate is PhysicalObject from Figure 17-7. In this case, the properties density and volume, and the component numbers, have distinct values for each system element described by the class, and are inherited by every subclass of PhysicalObject.

17.4.5 Using a Profile

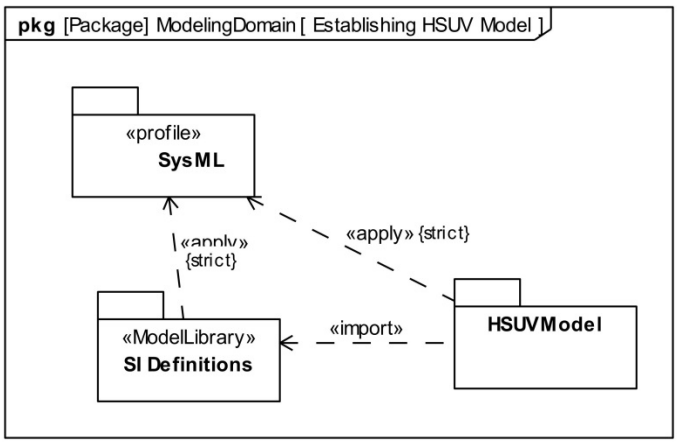


Figure 17-8: A model with applied profile and imported model library

The HSUVModel is a systems engineering model that needs to use stereotypes from SysML. It therefore needs to have the SysML profile applied to it. In order to use the predefined SI units, it also needs to import the SI Definitions model library. Having done this, elements in HSUVModel can be extended by SysML stereotypes and types like SIVolume can be used to type properties. Both the SI Definitions model library and HSUVModel have applied the profile strictly, which means that only those metaclasses directly referenced by SysML can be used in those models.

17.4.6 Using a Stereotype

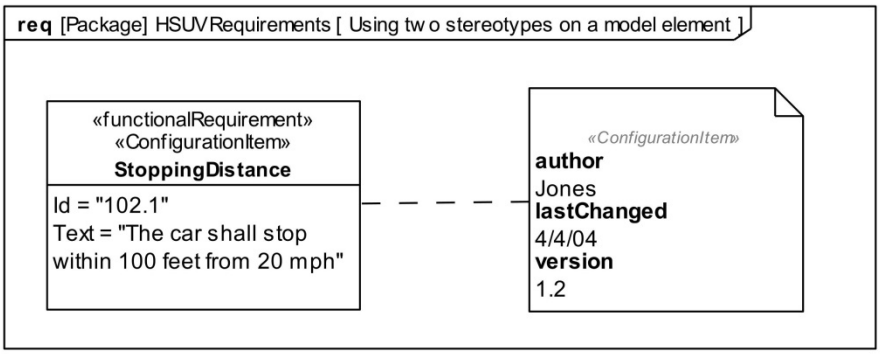


Figure 17-9: Using two stereotypes on a model element

StoppingDistance has two stereotypes applied:

- functionalRequirement, which identifies it as a requirement that is satisfied by a function, and
- configurationItem, which allows it to have configuration management properties.

The modeler has provided values for all the newly available properties; those for criticalRequirement are shown in a compartment in the node symbol for StoppingDistance; those for configurationItem are shown in a separate note.

17.4.7 Using a Model Library Element

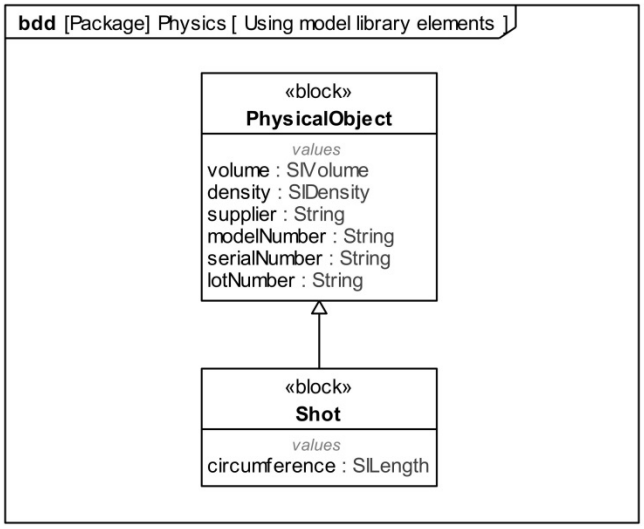


Figure 17-10: Using model library elements

Model library elements can be used just like any other model element of the same type. In this case, Shot is a specialization of PhysicalObject from the Physical model library. It adds a new property, circumference, of type SILength to measure the circumference of the (spherical) shot.

This page intentionally left blank.

ANNEXES

This page intentionally left blank.

Annex A: Diagrams

(informative)

A.1 Overview

SysML diagrams contain diagram elements (mostly nodes connected by paths) that represent model elements in the SysML model, such as activities, blocks, and associations. The diagram elements are referred to as the concrete syntax.

The SysML diagram taxonomy is shown in Figure A.1. This taxonomy is one example of how to organize the SysML diagrams. Other categories could also be defined, such as a grouping of the use case diagram and the requirement diagram into a category called Specification Diagrams.

SysML reuses many of the major diagram types of UML. In some cases, the UML diagrams are strictly reused, such as use case, sequence, state machine, and package diagrams, whereas in other cases they are modified so that they are consistent with SysML extensions. For example, the block definition diagram and internal block diagram are similar to the UML class diagram and composite structure diagram respectively, but include extensions as described in 8, “Blocks.” Activity diagrams have also been modified via the activity extensions. Tabular representations, such as the allocation table, are used in SysML but are not considered part of the diagram taxonomy.

SysML does not use all of the UML diagram types such as the object diagram, communication diagram, interaction overview diagram, timing diagram, deployment diagram, and profile diagram. This is consistent with the approach that SysML represents a subset of UML. In the case of deployment diagrams, the deployment of software to hardware can be represented in the SysML internal block diagram. In the case of interaction overview and communication diagrams, it was felt that the SysML internal block diagram. In the case of interaction overview and communication diagrams, it was felt that the SysML behavior diagrams provided adequate coverage for representing behavior without the need to include these diagram types. In the case of the profile diagram, profile definitions can be captured on a package diagram and the parametric diagram.

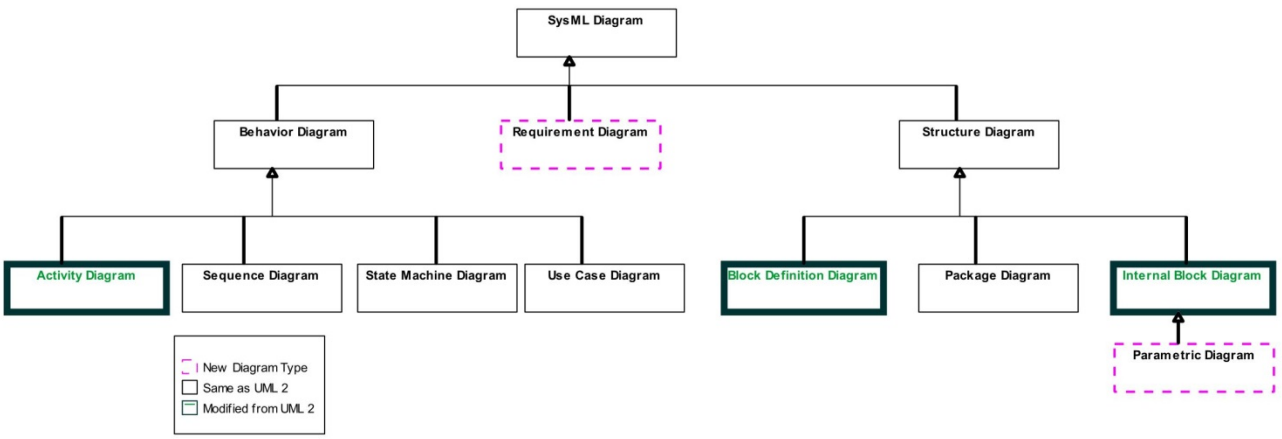


Figure A.1: SysML Diagram Taxonomy

The requirement diagram is a new SysML diagram type. A requirement diagram provides a modeling construct for text-based requirements, and the relationship between requirements and other model elements that satisfy or verify them.

The parametric diagram is a new SysML diagram type that describes the constraints among the properties associated with blocks. This diagram is used to integrate behavior and structure models with engineering analysis models such as performance, reliability, and mass property models.

Although the taxonomy provides a logical organization for the various major kinds of diagrams, it does not preclude the careful mixing of different kinds of diagram types, as one might do when one combines structural and behavioral elements (e.g., showing a state machine nested inside a compartment of a block). However, it is critical that the types of diagram elements that can appear on a particular diagram kind be constrained and well-specified. The diagram elements tables in each clause describe what symbols can appear in the diagram, but do not specify the different combinations of symbols that can be used.

The package diagram and the callout notation are two mechanisms that SysML provides for adding flexibility to represent a broad range of diagram elements on diagrams. The package diagram can be used quite flexibly to organize the model in packages and views. As such, a package diagram can include a wide array of packageable elements. The callout notation provides a mechanism for representing relationships between model elements that appear on different diagram kinds. In particular, they are used to represent allocations and requirements, such as the allocation of an activity to a block on a block definition diagram, or showing a part that satisfies a particular requirement on an internal block diagram. There are other mechanisms for representing this including the compartment notation that is generally described in Clause 17, “Profiles & Model Libraries,” 16, “Requirements,” and 15, “Allocations” provide specific guidance on how these notations are used.

The model elements and corresponding concrete syntax that are represented in each of the nine SysML diagram kinds are described in the SysML clauses as indicated below.

- activity diagram - activity
- block definition diagram - block, package, constraint block, or activity
- internal block diagram - block or constraint block
- package diagram - package, model, modelLibrary, profile
- parametric diagram - block or constraint block
- requirement diagram - package, requirement, modelLibrary, model
- sequence diagram - interaction
- state machine diagram - state machine
- use case diagram - package, block, model, modelLibrary

Each SysML diagram has a frame, with a contents area, a heading, and a Diagram Description (see Figure 17.12).

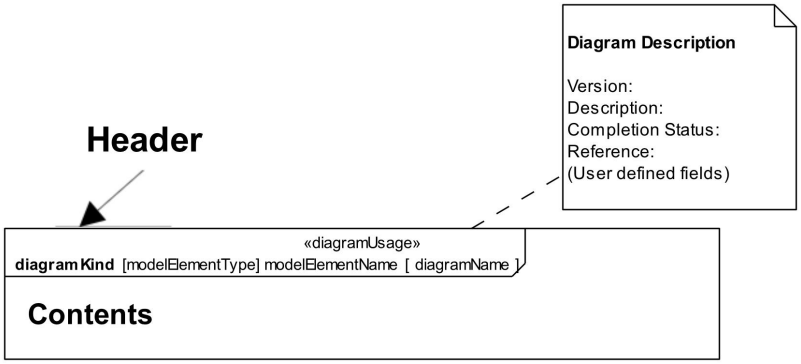


Figure A.2: Diagram Frame

The frame is a rectangle that is required for SysML diagrams (Note: the frame is optional in UML). The frame shall designate a model element that is the default namespace for the model elements enclosed in the frame. A qualified name for the model element within the frame shall be provided if it is not contained within default namespace associated with the frame. The following are some of the designated model elements associated with the different diagram kinds:

- Activity diagram - activity
- Block definition diagram - block, package, or constraint block
- Internal block diagram - block or constraint block
- Package diagram - package or model
- Parametric diagram - block or constraint block
- Requirement diagram - package or requirement
- Sequence diagram - interaction
- State machine diagram - state machine
- Use case diagram - package

The frame may include border elements associated with the designated model element, like:

- Ports for blocks
- Entry/exit points on statemachines
- Gates on interactions
- Parameters for activities
- Constraint parameters for constraint blocks.

The frame may sometimes be defined by the border of the diagram area provided by a tool.

The diagram contents area contains the graphical symbols. The diagram type and usage define the type of primary graphical symbols that are supported, e.g., a block definition diagram is a diagram where the primary symbols in the contents area are blocks and association symbols along with their adornments.

The heading name is a string contained in a name tag (rectangle with cutoff corner) in the upper leftmost corner of the rectangle, with the following syntax:

<diagramKind> [modelElementType] <modelElementName> [diagramName]

A space separates each of these entries. The diagramKind is bolded. The modelElementType and diagramName are in brackets. The heading name should always contain the diagram kind and model element name, and include the model element type and additional information to remove ambiguity. Ambiguity can occur if there is more than one model element type for a given diagram kind, or where there is more than one diagram for the same model element. If a model element type has a stereotype applied to the base model element, such as “modelLibrary” applied to a package or “controlOperator” applied to an activity, then either the stereotype name or the base model element may be used as the name for the model element type. In either case, the initial character of the name is shown in lower case. For a stereotype name, guillemet characters (« and ») are not shown. If more than one stereotype has been applied to the base model element, either the name of one of the applied stereotypes or a comma-separated list of any or all of the applied stereotype names may be shown. If a base model element name is used, this element is either a UML metaclass which SysML uses directly, such as package or activity, or a stereotype which SysML defines on a UML metaclass, such as block or view.

SysML diagram kinds should have the following names or (abbreviations) as part of the heading:

- Activity diagram (act)
- Block definition diagram (bdd)
- Internal block diagram (ibd)
- Package diagram (pkg)
- Parametric diagram (par)
- Requirement diagram (req)
- Sequence diagram (sd)
- State machine diagram (stm)
- Use case diagram (uc)

The diagram description can be defined by a comment attached to a diagram frame as indicated in Figure A.2 that includes version, description, references to related information, a completeness field that describes the extent to which the modeler asserts the diagram is complete, and other user defined fields. In addition, the diagram description may identify the view associated with the diagram, and the corresponding viewpoint that identifies the stakeholders and their concerns (refer to Model Elements clause). The diagram description can be made more explicit by the tool implementation.

SysML also introduces the concept of a diagram usage. This represents a unique usage of a particular diagram type, such as a context diagram as a usage of a block definition diagram, internal block diagram, or use case diagram. The diagram usage can be identified in the header above the diagramKind as «diagramUsage». An example of a diagram usage extension is shown in Figure A.3. For this example, the header in Figure A.2 would replace diagram kind with “uc” and «diagramUsage» with «ContextDiagram». Applying a stereotype approach to specify a diagram usage can allow a tool implementation to check that the diagram constraints defined by the stereotype are satisfied.

Diagram usage can be represented by creating stereotypes that extend SysMLDiagram (see Annex B).

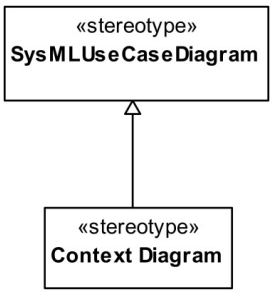


Figure A.3: Diagram Usages

Some typical diagram usages may include:

- Activity diagram usage with swim lanes - SwimLane Diagram.
- Block definition diagram usage for a block hierarchy - Block Hierarchy where block can be replaced by system, item, activity, etc.
- Use case diagram or internal block diagram to represent a Context Diagram.

A.2 Guidelines

The following provides some general guidelines that apply to all diagram types.

- Decomposition of a model element can be represented by the rake symbol. This does not always mean decomposition in a formal sense, but rather a reference to a more elaborated diagram of the model element that includes the rake symbol. This notation adds to the existing decomposition notations defined in UML (Composite state symbol for States that refer to StateMachines and rake symbol for CallBehaviorActions that refer to Activities). In SysML, the rake on a model element may also include the following:
    - Activity diagram - call behavior actions that can refer to another activity diagram.
    - Internal block diagram - parts that can refer to another internal block diagram.
    - Package diagram - package that can refer to another package diagrams.
    - Parametric diagram - constraint property that can refer to another parametric diagram.
    - Requirement diagram - requirement that can refer to another requirement diagram.
    - Sequence diagram - interaction fragments that can refer to another sequence diagram.
    - State machine diagram - state that can refer to another state machine diagram.
    - Use case diagram - use case can that may be realized by other behavior diagrams (activity, state, interactions).
- The primary mechanism for linking a text label outside of a symbol to the symbol is through proximity of the label to its symbol. This applies to ports, item flows, pins, etc.

- Page connectors (on-page connectors and off-page connectors) can be used to reduce the clutter on diagrams, but should be used sparingly since they are equivalent to go-to(s) in programming languages, and can lead to "spaghetti diagrams." Whenever practical, elaborate the model element designated by the frame instead of using a page connector. A page connector is depicted as a circle with a label inside (often a letter). The circle is shown at both ends of a line break and means that the two line end connect at the circle.
- When two lines cross, the crossing optionally may be shown with a small semicircular jog to indicate that the lines do not intersect (as in electrical circuit diagrams), as shown in Figure A.4.

Figure A.4: Optional Form of Line Crossing

- Diagram overlays are diagram elements that may be used on any diagram kind. An example of an overlay may be a geographic map to provide a spatial context for the symbols.
- SysML diagrams including the enhancements described in this sub clause are intended to conform to diagram definition and interchange standards to facilitate exchange of diagram and layout information.
- Tabular and matrix representation is an optional alternative notation that can be used in conjunction with the graphical symbols as long as the information is consistent with the underlying metamodel. Tabular and matrix representations are often used in systems engineering to represent detailed information and other views of the model such as interface definitions, requirements traceability, and allocation relationships between various types of model elements. They also can be convenient mechanisms to represent property values for selected properties, and basic relationships such as function and inputs/outputs in N2 charts. UML contains a tabular representation of a sequence diagram in an interaction matrix (refer to UML Annex with interaction matrix). The implementations of tabular and matrix representations are defined by the tool implementations and are not standardized in SysML at this time. However, tabular or matrix representations may be included in a frame with the heading designator «table» or «matrix» in bold.
- Graph and tree representations are also optional, alternative notations that can be used in conjunction with graphical symbols as long as the information is consistent with the underlying metamodel. These representations can be used for describing complex series of relationships that represent other views of the model. One example is the browser window in many tools that depicts a hierarchical view of the model. The implementations of graphs and trees are defined by the tool implementations and are not standardized in SysML at this time. However, graph and tree representations may be included in a frame with the heading designator «graph» or «tree» in bold.

Annex B: SysML Diagram Interchange

(informative)

B.1 Overview

This annex provides information regarding the exchange of SysML diagrams. It is an extension of the UML Diagram Interchange (DI) to support the graphical notation specific to SysML. A first part presents stereotypes that extend the UML DI. A second part presents modifications in the use of UML DI in SysML diagrams.

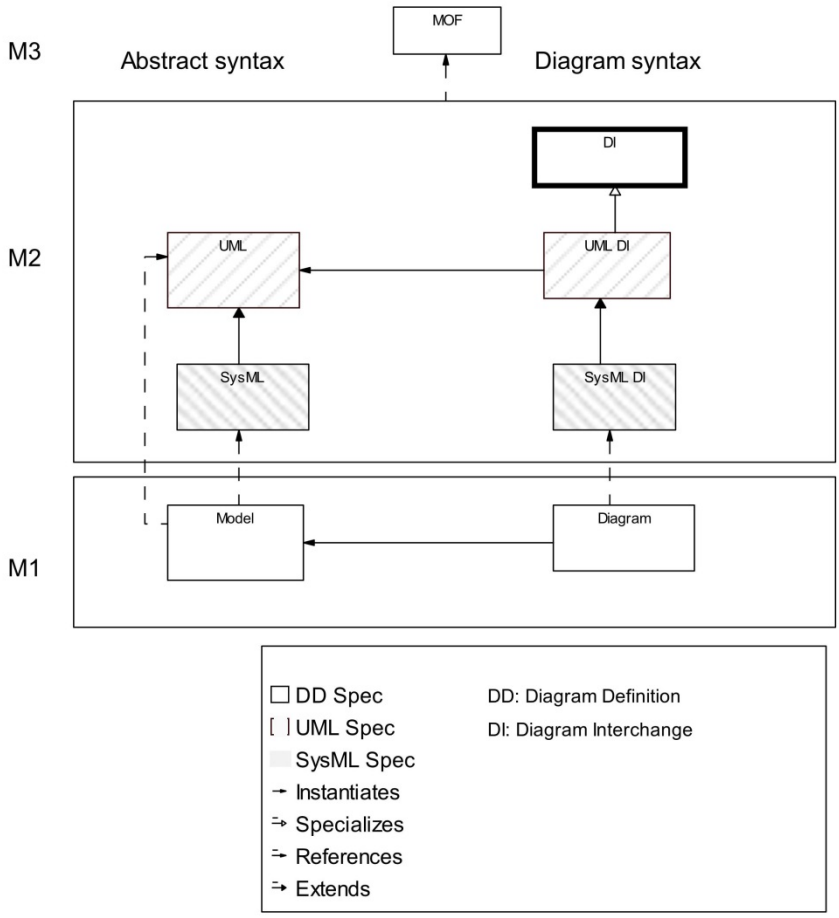


Figure B.1: SysML DI architecture

B.2 Stereotypes

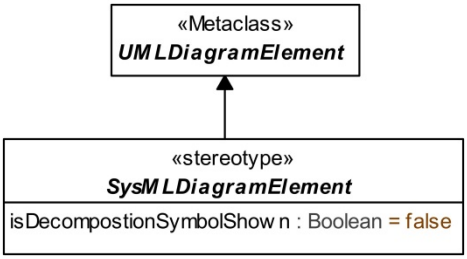


Figure B.2: Abstract Syntax Extension for SysMLDiagramElement

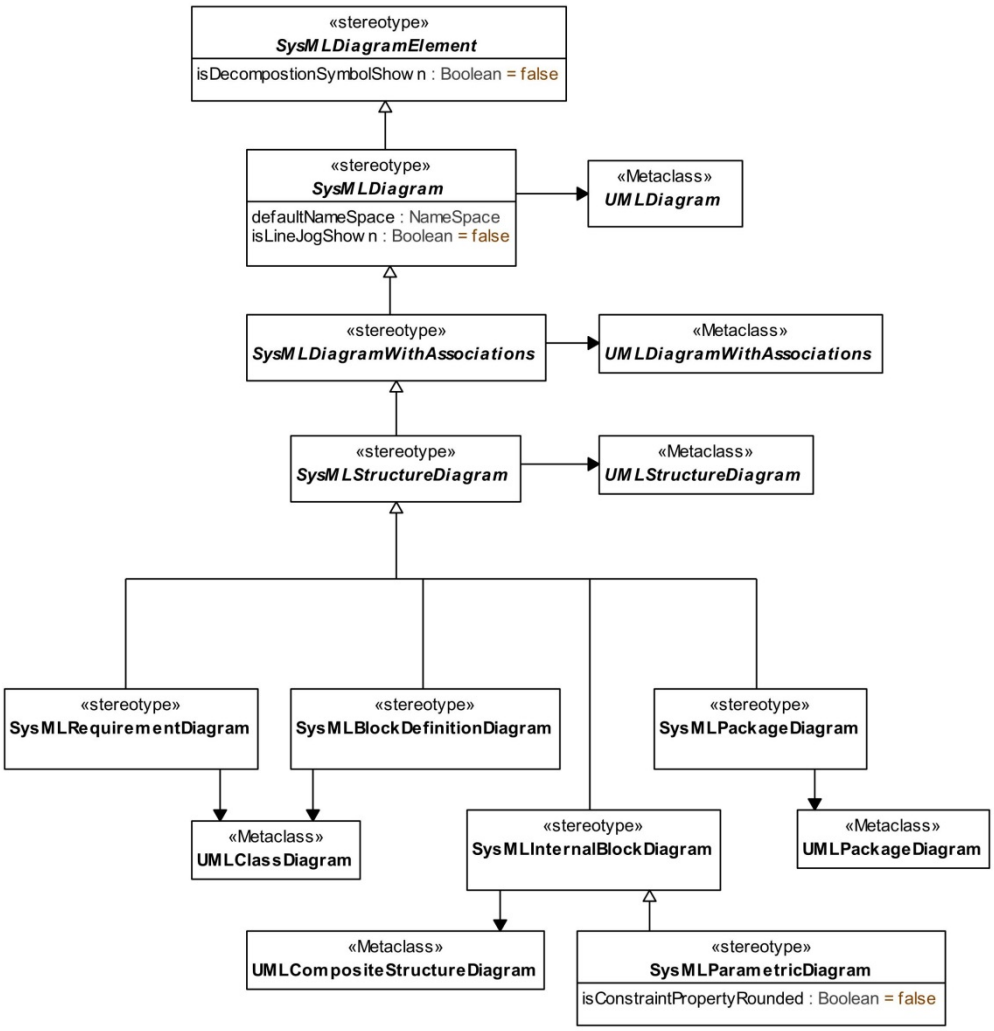


Figure B.3: Abstract syntax extensions for SysML diagrams (1)

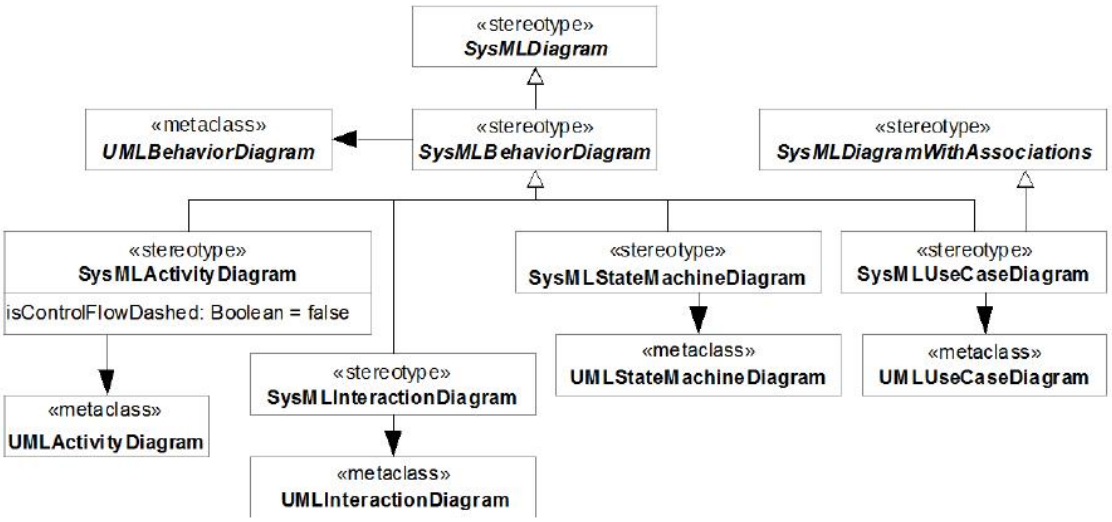


Figure B.4: Abstract syntax extensions for SysML diagrams (2)

B.2.1 SysML Activity Diagram

Description

A SysMLActivityDiagram represents an activity diagram. It extends UMLActivityDiagram.

Attributes

- isControlFlowDashed : Boolean [1] = false
  Specifies whether the control flows in the activity diagram are dashed (isControlFlowDashed=true) or not (isControlFlowDashed=false).

Constraints

[1] A SysMLActivityDiagram shall have as a defaultNamespace an Activity.

[2] SysMLActivityDiagram shall only be applied to a UMLActivityDiagram.The principal of an applied AdjunctProperty shall be a Connector, CallAction, ObjectNode, Variable, Parameter, submachine State, or InteractionUse.

B.2.2 SysML Behavior Diagram

Description

SysMLBehaviorDiagram is an abstract stereotype for all SysML behavior diagrams. It extends UMLBehaviorDiagram.

Constraints

[1] SysMLBehaviorDiagram shall only be applied to a UMLBehaviorDiagram.

[2] SysMLActivityDiagram shall only be applied to a UMLActivityDiagram.The principal of an applied AdjunctProperty shall be a Connector, CallAction, ObjectNode, Variable, Parameter, submachine State, or InteractionUse.

B.2.3 SysMLBlockDefinitionDiagram

Description
A SysMLBlockDefinitionDiagram represents a block definition diagram. It extends UMLPackageDiagram.

Constraints
[1] A SysMLBlockDefinitionDiagram shall have as a defaultNamespace a Class with a Block stereotype or one of its specializations applied or a Package.

[2] SysMLBlockDefinitionDiagram shall only be applied to a UMLClassDiagram.

B.2.4 SysMLDiagram

Description
SysMLDiagram is an abstract stereotype for all SysML diagrams. It extends UMLDiagram.

Attributes

- defaultNamespace : Namespace [1]
  Specifies the default namespace of the SysML diagram.

- isLineJogShown : Boolean [1] = false
  Show semi-circular jogs in the stereotyped diagram when two lines are crossing (see Annex A).

Constraints

[1] A UMLDiagram stereotyped by a specialization of SysMLDiagram shall have isFrame=true.

[2] A UMLDiagram stereotyped by a specialization of SysMLDiagram shall have a heading.

[3] A SysMLDiagram that stereotypes a UMLDiagram with a modelElement shall have this modelElement as defaultNamespace.

[4] SysMLDiagram shall only be applied to a UMLDiagram.

B.2.5 SysMLDiagramElement

Description
SysMLDiagramElement is an abstract generalization of all the other SysML DI stereotypes.

Attributes

- isDecompositionSymbolShown : Boolean [1]
  Display a decomposition symbol in a diagram element to indicate the corresponding model element is decomposed in another diagram. Diagram elements that may have a decomposition symbol are listed in Annex A.

B.2.6 SysMLDiagramWithAssociations

Description
SysMLDiagramWithAssociations is an abstract stereotype for all SysML diagrams with associations. It extends UMLDiagramWithAssociations.

Constraints
[1] A UMLDiagramWithAssociations stereotyped by a specialization of SysMLDiagramWithAssociations shall have isAssociationDotShown=false.

[2] A UMLDiagramWithAssociations stereotyped by a specialization of SysMLDiagramWithAssociations shall have navigabilityNotation=oneWay.

[3] A UMLDiagramWithAssociations stereotyped by a specialization of SysMLDiagramWithAssociations shall have nonNavigabilityNotation=never.

[4] SysMLDiagramWithAssociations shall only be applied to a UMLDiagramWithAssociations.

B.2.7 SysMLInteractionDiagram

Description
A SysMLInteractionDiagram represents an interaction diagram. It extends UMLInteractionDiagram.

Constraints
[1] A SysMLInteractionDiagram shall have as a defaultNamespace an Interaction.

[2] A UMLInteractionDiagram stereotyped by SysMLInteractionDiagram shall have kind=sequence.

[3] SysMLInteractionDiagram shall only be applied to a UMLInteractionDiagram.

B.2.8 SysMLInternalBlockDiagram

Description
A SysMLInternalBlockDiagram represents an internal block diagram. It extends UMLCompositeStructureDiagram.

Constraints
[1] A SysMLInternalBlockDiagram shall have as a defaultNamespace a Class with a Block stereotype or one of its specializations applied.

[2] SysMLInternalBlockDiagram shall only be applied to a UMLCompositeStructureDiagram.

B.2.9 SysMLPackageDiagram

Description
A SysMLPackageDiagram represents a package diagram. It extends UMLPackageDiagram.

Constraints
[1] A SysMLPackageDiagram shall have as a defaultNamespace a Package.

[2] SysMLPackageDiagram shall only be applied to a UMLPackageDiagram.

B.2.10 SysMLParametricDiagram

Description
A SysMLParametricDiagram represents a parametric diagram. It is a specialization of SysMLInternalBlockDiagram.

Attributes

- isConstraintPropertyRounded: Boolean = false
  Specifies whether the constraint properties in the parametric diagram have rounded corners (isConstraintPropertyRounded=true) or not (isConstraintPropertyRounded=false).

Constraints
[1] A SysMLParametricDiagram shall have as a defaultNamespace a Class with a Block stereotype or one of its specializations applied.

[2] SysMLParametricDiagram shall only be applied to a UMLCompositeStructureDiagram.

B.2.11 SysMLRequirementDiagram

Description
A SysMLRequirementDiagram represents a requirement diagram. It is based on the UML class diagram.

Constraints
[1] A SysMLRequirementDiagram shall have as a defaultNamespace a Package or a Class with a Requirement stereotype or one of its specializations applied.

[2] SysMLRequirementDiagram shall only be applied to a UMLClassDiagram.

B.2.12 SysMLStateMachineDiagram

Description
A SysMLStateMachineDiagram represents a state machine diagram. It extends UMLStateMachineDiagram.

Constraints
[1] A SysMLStateMachineDiagram shall have as a defaultNamespace a StateMachine.

[2] SysMLStateMachineDiagram shall only be applied to a UMLStateMachineDiagram.

B.2.13 SysMLUseCaseDiagram

Description
A SysMLUseCaseDiagram represents a use case diagram. It extends UMLUseCaseDiagram.

Constraints

[1] A SysMLUseCaseDiagram shall have as a defaultNamespace a Package.

[2] SysMLUseCaseDiagram shall only be applied to a UMLUseCaseDiagram.

B.3 SysML DI Usage Notes

This clause provides additional notes on how the SysML notation is modeled.

A UMLEdge with a Connector as modelElement may be the source or the target of a UMLEdge with no modelElement. The target or the source of the latter UMLEdge is a UMLShape with a Property stereotyped by ConnectorProperty or one of its specializations as modelElement. This UMLEdge is rendered as a dotted line.

Property names with property-specific types (in parentheses) are modeled with UMLTypedElementLabels.

UMLCompartmentableShapes that have a modelElement stereotyped by Allocated or one of its specializations may have a compartment titled “allocatedFrom” and a compartment titled “allocatedTo.” These compartments contain UMLLabels with modelElements that are the values of the allocatedFrom and allocatedTo properties, respectively, of the Allocated stereotype.

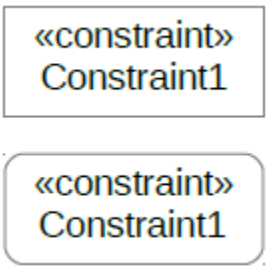
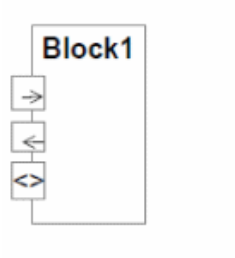
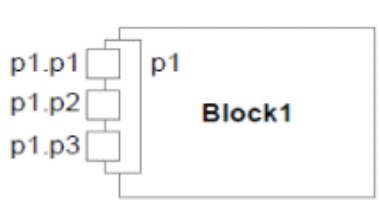
A UMLShape with a modelElement stereotyped by Allocated or one of its specializations may be the source or the target of a UMLEdge with no modelElements. The target or the source of this UMLEdge is a UMLShape with no modelElement. This UMLShape may contain UMLLabels with text “allocatedFrom” and “allocatedTo,” each being followed by UMLLabels with modelElements that are the values of the allocatedFrom properties of the Allocated stereotype or the values of the allocatedTo properties, respectively, of the Allocated stereotype.

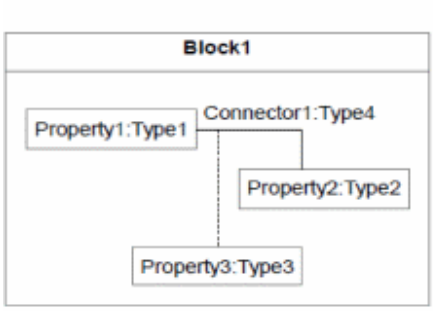
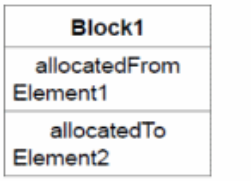
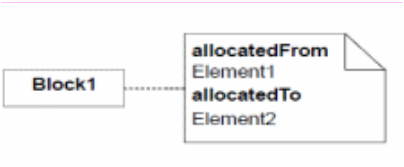
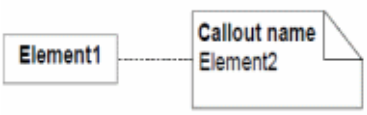
SysML callout notation (MasterCallout, DeriveCallout, SatisfyCallout, VerifyCallout, RefineCallout, TraceCallout) can be modeled by a UMLShape with no modelElement. This UMLShape contains a UMLLabel with text specified by the callout notation, followed by a UMLLabel with modelElement that is the element with text shown by the callout notation.

B.4 SysML Notation and DI Representation

This sub clause summarizes Annex B by showing how SysML-specific notations shall be modeled using UML and SysML UML DI. It does not cover all of Annex B or all notations in previous Clauses. The left column shows an example of SysML notation. The middle column shows UML DI and SysML DI elements corresponding to the notation. These elements are presented in a containment hierarchy. Elements with the same container are ordered according to the notation shown in the left column, read from left to right, top to bottom. For each element, the type of diagram element is given, followed by the type of modelElement and sometimes other constraints that apply to the diagram element, put between parentheses. The type of modelElement is followed by a '+' when multiple modelElements of this type can be assigned to one diagram element. A '+' sign between a metaclass and a stereotype corresponds to an element that instantiates the metaclass and that has the stereotype applied. The right column references “Notation” clauses and figures where the notation is defined.

Table B.1: SysML Diagram Elements

Notation	Diagram Elements	Ref.
	UMLEdge (ControlFlow, isControlFlowDashed=false) UMLEdge+SysMLControlFlowEdge (ControlFlow, isControlFlowDashed=true)	11.3.1.3.1
	UMLClassifierShape (Property+ConstraintProperty, isConstraintPropertyRounded =false) - UMLLabel (Stereotype) - UMLTypedElementLabel (Property) UMLClassifierShape (Property+ConstraintProperty is ConstraintPropertyRounded=true) - UMLLabel (Stereotype) - UMLTypedElementLabel (Property)	10.3.1.2.1
	UMLClassifierShape (Class+Block) - UMLNameLabel (Class) - UMLShape+SysMLPort (Port, in flows, isIcon=true) - UMLShape+SysMLPort (Port, out flows, isIcon=true) - UMLShape+SysMLPort (Port, inout flows, isIcon=true)	9.3.1.6
	UMLClassifierShape (Class+Block) - UMLNameLabel (Class) - UMLShape (Port) - UMLNameLabel (Port - UMLShape (Port - UMLNameLabel (Port) - UMLShape (Port) - UMLNameLabel (Port) - UMLShape (Port) - UMLNameLabel (Port)	9.3.1.6

Notation	Diagram Elements	Ref.
	UMLClassifierShape (Class) - UMLNameLabel (Class) - UMLCompartment --- UMLShape (Property) ----- UMLTypedElementLabel (Property) --- UMLEdge (Connector) ----- UMLTypedElementLabel (Property) --- UMLShape (Property) ----- UMLTypedElementLabel (Property) --- UMLEdge --- UMLShape (Property) ----- UMLTypedElementLabel	8.3.2.3
	UMLClassifierShape (Class) - UMLNameLabel (Class) - UMLCompartment --- UMLLabel --- UMLLabel (Element) - UMLCompartment --- UMLLabel --- UMLLabel (Element)	15.3.1.3
	UMLClassifierShape (Class) - UMLNameLabel (Class) UMLEdge UMLShape - UMLLabel - UMLLabel (Element) - UMLLabel - UMLLabel (Element)	15.3.1.4
	UMLShape (Element) - UMLNameLabel (Element) UMLEdge UMLShape - UMLLabel - UMLLabel (Element)	16.3.1.3

Annex C: Deprecated Elements and Migration

(Informative)

C.1 Overview

This annex

- Defines SysML elements that are deprecated, but included for backward compatibility (see Subannexes C.1.1 and C.1.2).
- Provides guidlines for migrating elements to this version of SysML that are deprecated (see above) or that changed significantly between versions of SysML (see Subannexes C.5 through C.7).

C.1.1 Flow Ports

Flow Port and Flow Specification are deprecated in this version of SysML and are defined for backward compatibility. This annex contains the definition of these concepts as they are defined by SysML 1.2. In addition it provides some guidelines on how to convert FlowPort to ports in this version of SysML.

A flow port specifies the input and output items that may flow between a block and its environment. Flow ports are interaction points through which data, material, or energy can enter or leave the owning block. The specification of what can flow is achieved by typing the flow port with a specification of things that flow. This can include typing an atomic flow port with a single type representing the items that flow in or out, or typing a nonatomic flow port with a flow specification which lists multiple items that flow. A block representing an automatic transmission in a car could have an atomic flow port that specifies “Torque” as an input and another atomic flow port that specifies “Torque” as an output. A more complex flow port could specify a set of signals and/or properties that flow in and out of the flow port. In general, flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions. Flow ports exten UML 2 ports.

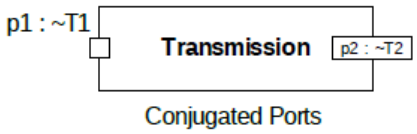
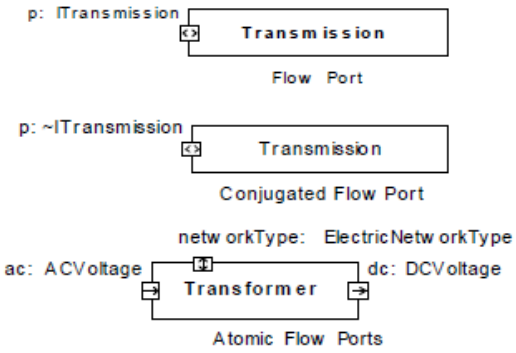
C.1.2 Conjugated Ports

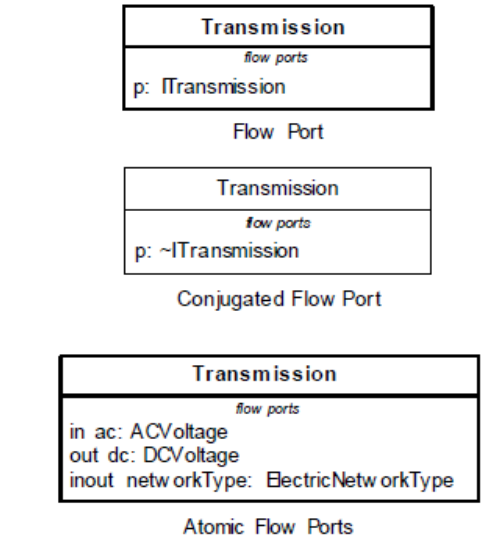
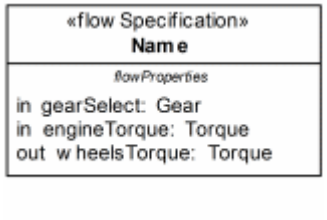
UMLs conjugated ports (UML::Port:: isConjugated) are deprecated in this version of SysML and included for backward compatibility. This annex contains the description of port conjugation in SysML 1.5. In addition it provides guidelines on how to convert conjugated ports to ports in this version of SysML.

C.2 Diagram Elements

C.2.1 Block Definition Diagram

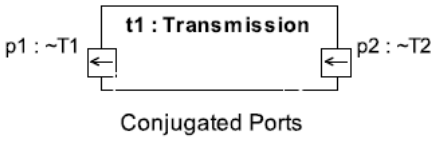
Table C-1: Graphical nodes defined in block definition diagrams

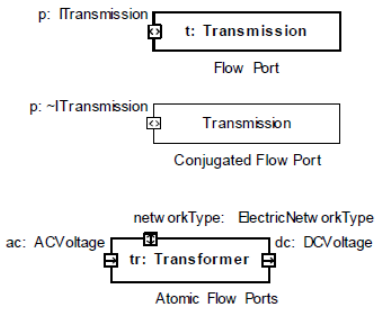
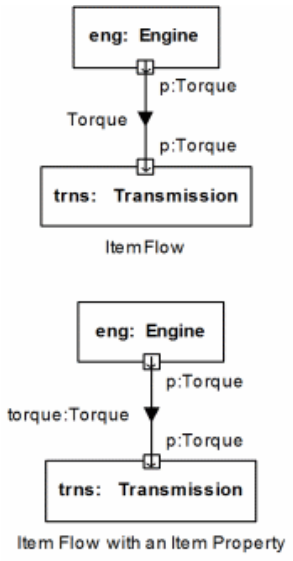
Node Name	Concrete Syntax	Abstract Syntax Reference
Port		**UML4SysML::Port**
FlowPort		**SysML::PortsAndFlows::FlowPort**

Node Name	Concrete Syntax	Abstract Syntax Reference
FlowPort (Compartment Notation)		**SysML::PortsAndFlows::FlowPort**
FlowSpecification		**SysML::PortsAndFlows:: FlowSpecification**

C.2.2 Internal Block Diagram

Table C-2: Graphical nodes defined in internal block diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Port		**UML4SysML::Port**

Node Name	Concrete Syntax	Abstract Syntax Reference
FlowPort		**SysML::PortsAndFlows::FlowPort**
ItemFlow		**SysML::PortsAndFlows::ItemFlow**

C.3 UML Extensions

C.3.1 Diagram Extensions

C.3.1.1 Conjucated Ports

Conjugated ports have UMLs Port::isConjugated property equal to true. Arrows in port rectangles indicated flow property direction are reversed in conjugated ports. Conjugated ports in conjugated ports (nested conjugated ports) behave as if they were not conjugated. Full ports also cannot be conjugated, because their types can have behaviors and can be reused on non-conjugated ports. This would require the same behaviors to use the directed features and flow properties in opposite directions at the same time.

The meaning of DirectedFeature::featureDirection property is reversed for conjugated ports. On conjugated ports, directed features with a feature direction "provided" are required and those with a feature direction "required" are provided. Port conjugation has no impact on "providedrequired" directed features. The meanings of the "required" and "provided" literals in FeatureDirection are switched for conjugated ports. In these cases the actual use is in the opposite direction than the one specified by the enumeration literal.

The meaning of FlowProperty::direction is reversed for conjugated ports. On conjugated ports, flow properties with direction "in" are out flow properties and those with direction "out" are in flow properties. Port conjugation has no impact on "inout" flow properties. The meanings of the "in" and "out" literals in FlowDirection are switched for conjugated ports. In these cases the actual flow direction is in the opposite direction than the one specified by the enumeration literal.

C.3.1.2 FlowPort

A FlowPorts is an interaction point through which input and/or output of items such as data, material, or energy may flow. The notation of flow port is a square on the boundary of the owning block or its usage. The label of the flow port is in the format portName: portType. Atomic flow ports have an arrow inside them indicating the direction of the port with respect to the owning Block. A nonatomic flow port has two open arrow heads facing away from each other (i.e., < >). The fill color of the square is white and the line and text colors are black.

In addition, flow ports can be listed in a special compartment labeled “flow ports.” The format of each line is:

in | out | inout portName:portType [{conjugated}]

C.3.1.3 FlowSpecification

A FlowSpecification specifies inputs and outputs as a set of flow properties. It has a “flowProperties” compartment that lists the flow properties.

C.3.2 Stereotypes

C.3.2.1 Package PortsAndFlows

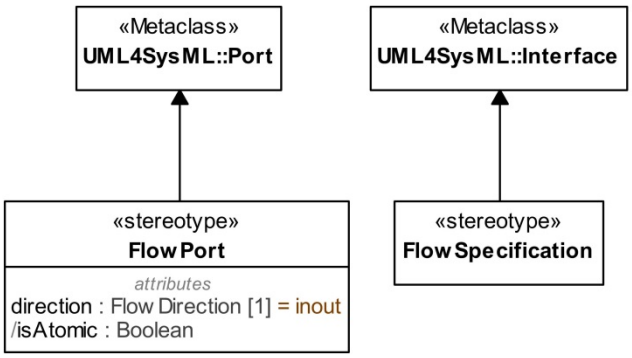


Figure C.1: Deprecated Stereotypes

C.3.2.2 FlowPort

Description

A FlowPort is an interaction point through which input and/or output of items such as data, material, or energy may flow. This enables the owning block to declare which items it may exchange with its environment and the interaction points through which the exchange is made.

We distinguish between atomic flow port and a nonatomic flow port. Atomic flow ports relay items that are classified by a single Block, ValueType, or Signal classifier. A nonatomic flow port relays items of several types as specified by a FlowSpecification.

The distinction between atomic and nonatomic flow ports is made according to the flow port's type: If a flow port is typed by a flow specification, then it is nonatomic; if a flow port is typed by a Block, ValueType, or Signal classifier, then it is atomic.

Flow ports and associated flow specifications define "what can flow" between the block and its environment, whereas item flows specify "what does flow" in a specific usage context.

Flow ports relay items to their owning block or to a connector that connects them with their owner's internal parts (internal connector).

The isBehavior attribute inherited from UML port is interpreted in the following way: if isBehavior is set to true, then the items are relayed to/from the owning block. More specifically, every flow property within the flow port is bound to a property owned by the port's owning block or to a parameter of its behavior. If isBehavior is set to false, then the flow port shall be connected to an internal connector, which in turn related the items via the port. The need for isBehavior is mainly to allow specification of internal parts relaying items to their containing part via flow ports.

The isConjugated attribute inherited from the UML Port metaclass is interpreted as follows: It indicates if the flows of items of a nonatomic flow port maintain the directions specified in the flow specification or if the direction of every flow property specified in the flow specification is reversed (IN becomes OUT and vice versa). If set to True, then all the directions of the flow properties specified by the flow specification that types a nonatomic flow port are relayed in the opposite direction (i.e., an "in" flow property is treated as an "out" flow property by the flow port and vice-versa). By default, the value is False. This attribute applies only to nonatomic flow ports since atomic flow ports have a direction attribute signifying the direction of the flow.

In case of flow properties or atomic flow ports of type Signal, inbound properties or atomic flow port are mapped to a Reception of the signal type (or a subtype) of the flow property's type. Outbound flow properties only declare the ability of the flow port to relay the signal over external connectors attached to it and are not mapped to a property of the flow port's owning block.

C.3.2.3 Semantic Variation Points

The binding of the flow properties on the ports to behavior parameters and/or block properties is a semantic variation point. One approach is to perform name and type matching. Another approach is to explicitly use binding relationships between the ports properties and behavior parameters or block properties.

Attributes

- /isAtomic : Boolean (derived)

  This is a derived attribute (derived from the flow port's type). For a flow port typed by a flow specification the value of this attribute is False, otherwise the value is True.
- direction : FlowDirection

Indicates the direction in which an atomic flow port relays its items. If the direction is set to “in,” then the items are relayed from an external connector via the flow port into the flow port’s owner (or one of its parts). If the direction is set to “out,” then the items are relayed from the flow port’s owner, via the flow port, through an external connector attached to the flow port. If the direction is set to “inout,” then items can flow both ways. By default, the value is inout.

Constraints

[1] A FlowPort shall be typed by a FlowSpecification, Block, Signal, or ValueType.

[2] If the FlowPort is atomic (by its type), then isAtomic=True, the direction shall be specified (has a value), and isConjugated is not specified (has no value).

[3] If the FlowPort is nonatomic, and the FlowSpecification typing the port has flow properties with direction “in,” the FlowPort direction shall be “in” (or “out” if isConjugated=true). If the flow properties are all out, the FlowPort direction shall be out (or in if isConjugated=true). If flow properties are both in and out, the direction shall be inout.

[4] A FlowPort can be connected (via connectors) to one or more flow ports that have matching flow properties. The matching of flow properties shall be done in the following steps:

1.Type Matching: The type being sent shall be the same type or a subtype of the type being received.

2.Direction Matching: If the connector connects two parts that are external to one another, then the direction of the flow properties shall be opposite, or at least one of the ends should be inout. If the connector is internal the owner of one of the flow ports, then the direction shall be the same or at least one of the ends shall be inout.

3.Name Matching: In case there is type and direction match to several flow properties at the other end, the property that has the same name at the other end shall be selected. If there is no such property, then the connection is ambiguous (ill-formed).

[5] If a flow port is not connected to an internal part, then isBehavior shall be set to true.

The item flows specified as flowing on a connector between flow ports shall match the flow properties of the ports at each end of the connector: the source of the item flow should be the port that has an outbound/bidirectional flow property that matches the item flow’s type and the target of the item flow should be the port that has an inbound/bidirectional flow property that matches the type of the item flow.

If a flow port is connected to multiple external and/or internal connectors, then the items are propagated (broadcast) over all connectors that have matching properties at the other end.

C.3.2.4 FlowSpecification

Description

A FlowSpecification specifies inputs and outputs as a set of flow properties. A flow specification is used by flow ports to specify what items can flow via the port.

Constraints

[1] Flow specifications shall not own operations or receptions (they can only own FlowProperties).

[2] Every “ownedAttribute” of a FlowSpecification shall be a FlowProperty.

C.3.2.5 ItemFlow (deprecated compatibility rule)

ItemFlows are not deprecated, but when used with atomic flows ports, have a deprecated modification of item flow compatibility rules that treats types of source and target atomic ports as if they were types of flow properties on types of those ports.

C.4 Transitioning SysML 1.2 Flow Ports to SysML 1.3 Ports (informative)

To convert a SysML 1.2 flow port to ports in this version of SysML it is recommended to use the following guidelines:

1. Decide if the port should be converted to a proxy port, a full port, or an unstereotyped port.
2. Based on the decision in step 1, create a block (for proxy ports, it shall be an interface block specifically).
3. If the original flow port is non-atomic:

   a. Copy all the flow properties owned by the flow port's type, a flow specification, to the block created in step 2 (meaning the flow properties will be owned by the newly created block).

   b. Replace the type of the port with the block created in step 2.

   c. Remove the flow port stereotype from the port.

   d. Based on the decision in step 1, apply the ProxyPort or FullPort stereotype, or do nothing if the decision is not to use either one.

   e. If the proxy stereotype is applied in step 3d, and there is a single connector from the port to a part, the BindingConnector may be applied to the connector.

   f. If the flow specification is not referenced by other model elements, delete it.
4. If the original flow port is atomic:

   a. On the block created in step 2, specify a flow property typed by the same type as the flow port and with the same direction as the original flow port.

   b. Do steps b to d from step 3 about non-atomic flow ports.

C.5 Transitioning SysML 1.3 Viewpoint and View to SysML 1.4 (informative)

Refactoring a view model build from the SysML 1.3 defined viewpoint, view, conforms, and the UML package import mechanism could be performed as follows:

- Conform
    - Replace v1.3 Conform with v1.4 Conform. The conform target in 1.3 becomes the general classifier in 1.4.
- View
    - Replace v1.3 View package with 1.4 View class
- Viewpoint
    - For each Stakeholder string, create a stakeholder with the string as the name
    - Update the stakeholder property on the new viewpoint with the created stakeholder
    - For each method string of the 1.3 viewpoint, create the operation «create»View() and append the string to the body of a comment that annotates the operation.
- Element and package import
    - Replace each package and element import with an expose relationship.

C.6 Transitioning SysML 1.3 Units and QuantityKinds to SysML 1.4 (informative)

Changing units and quantity kinds from SysML 1.3 to SysML 1.4 can be accomplished as follows, depending on the kind of element being changed:

- An InstanceSpecification stereotyped by SysML 1.3 Unit:
    - Unapply the SysML 1.3 Unit stereotype.
    - Classify the instance specification by SysML::Libraries::UnitAndQuantityKind::Unit.
    - Set the values of SysML 1.4 Unit properties (symbol, description, definitionURI) to the values of the Unit stereotype properties of the same name (symbol, description, definitionURI).
- An InstanceSpecification stereotyped by SysML 1.3 QuantityKind:
    - Unapply the SysML 1.3 QuantityKind stereotype.
    - Classifying the instance specification by SysML::Libraries::UnitAndQuantityKind::QuantityKind.

    - Set the values of SysML 1.4 QuantityKind properties (symbol, description, definitionURI) to the values of the QuantityKind stereotype properties of the same name (symbol, description, definitionURI).

- An InstanceSpecification classified by SysML 1.3 QUDV::Unit or one of its specializations:
    - If the instance specification has no value for the SysML 1.3 QUDV::Unit::name property, no further changes are needed.
    - If the instance specification has a value for the SysML 1.3 QUDV::Unit::name property and the instance specification has no name, then set its name to the value of the SysML 1.3 QUDV::Unit::name property.
    - If the instance specification has a value for the SysML 1.3 QUDV::Unit::name property and the instance specification has a name, then choose whether to keep the same name for the instance specification or use the value of the SysML 1.3 QUDV::Unit::name property.

- An InstanceSpecification classified SysML 1.3 QUDV::QuantityKind or one of its specializations:
    - If the instance specification has no value for the SysML 1.3 property QUDV::QuantityKind::name, then no further changes are needed.
    - If the instance specification has a value for the SysML 1.3 property QUDV::QuantityKind::name and the instance specification has no name, then set the name of the instance specification to the value of the SysML 1.3 QUDV::QuantityKind::name property.
    - If the instance specification has a value for the SysML 1.3 property QUDV::QuantityKind::name and the instance specification has a name, then choose whether to keep the same name for the instance specification or use the value of the SysML 1.3 QUDV::QuantityKind::name property.

- An InstanceSpecification An InstanceSpecification classified by SysML 1.3 QUDV::Scale. Each SysML 1.3 QUDV::ScaleValueDefinition becomes an EnumerationLiteral such that:
    - The numeric value of SysML 1.3 QUDV::ScaleValueDefinition::value becomes a specification of the corresponding EnumerationLiteral.
    - The string value of SysML 1.3 QUDV::ScaleValueDefinition::description becomes a comment on the corresponding EnumerationLiteral.

- Blocks defined as specializations of SysML 1.3 QUDV::Unit do not require changes in SysML 1.4.

- Blocks defined as specializations of SysML 1.3 QUDV::QuantityKind do not require changes in SysML 1.4 except for the following:
    - Blocks defined specializations of QUDV::SpecializedQuantityKind in SysML 1.3 become corresponding Blocks defined as specializations of QUDV::QuantityKind in SysML 1.4.
    - Usages of SysML 1.3 QUDV::SpecializedQuantityKind::general property become corresponding usages of QUDV::QuantityKind::general in SysML 1.4.

C.7 Transitioning SysML 1.5 conjugated port typed by InterfaceBlock to SysML 1.6 conjugated InterfaceBlock (informative)

Here are the migration rules from former versions of SysML in pseudo code, they can be easily automated:

```
For each port with isConjugated=true
do {
  assume t1 is the type the port
  if t1 is a kind of InterfaceBlock then
  {
    if t1.getConjugated() return an empty result then
    {
        create  a new InterfaceBlock t2 with the name of t1 prepended  by a tilde symbol (~)
        For each feature of t1
        do {
           create the exact same feature f' in t2
           if f' has the FlowProperty stereotype applied
           {
              if the direction of f' is "in" then
                 set f' direction to "out"
              else if direction of f' is "out" then
                 set f' direction to "in"
              else do nothing
           }
           else if f' has the DirectedFeature stereotype applied
           {
               if the direction of f' is "provided" then
                 set f' direction to "required"
              else if direction of f' is "required" then
                 set f' direction to "provided"
```

```
            else do nothing
          }
          else
         {
           apply the DirectedFeature stereotype to f'
           set f' direction to "required"
         }
       }

       For each owned rule r of t1
       do {
          create the exact same owned rule r' in t2
       }
       create a dependency from t2 to t1 with the Conjugation stereotype applied
    }
    set this port type to t2
    set this port isConjugated to false
  }
}
```

Annex D: Sample Problem

(Informative)

D.1 Purpose

The purpose of this annex is to illustrate how SysML can support the specification, analysis, and design of a system using some of the basic features of the language.

D.2 Scope

The scope of this example is to provide at least one diagram for each SysML diagram type. The intent is to select simplified fragments of the problem to illustrate how the diagrams can be applied, and to demonstrate some of the possible inter-relationships among the model elements in the different diagrams. The sample problem does not highlight all of the features of the language. The reader should refer to the individual clauses for more detailed features of the language. The diagrams selected for representing a particular aspect of the model, and the ordering of the diagrams are intended to be representative of applying a typical systems engineering process, but this will vary depending on the specific process and methodology that is used.

D.3 Problem Summary

The sample problem describes the use of SysML as it applies to the development of an automobile, in particular a Hybrid gas/electric powered Sport Utility Vehicle (SUV). This problem is interesting in that it has inherently conflicting requirements, viz. desire for fuel efficiency, but also desire for large cargo carrying capacity and off-road capability. Technical accuracy and the feasibility of the actual solution proposed were not high priorities. This sample problem focuses on design decisions surrounding the power subsystem of the hybrid SUV; the requirements, performance analyses, structure, and behavior.

This annex is structured to show each diagram in the context of how it might be used on such an example problem. The first sub clause shows SysML diagrams as they might be used to establish the system context; establishing system boundaries, and top level use cases. The next sub clause is provided to show how SysML diagrams can be used to analyze top level system behavior, using sequence diagrams and state machine diagrams. The following sub clause focuses on use of SysML diagrams for capturing and deriving requirements, using diagrams and tables. A sub clause is provided to illustrate how SysML is used to depict system structure, including block hierarchy and part relationships. The relationship of various system parameters, performance constraints, analyses, and timing diagrams are illustrated in the next sub clause. A sub clause is then dedicated to illustrating definition and depiction of interfaces and flows in a structural context. The final sub clause focuses on detailed behavior modeling, functional and flow allocation.

D.4 Diagrams

D.4.1 Package Overview (Structure of the Sample Model)

D.4.1.1 Package Diagram – Applying the SysML Profile

As shown in Figure D.1, the HSUVModel is a package that represents the user model. The SysML Profile shall be applied to this package in order to include stereotypes from the profile. The HSUVModel may also require model libraries, such as the SI Units Types model library. The model libraries shall be imported into the user model as indicated.

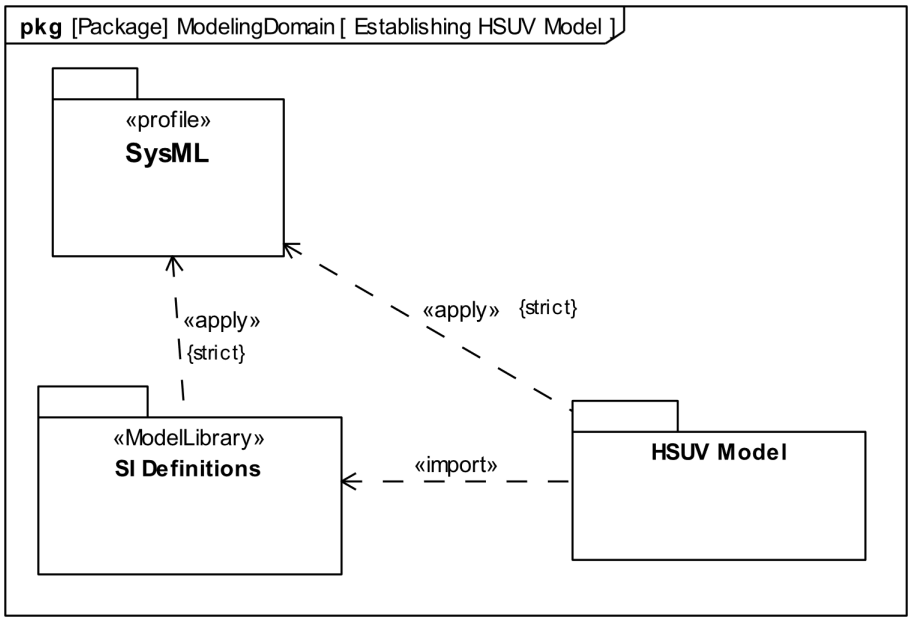


Figure D.1: Establishing the User Model by importing and applying SysML Profile & Model Library (Package Diagram)

Figure D.2 details the specification of units and valueTypes employed in this sample problem.

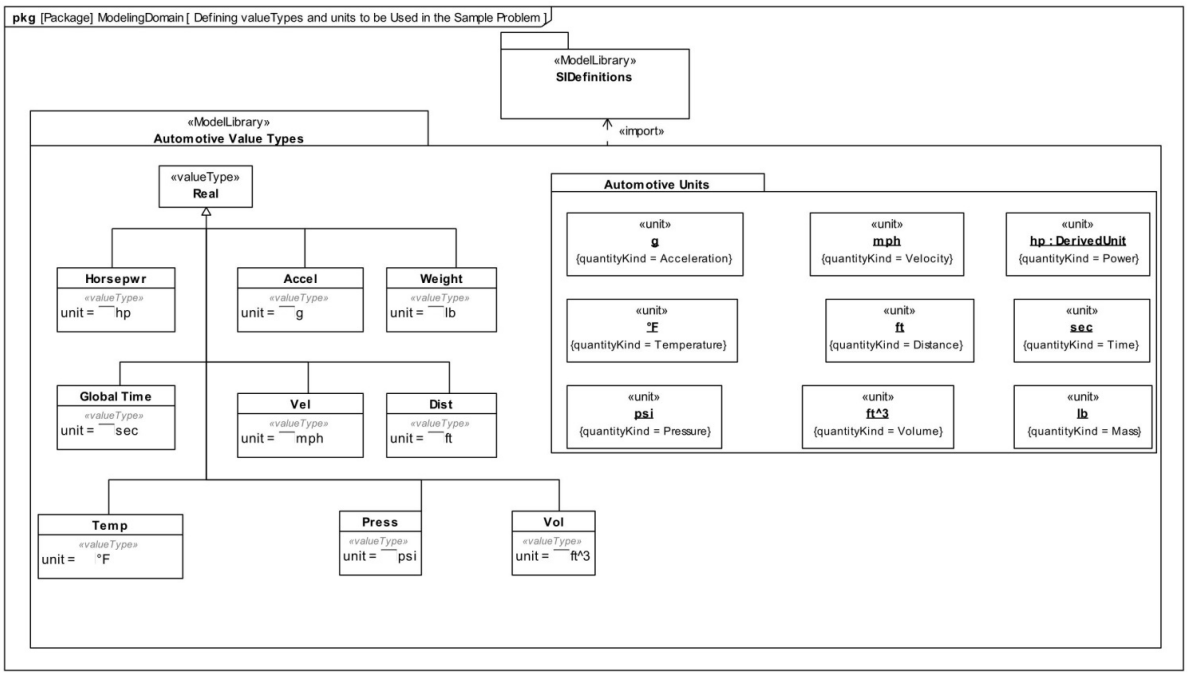


Figure D.2: Defining value Types and units to be used in the Sample Problem

D.4.1.2 Package Diagram – Showing Package Structure of the Model

The package diagram (Figure D.3) shows the structure of the model used to evaluate the sample problem. Model elements are contained in packages, and relationships between packages (or specific model elements) are shown on this diagram. The relationship between the views (OperationalView and PerformanceView) and the rest of the user model are explicitly expressed using the «import» relationship. Note that the «view» models contain no model elements of their own, and that changes to the model in other packages are automatically updated in the Operational and Performance Views.

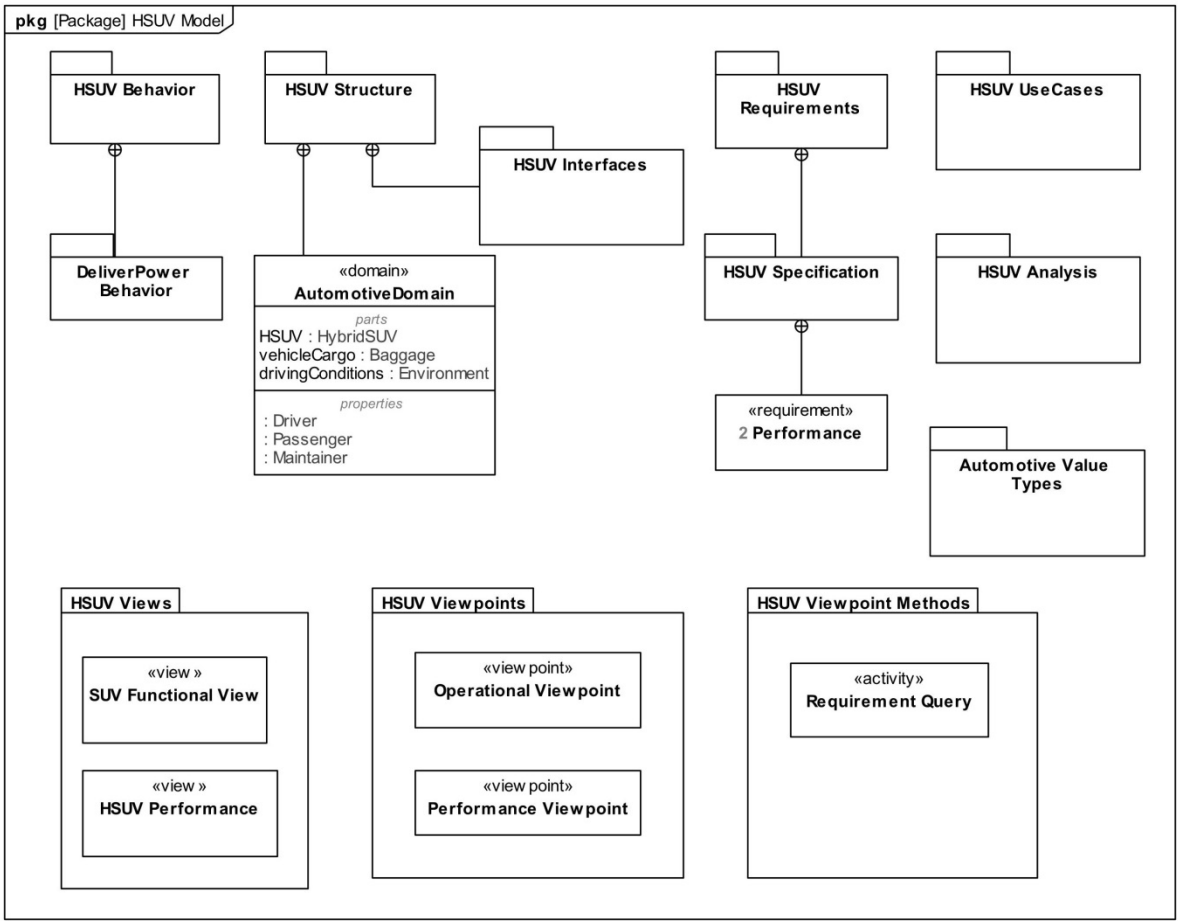


Figure D.3 Establishing Structure of the User Model using Packages and Views (Package Diagram)

D.4.2 Setting the Context (Boundaries and Use Cases)

D.4.2.1 Internal Block Diagram – Setting Context

The term "context diagram," in Figure D.4, refers to a user-defined usage of an internal block diagram, which depicts some of the top-level entities in the overall enterprise and their relationships. The diagram usage enables the modeler or

methodologist to specify a unique usage of a SysML diagram type using the extension mechanism described in Annex A, “Diagrams.” The entities are conceptual in nature during the initial phase of development, but will be refined as part of the development process. The «system» and «external» stereotypes are user defined, not specified in SysML, but help the modeler to identify the system of interest relative to its environment. Each model element depicted may include a graphical icon to help convey its intended meaning. The spatial relationship of the entities on the diagram sometimes conveys understanding as well, although this is not specifically captured in the semantics. Also, a background such as a map can be included to provide additional context. The associations among the classes may represent abstract conceptual relationships among the entities, which would be refined in subsequent diagrams. Note how the relationships in this diagram are also reflected in the Automotive Domain Model Block Definition Diagram, Figure D.15.

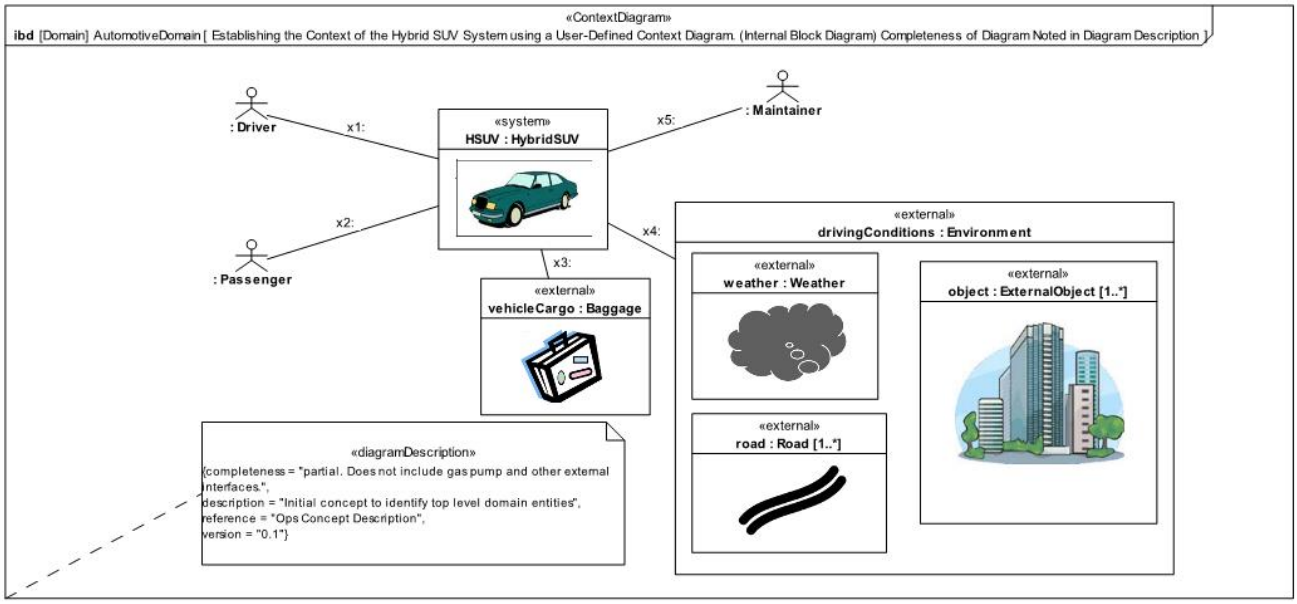


Figure D.4 Establishing the Context of the Hybrid SUV System using a User-Defined Context Diagram. (Internal Block Diagram) Completeness of Diagram Noted in Diagram Description

The use case diagram for “Drive Vehicle” in Figure D.5 depicts the drive vehicle usage of the vehicle system. The subject (HybridSUV) and the actors (Driver, Registered Owner, Maintainer, Insurance Company, DMV) interact to realize the use case.

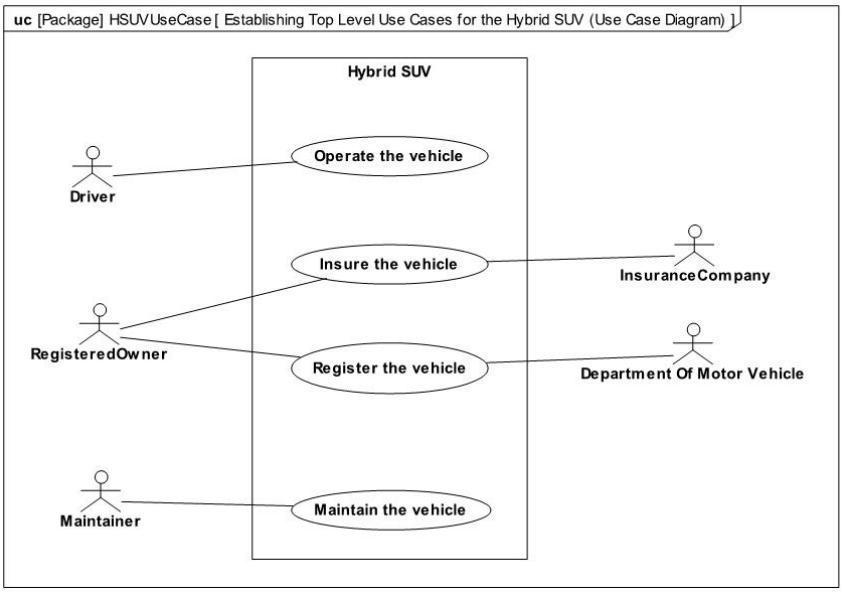


Figure D.5 Establishing Top Level Use Cases for the Hybrid SUV (Use Case Diagram)

D.4.2.2 Use Case Diagram – Operational Use Cases

Goal-level Use Cases associated with "Operate the Vehicle" are depicted in the following diagram. These use cases help flesh out the specific kind of goals associated with driving and parking the vehicle. Maintenance, registration, and insurance of the vehicle would be covered under a separate set of goal-oriented use cases.

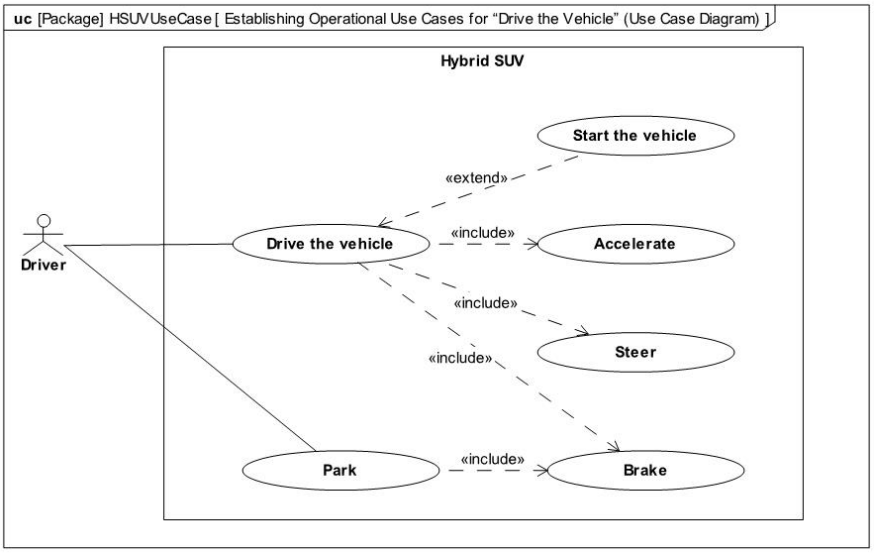


Figure D.6 Establishing Operational Use Cases for "Drive the Vehicle" (Use Case Diagram)

D.4.3 Elaborating Behavior (Sequence and State Machine Diagrams)

D.4.3.1 Sequence Diagram – Drive Black Box

Figure D.7 shows the interactions between driver and vehicle that are necessary for the "Drive the Vehicle" Use Case. This diagram represents the "DriveBlackBox" interaction, with is owned by the AutomotiveDomain block. "BlackBox" for the purpose of this example, refers to how the subject system (HybridSUV block) interacts only with outside elements, without revealing any interior detail.

The conditions for each alternative in the alt controlSpeed sub clause are expressed in OCL, and relate to the states of the HybridSUV block, as shown in Figure D.8.

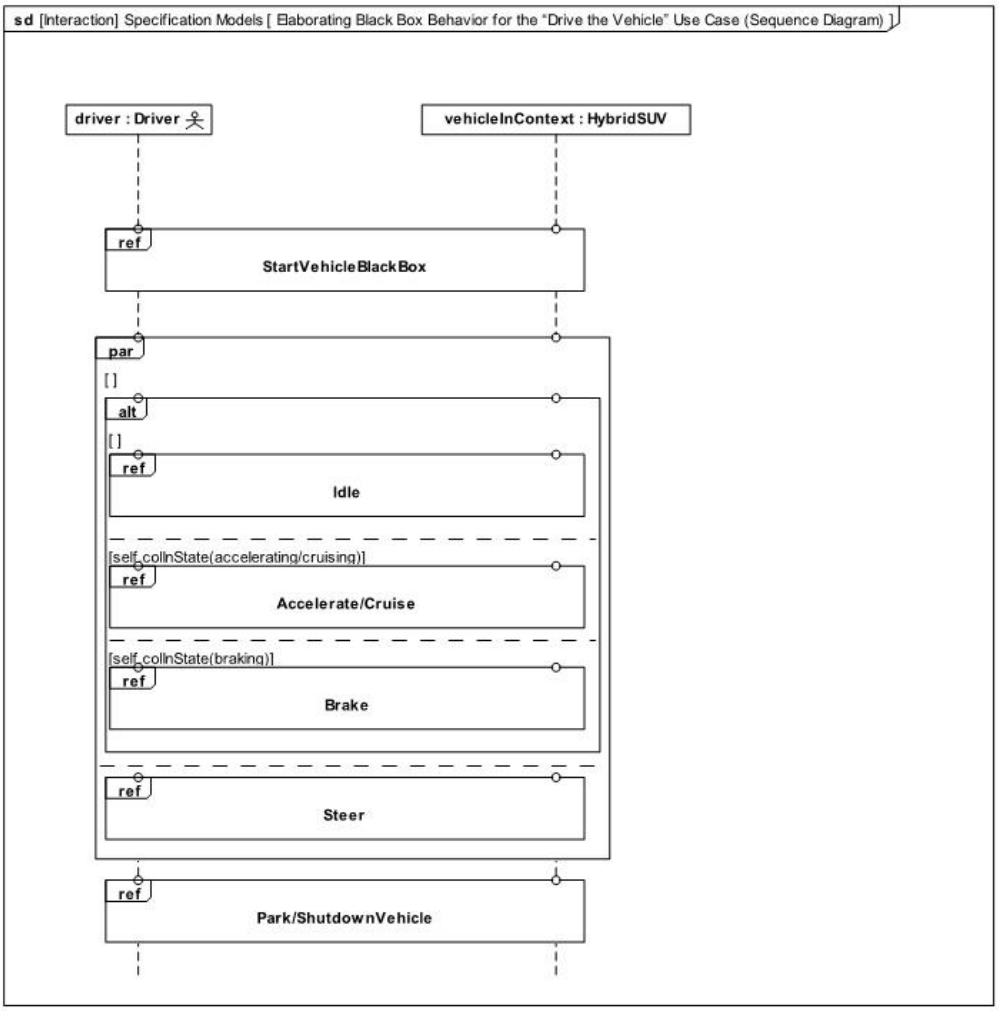


Figure D.7 Elaborating Black Box Behavior for the "Drive the Vehicle" Use Case (Sequence Diagram)

D.4.3.2 State Machine Diagram – HSUV Operational States

Figure D.8 depicts the operational states of the HSUV block, via a State Machine named "HSUVOperationalStates." Note that this state machine was developed in conjunction with the DriveBlackBox interaction in Figure D.7. Also note

that this state machine refines the requirement “PowerSourceManagment,” which will be elaborated in the requirements sub clause of this sample problem. This diagram expresses only the nominal states. Exception states, like “acceleratorFailure,” are not expressed on this diagram.

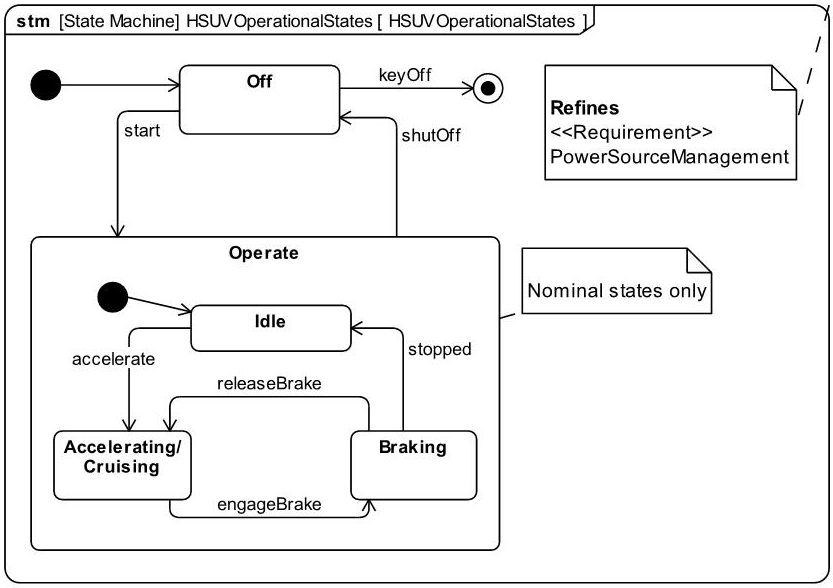


Figure D.8 Finite State Machine Associated with “Drive the Vehicle” (State Machine Diagram)

D.4.3.3 Sequence Diagram – Start Vehicle Black Box & White Box

Figure D.9 shows a “black box” interaction, but references “StartVehicleWhiteBox” (Figure D.10), which will decompose the lifelines within the context of the HybridSUV block.

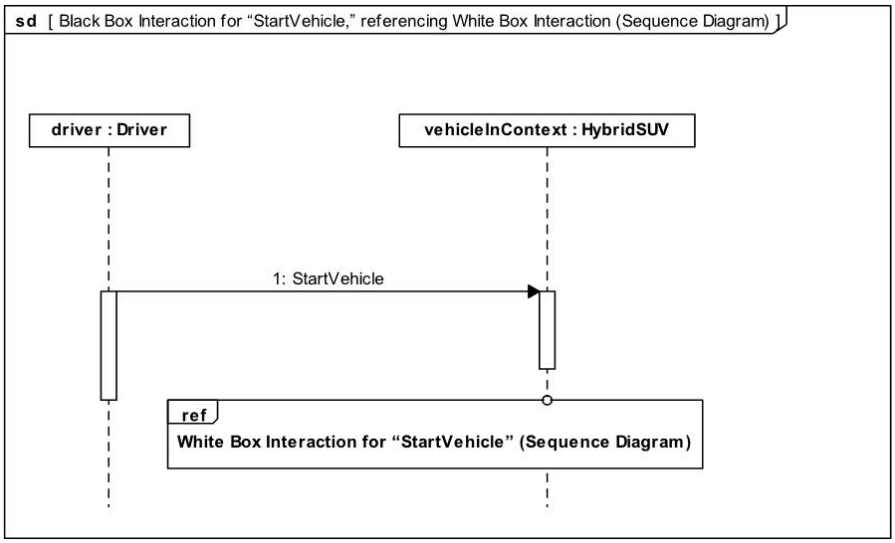


Figure D.9 Black Box Interaction for “StartVehicle,” referencing White Box Interaction (Sequence Diagram)

The lifelines on Figure D.10 (“whitebox” sequence diagram) need to come from the Power System decomposition. This now begins to consider parts contained in the HybridSUV block.

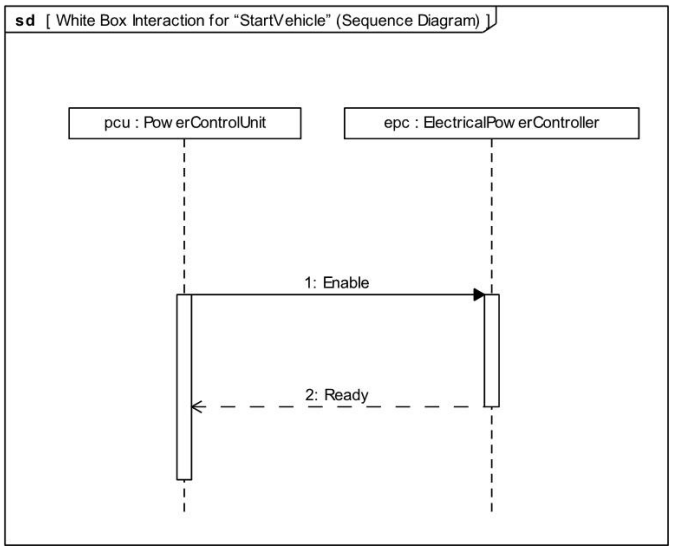


Figure D.10 White Box Interaction for “StartVehicle” (Sequence Diagram)

D.4.4 Establishing Requirements (Requirements Diagrams and Tables)

D.4.4.1 Requirement Hierarchy

The vehicle system specification contains many text based requirements. A few requirements are highlighted in Figure D.11, including the requirement for the vehicle to pass emissions standards, which is expanded for illustration purposes. The containment (cross hair) relationship, for purposes of this example, refers to the practice of decomposing a complex requirement into simpler, single requirements.

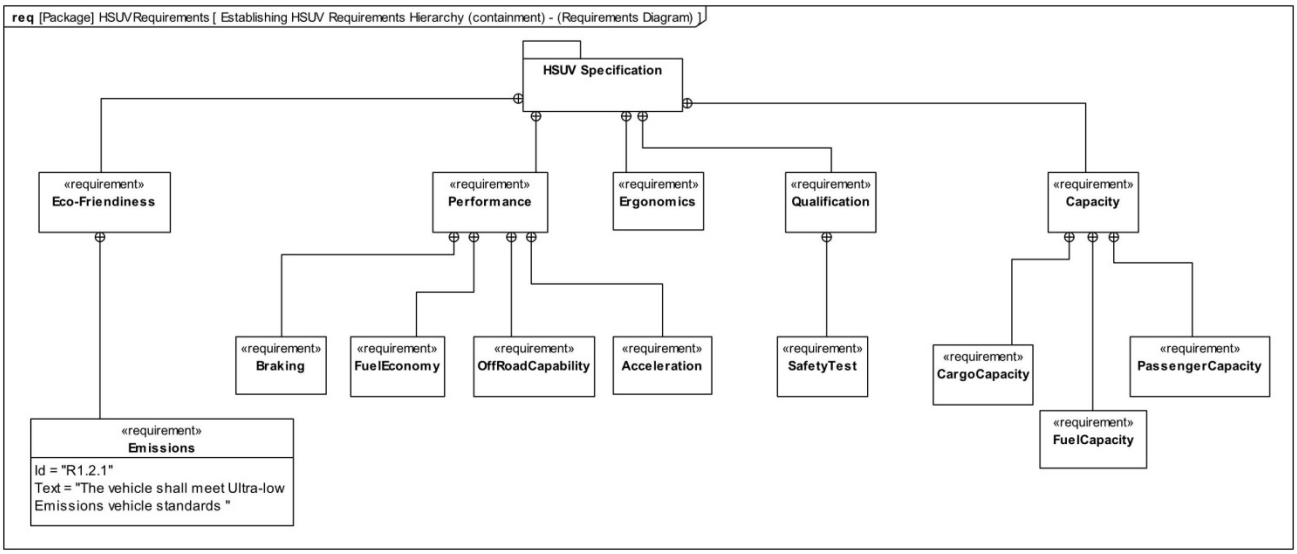


Figure D.11 Establishing HSUV Requirements Hierarchy (containment) – (Requirements Diagram)

D.4.4.2 Requirement Diagram – Derived Requirements

Figure D.12 shows a set of requirements derived from the lowest tier requirements in the HSUV specification. Derived requirements, for the purpose of this example, express the concepts of requirements in the HSUVSpecification in a manner that specifically relates them to the HSUV system. Various other model elements may be necessary to help develop a derived requirement, and these model element may be related by a «refinedBy» relationship. Note how PowerSourceManagement is “RefinedBy” the HSUVOperationalStates model (Figure D.8). Note also that rationale can be attached to the «deriveReqt» relationship. In this case, rationale is provided by a referenced document “Hybrid Design Guidance.”

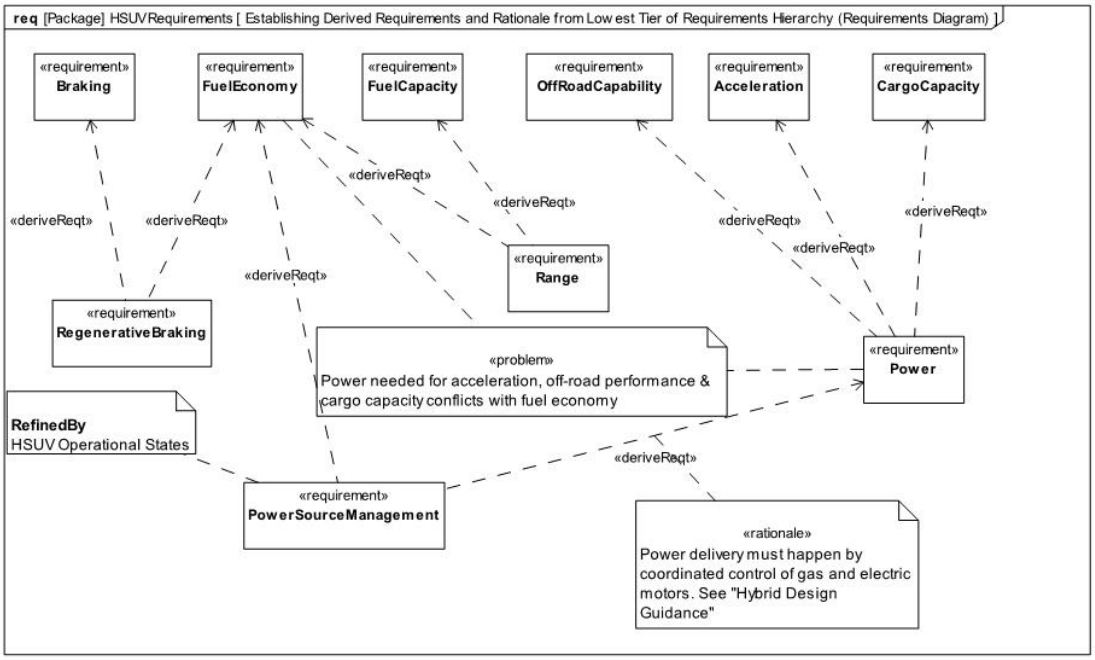


Figure D.12 Establishing Derived Requirements and Rationale from Lowest Tier of Requirements Hierarchy (Requirements Diagram)

D.4.4.3 Requirement Diagram – Acceleration Requirement Relationships

Figure D.13 focuses on the Acceleration requirement, and relates it to other requirements and model elements. The "refine" relation, introduced in Figure D.12, shows how the Acceleration requirement is refined by a similarly named use case. The Power requirement is satisfied by the PowerSubsystem, and a Max Acceleration test case verifies the Acceleration requirement.

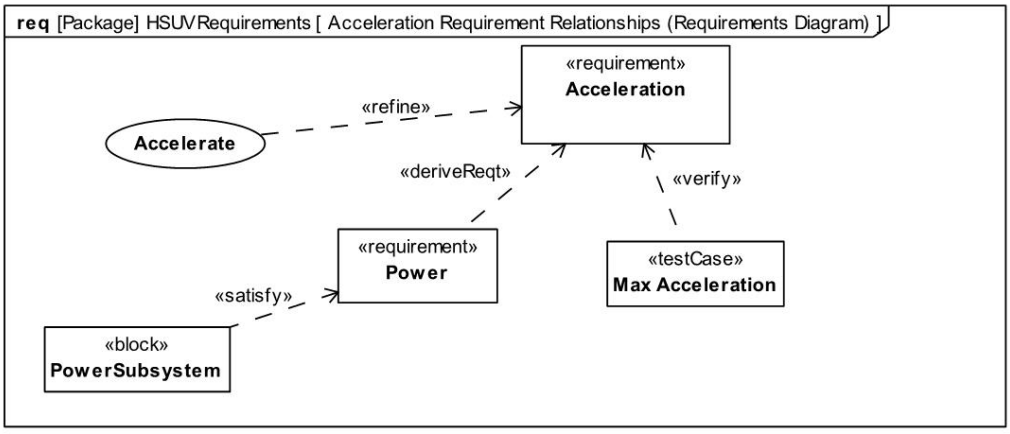


Figure D.13 Acceleration Requirement Relationships (Requirements Diagram)

D.4.4.4 Table – Requirements Table

Figure D.14 contains two diagrams that show requirement containment (decomposition), and requirements derivation in tabular form. This is a more compact representation than the requirements diagrams shown previously.

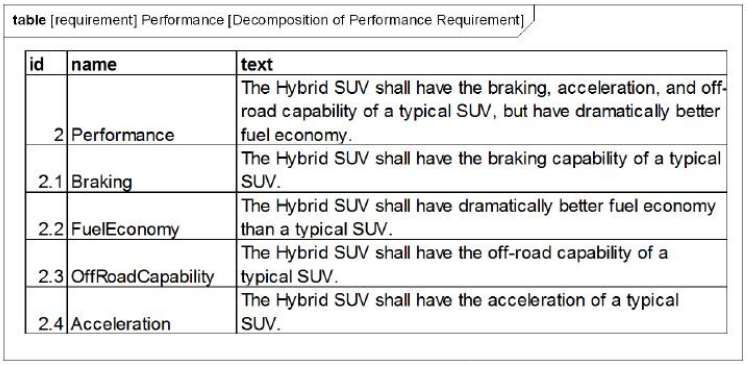


table [requirement] Performance [Decomposition of Performance Requirement]

id	name	text
2	Performance	The Hybrid SUV shall have the braking, acceleration, and off-road capability of a typical SUV, but have dramatically better fuel economy.
2.1	Braking	The Hybrid SUV shall have the braking capability of a typical SUV.
2.2	FuelEconomy	The Hybrid SUV shall have dramatically better fuel economy than a typical SUV.
2.3	OffRoadCapability	The Hybrid SUV shall have the off-road capability of a typical SUV.
2.4	Acceleration	The Hybrid SUV shall have the acceleration of a typical SUV.

table [requirement] Performance [Tree of Performance Requirements]

id	name	relation	id	name	relation	id	name
2.1	Braking	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.2	Range			
4.2	FuelCapacity	deriveReqt	d.2	Range			
2.3	OffRoadCapability	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
2.4	Acceleration	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
4.1	CargoCapacity	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement

Figure D.14 Requirements Relationships Expressed in Tabular Format (Table)

D.4.5 Breaking Down the Pieces (Block Definition Diagrams, Internal Block Diagrams)

D.4.5.1 Block Definition Diagram – Automotive Domain

Figure D.15 provides definition for the concepts previously shown in the context diagram. Note that the interactions DriveBlackBox and Stac4rtVehicleBlackBox (described in D.4.3 Elaborating Behavior (Sequence and State Machine Diagrams), are depicted as owned by the AutomotiveDomain block.

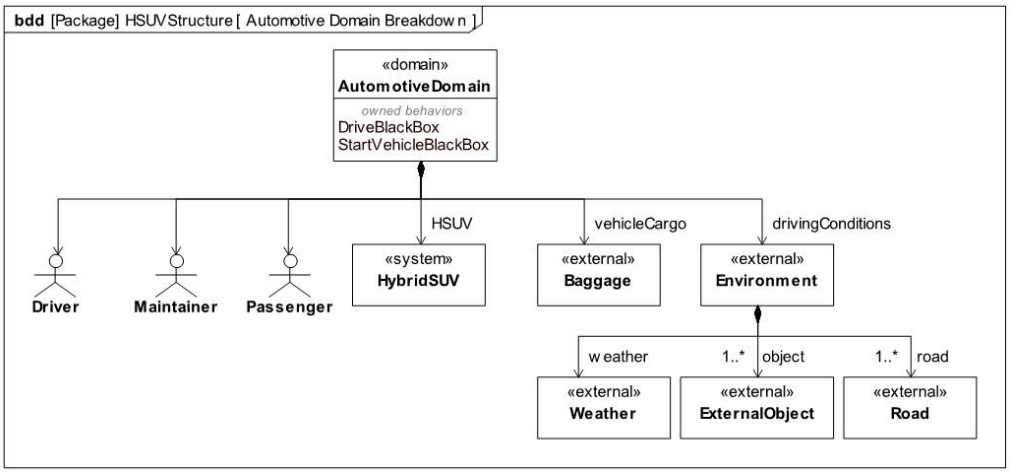


Figure D.15 Defining the Automotive Domain (compare with Figure D.4) – (Block Definition Diagram)

D.4.5.2 Block Definition Diagram – Hybrid SUV

Figure D.16 defines components of the HybridSUV block. Note that the BrakePedal and WheelHubAssembly are used by, but not contained in, the PowerSubsystem block.

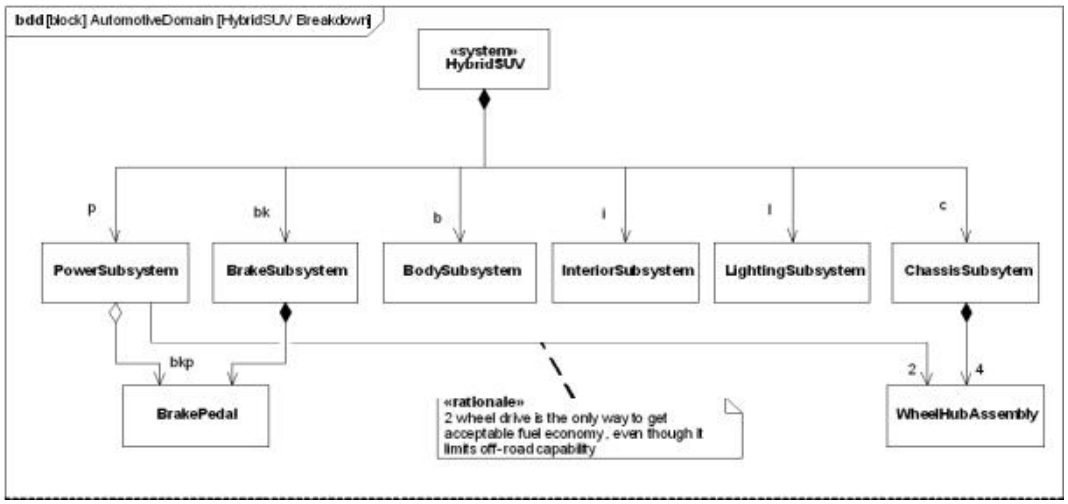


Figure D.16 Defining Structure of the Hybrid SUV System (Block Definition Diagram)

D.4.5.3 Internal Block Diagram – Hybrid SUV

Figure D.17 shows how the top level model elements in the above diagram are connected together in the HybridSUV block.

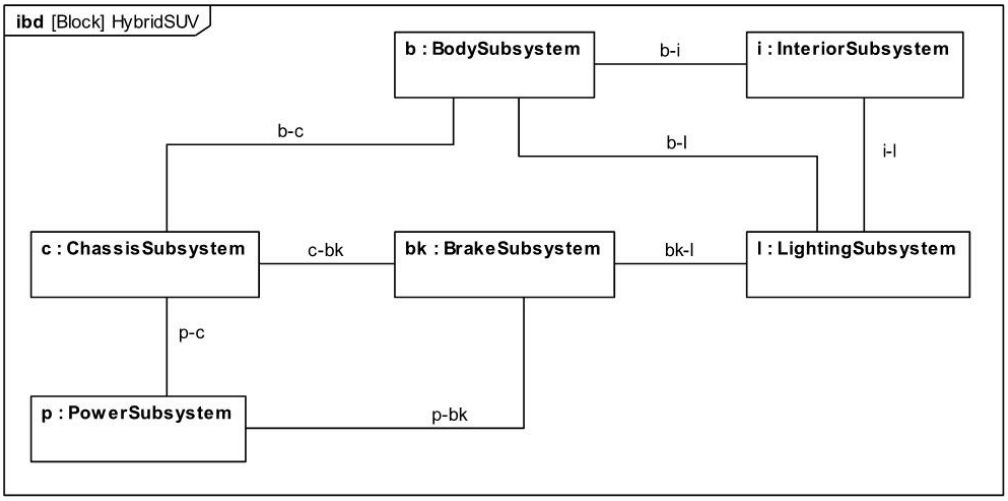


Figure D.17 Internal Structure of Hybrid SUV (Internal Block Diagram)

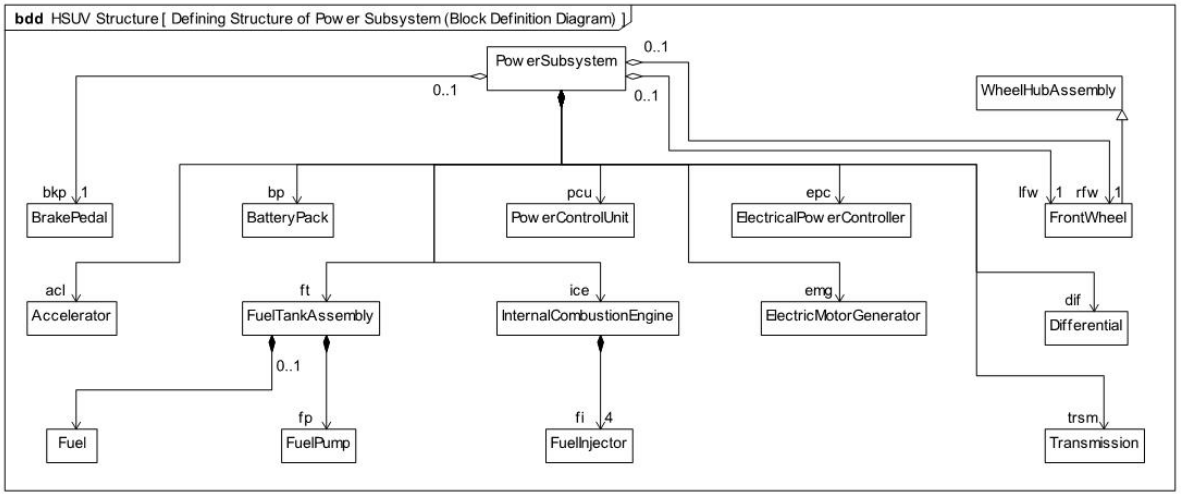


Figure D.18 Defining Structure of Power Subsystem (Block Definition Diagram)

D.4.5.4 Internal Block Diagram for the “Power Subsystem”

Figure D.19 shows how the parts of the PowerSubsystem block, as defined in the diagram above, are used. It shows connectors between parts, ports, and connectors with item flows. The dashed borders on FrontWheel and BrakePedal denote the “use-not-composition” relationship depicted elsewhere in Figure D.16 and Figure D.18. The dashed borders on Fuel denote a store, which keeps track of the amount and mass of fuel in the FuelTankAssy. This is also depicted in Figure D.18.

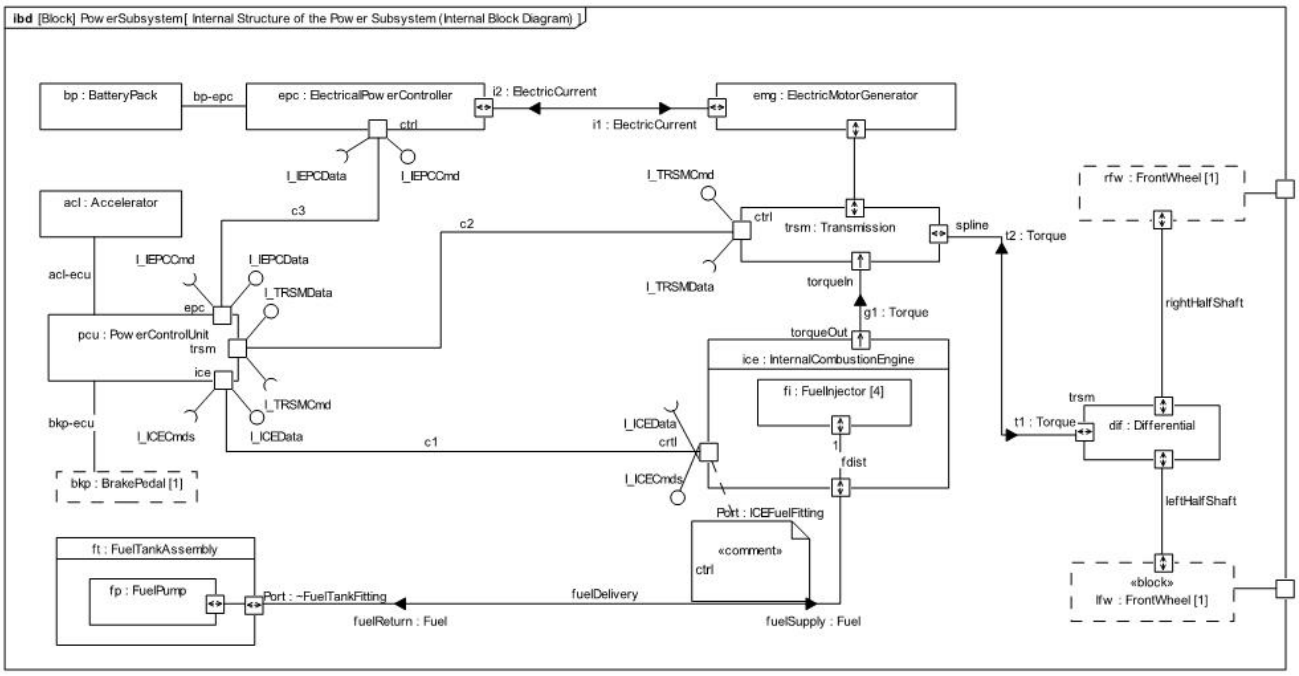


Figure D.19 Internal Structure of the Power Subsystem (Internal Block Diagram)

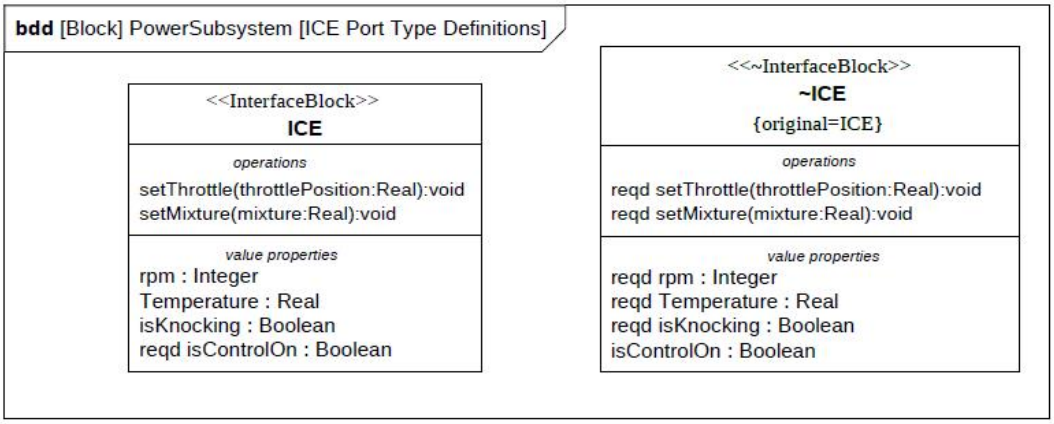


Figure D.20 Blocks Typing Ports in the Power Subsystem (Block Definition Diagram)

Figure D.20 provides definition of the block that types the ports linked by connector c1 in Figure D.19.

D.4.6 Defining Ports and Flows

D.4.6.1 Block Definition Diagram – ICE Flow Properties

For purposes of example, the ports, flows, and related point-to-point connectors in Figure D.19 are being refined into a common bus architecture. For this example, ports with flow properties have been used to model the bus architecture. Figure D.21 is an incomplete first step in the refinement of this bus architecture, as it begins to specify the flow properties for InternalCombustionEngine, the Transmission, and the ElectricalPowerController.

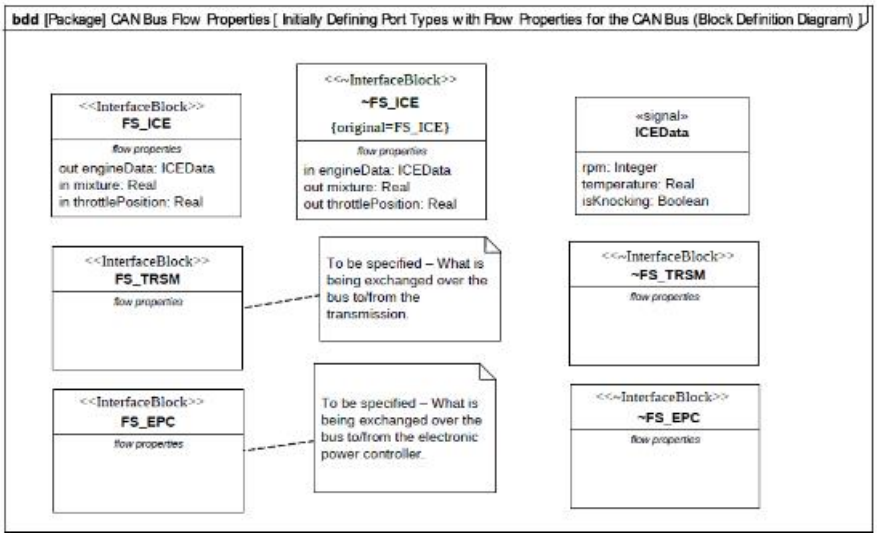


Figure D.21 Initially Defining Port Types with Flow Properties for the CAN Bus (Block Definition Diagram)

D.4.6.2 Internal Block Diagram – CANbus

Figure D.22 continues the refinement of this Controller Area Network (CAN) bus architecture using ports. The explicit structural allocation between the original connectors of Figure D.19 and this new bus architecture is shown in Figure D.39.

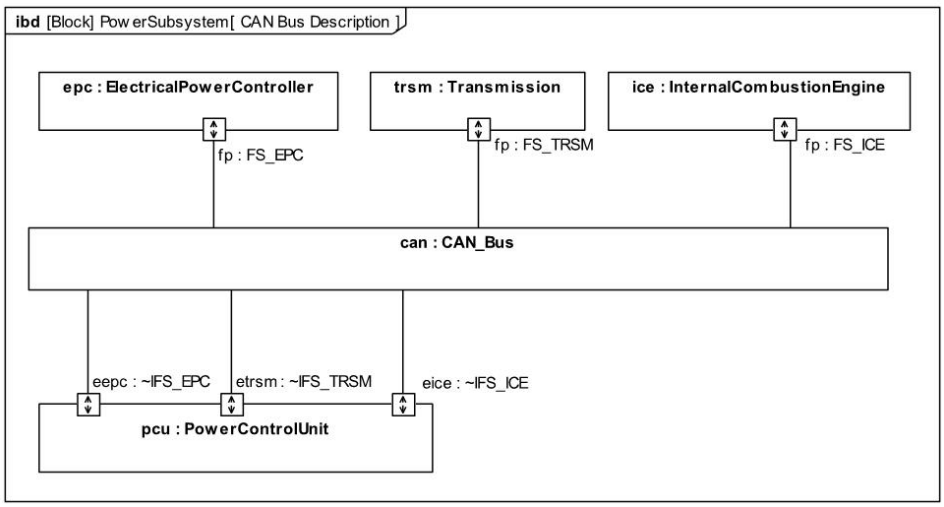


Figure D.22 Consolidating Connectors into the CAN Bus. (Internal Block Diagram)

D.4.6.3 Block Definition diagram – Fuel Flow Properties

The ports on the FuelTankAssembly and InternalCombustionEngine (as shown in Figure D.19) are defined in Figure D.23.

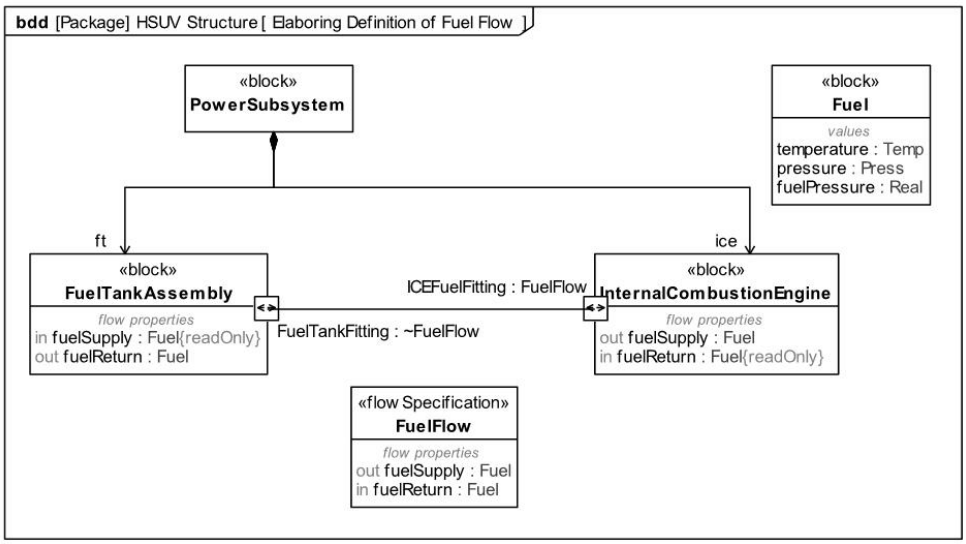


Figure D.23 Elaborating Definition of Fuel Flow. (Block Definition Diagram)

D.4.6.4 Parametric Diagram – Fuel Flow

Figure D.24 is a parametric diagram showing how fuel flowrate is related to FuelDemand and FuelPressure value properties.

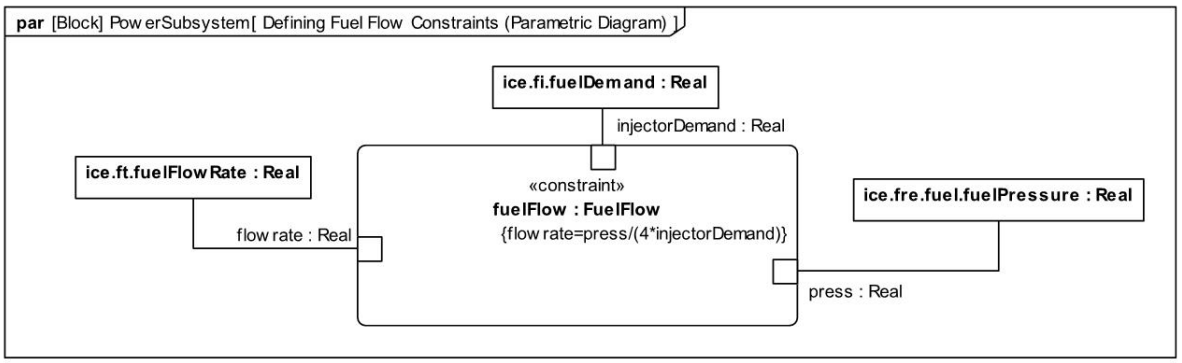


Figure D.24 Defining Fuel Flow Constraints (Parametric Diagram)

D.4.6.5 Internal Block Diagram – Fuel Distribution

Figure D.25 shows how the connectors fuelDelivery and fdist on Figure D.19 have been expanded to include design detail. The fuelDelivery connector is actually two connectors, one carrying fuelSupply and the other carrying fuelReturn. The fdist connector inside the InternalCombustionEngine block has been expanded into the fuel regulator and fuel rail parts. These more detailed design elements are related to the original connectors using the allocation relationship. The Fuel store represents a quantity of fuel in the FuelTankAssy, which is drawn by the FuelPump for use in the engine, and is refreshed, to some degree, by fuel returning to the FuelTankAssy via the FuelReturnLine.

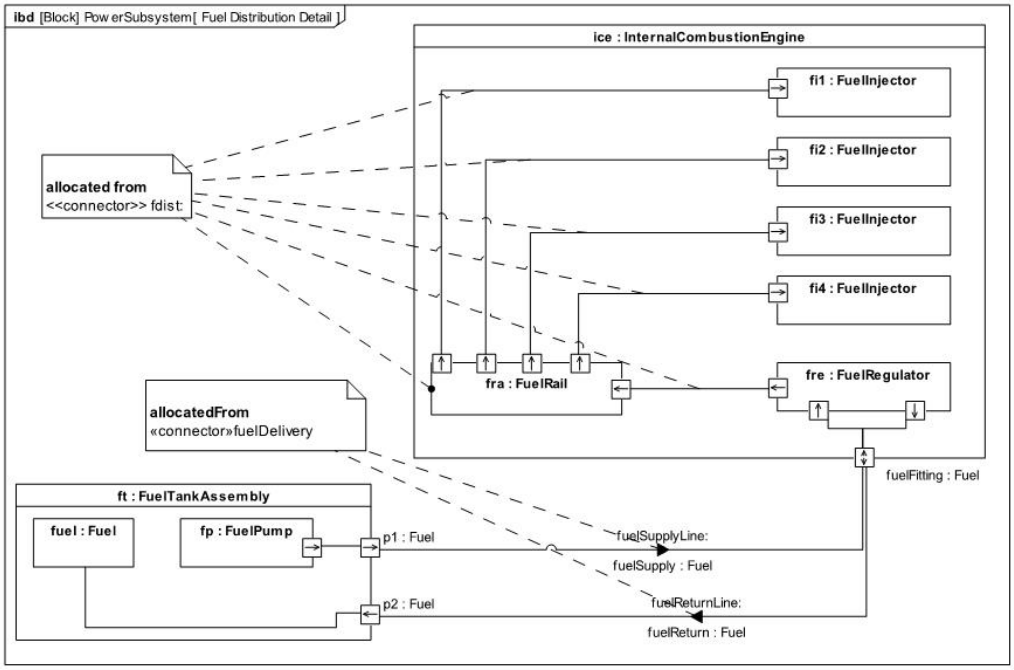


Figure D.25 Detailed Internal Structure of Fuel Delivery Subsystem (Internal Block Diagram)

D.4.7 Analyze Performance (Constraint Diagrams, Timing Diagrams, Views)

D.4.7.1 Block Definition Diagram – Analysis Context

Figure D.26 defines the various model elements that will be used to conduct analysis in this example. It depicts each of the constraint blocks/equations that will be used for the analysis, and key relationships between them.

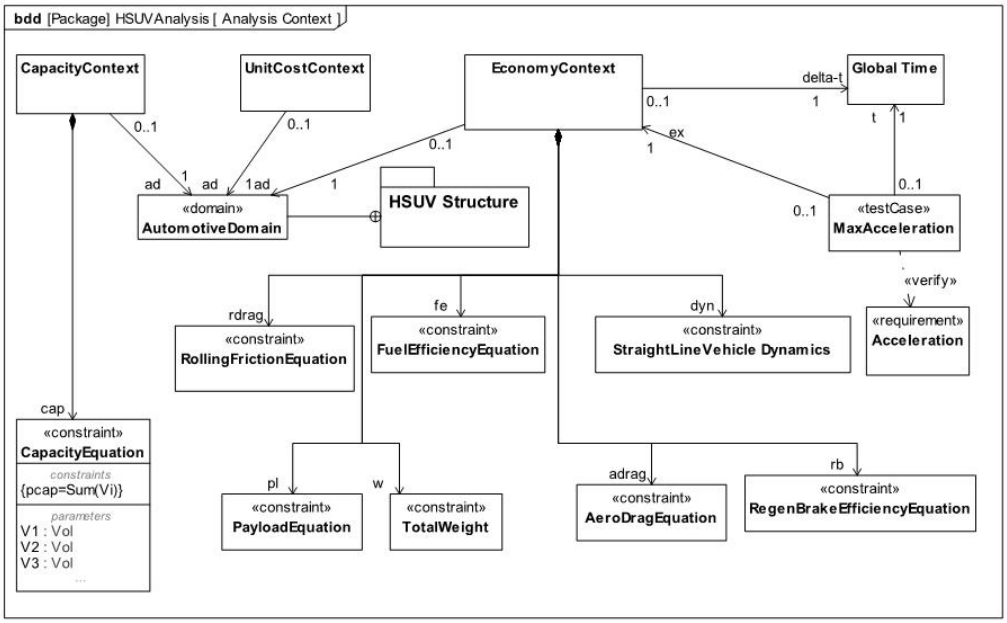


Figure D.26 Defining Analyses for Hybrid SUV Engineering Development (Block Definition Diagram)

D.4.7.2 Package Diagram – Performance View Definition

Figure D.27 shows the user-defined Performance Viewpoint, and the elements that populate the HSUV specific PerformanceView. The PerformanceView itself may contain a number of diagrams depicting the elements it contains.

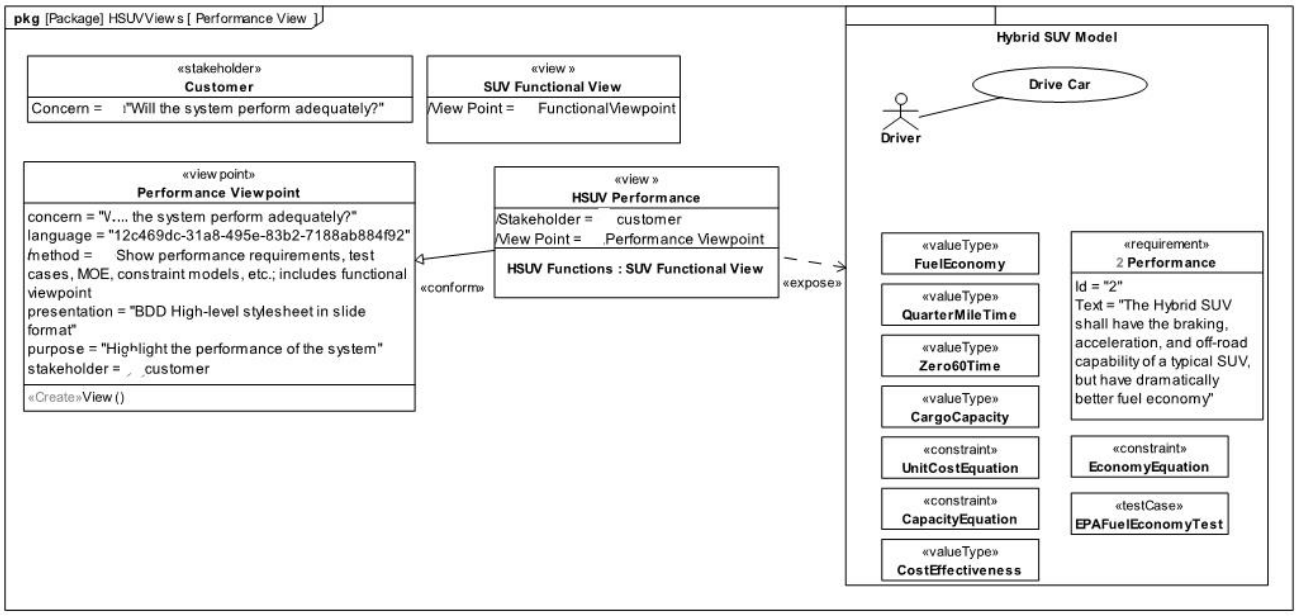


Figure D.27 Establishing a Performance View of the User Model (Package Diagram)

D.4.7.3 Package Diagram – Viewpoint Definition

Figure D.28 shows the Requirements and VnV viewpoint definitions with relationships to stakeholders, concerns and views. The stakeholder and viewpoint share the same concern via comments that are shown textually as values of the concern property. The comments could be shown graphically with annotation relationships to stakeholders and viewpoints, if needed. Note that the value of the stakeholder property is an instance of the stereotype not the class to which the stereotype is applied.

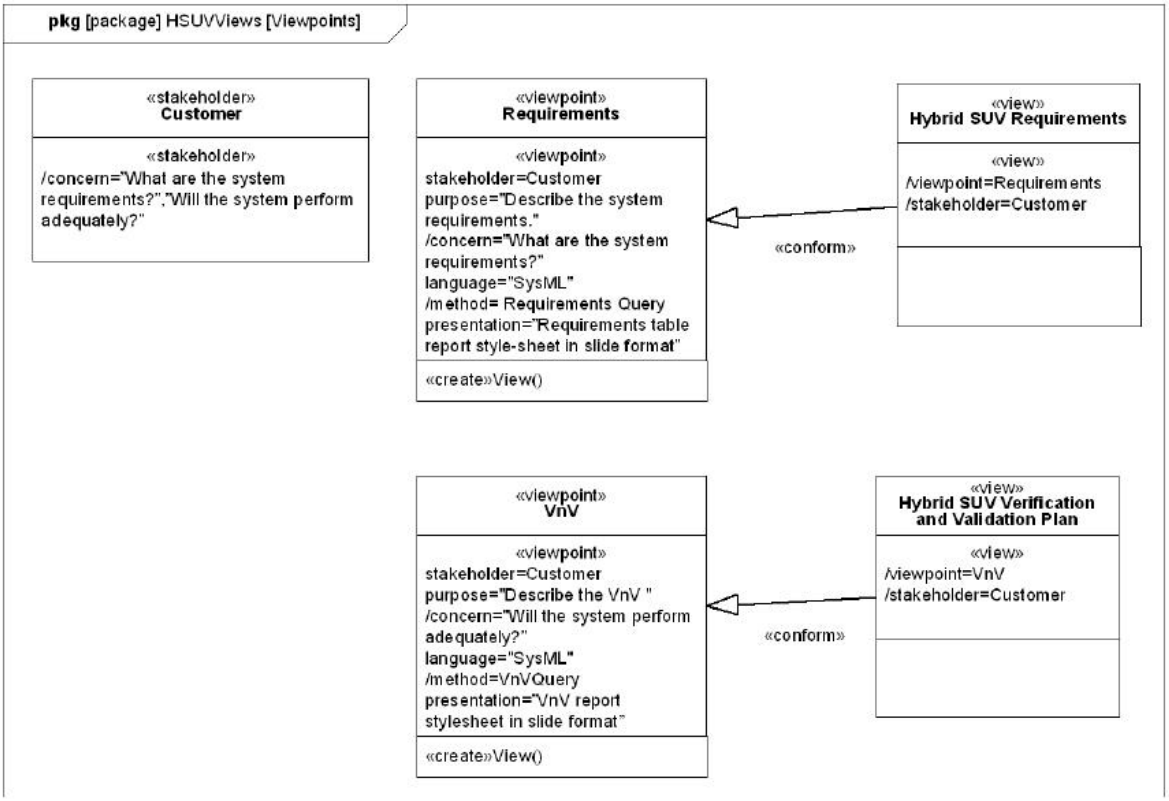


Figure D.28 Defining Requirements and VnV viewpoints (Package Diagram)

D.4.7.4 Package Diagram – View Definition

Figure D.29 shows the Requirements and VnV views and the model elements they expose. Note that the expose relationship relies on the viewpoint method to identify the entire set of elements that appear in the view.

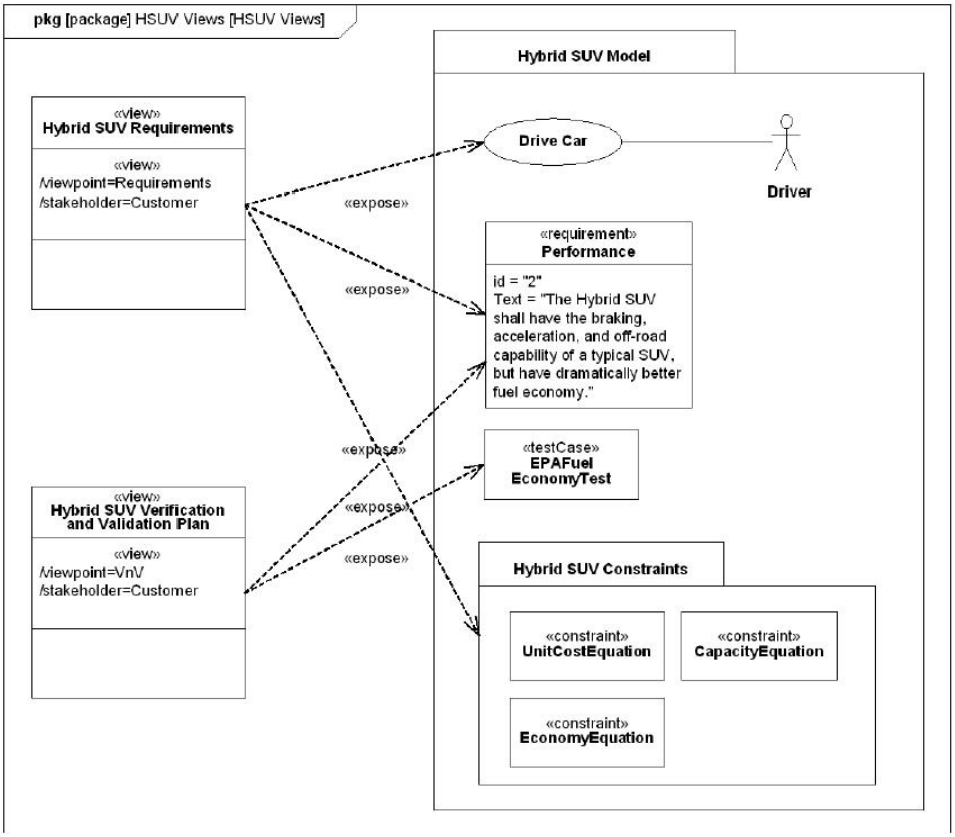


Figure D.29 Requirements and VnV views exposing elements from the model (Package Diagram)

D.4.7.5 Package Diagram – View Hierarchy

Figure D.30 shows the Requirements and VnV views and the supporting views that complete the description of Requirements and VnV respectively for the Hybrid SUV.

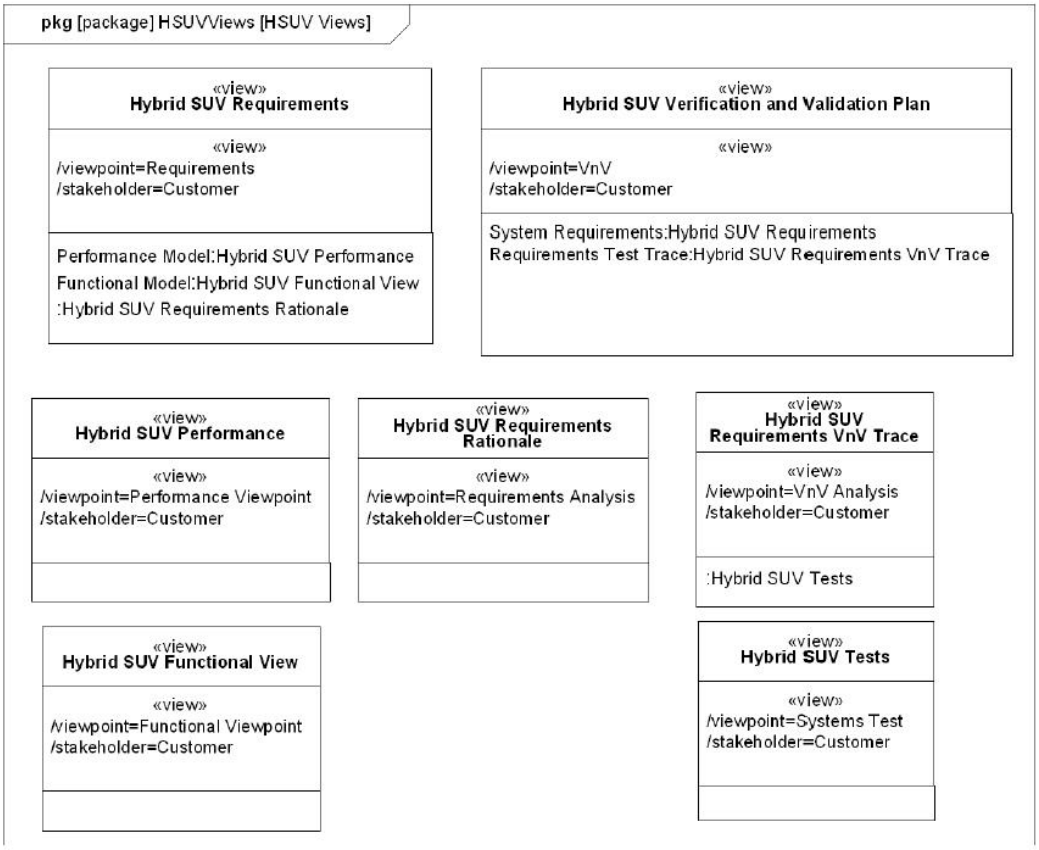


Figure D.30 The Requirements and VnV views with supporting views (Package Diagram)

D.4.7.6 Package Diagram – Measures of Effectiveness

Measure of Effectiveness is a user defined stereotype. Figure D.31 shows how the overall cost effectiveness of the HSUV will be evaluated. It shows the particular measures of effectiveness for one particular alternative for the HSUV design, and can be reused to evaluate other alternatives.

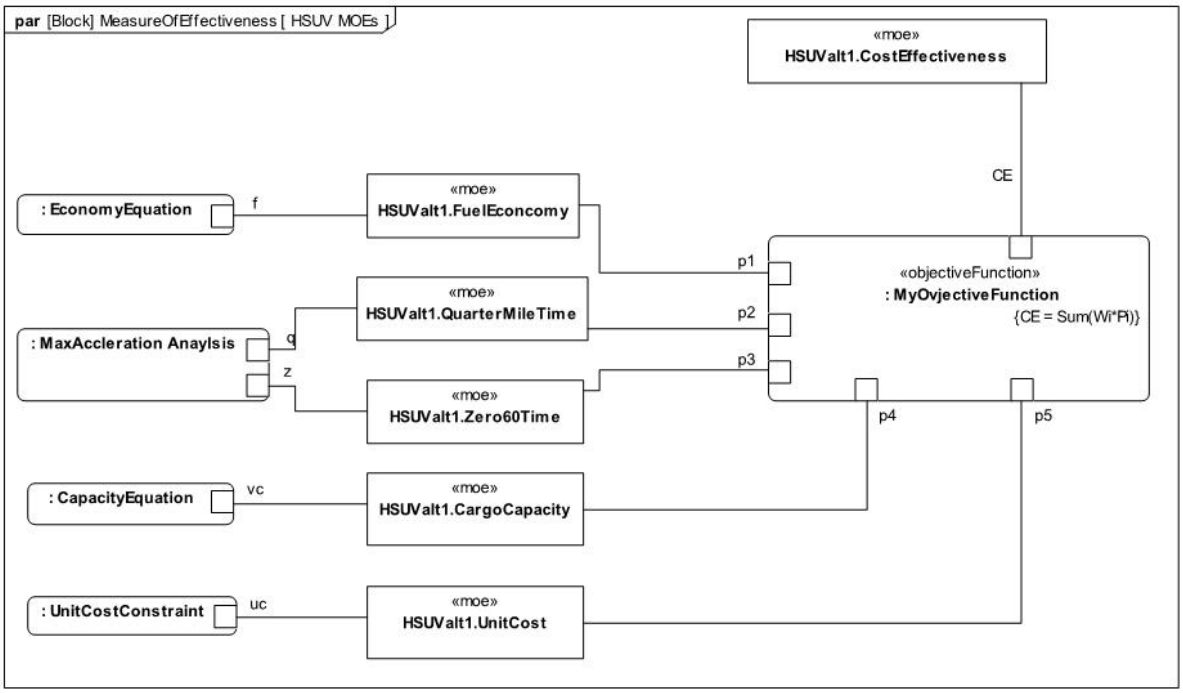


Figure D.31 Defining Measures of Effectiveness and Key Relationships (Parametric Diagram)

D.4.7.7 Parametric Diagram – Economy

Since overall fuel economy is a key requirement on the HSUV design, this example applies significant detail in assessing it. Figure D.32 shows the constraint blocks and properties necessary to evaluate fuel economy.

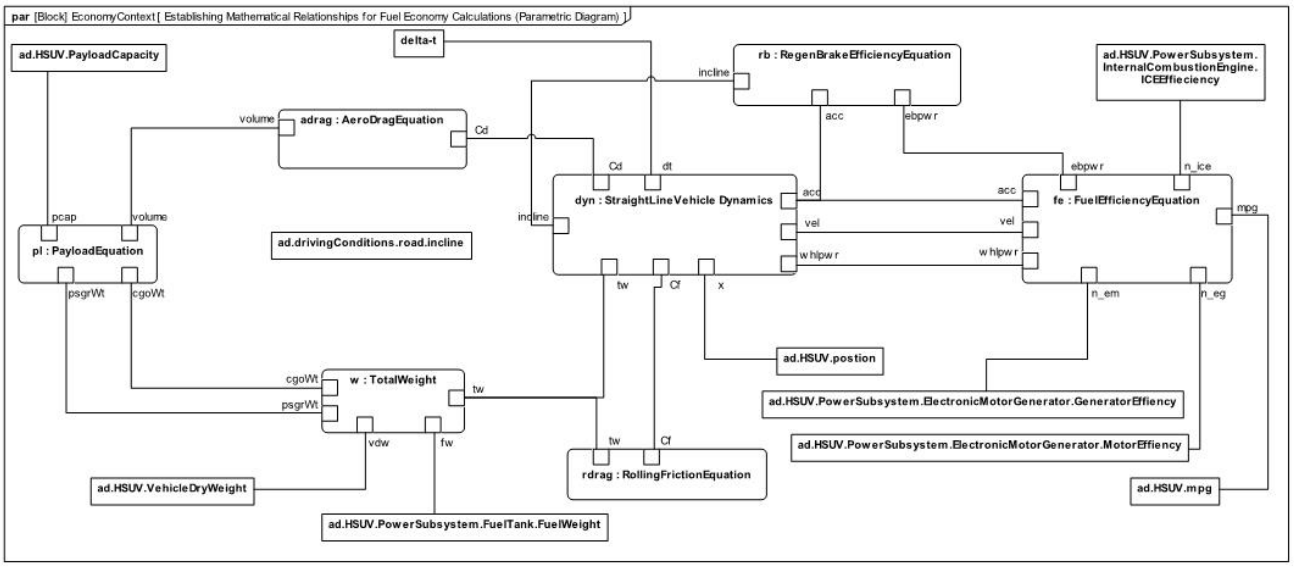


Figure D.32 Establishing Mathematical Relationships for Fuel Economy Calculations (Parametric Diagram)

D.4.7.8 Parametric Diagram – Dynamics

The StraightLineVehicleDynamics constraint block from Figure D.32 has been expanded in Figure D.33. ConstraintNotes are used, which identify each constraint using curly brackets {}. In addition, Rationale has been used to explain the meaning of each constraint maintained.

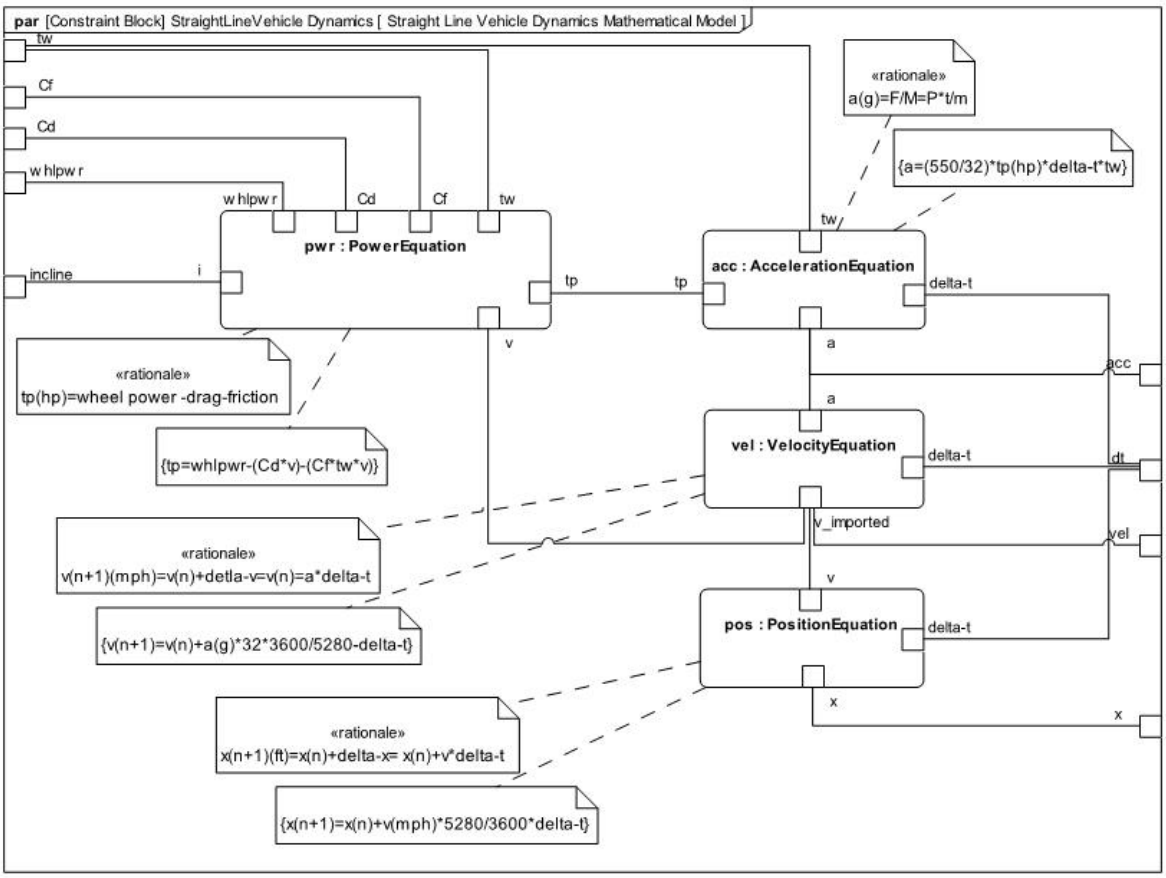


Figure D.33 Straight Line Vehicle Dynamics Mathematical Model (Parametric Diagram)

The constraints and parameters in Figure D.33 are detailed in Figure D.34 in Block Definition Diagram format.

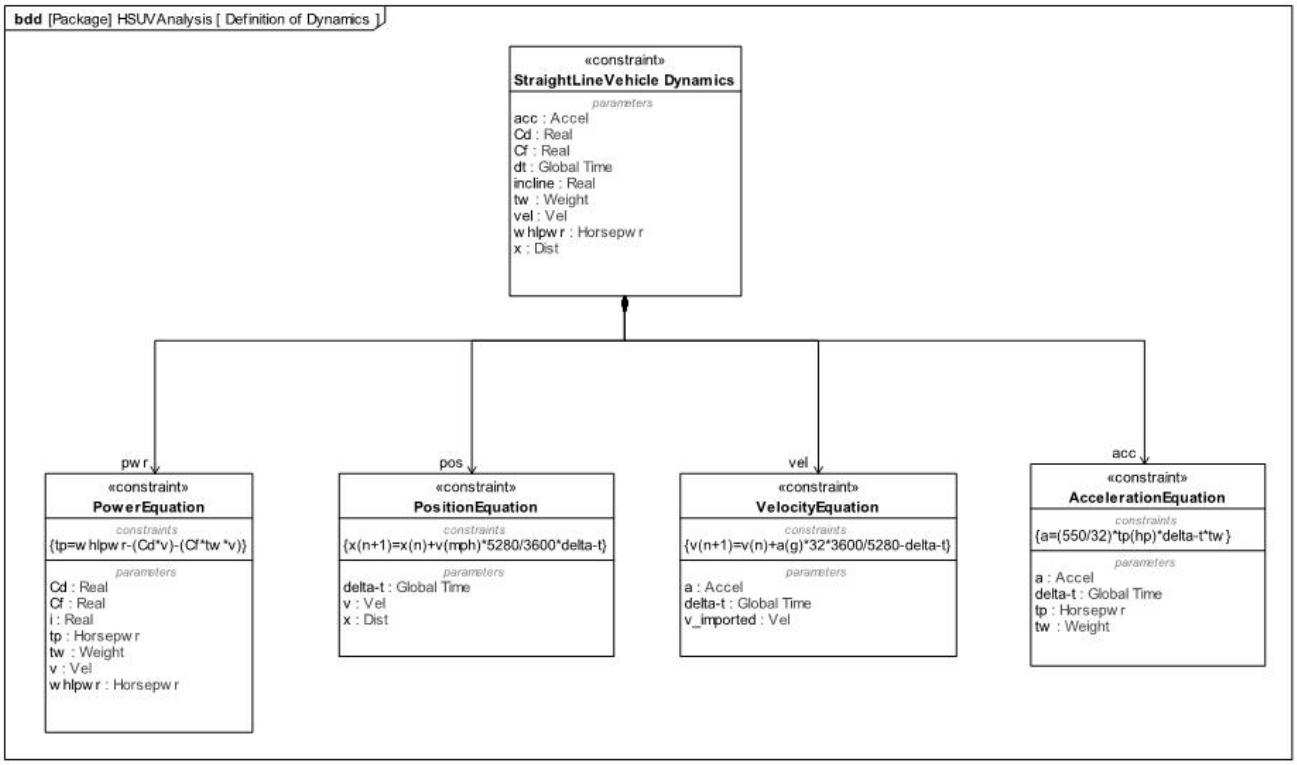


Figure D.34 Defining Straight-Line Vehicle Dynamics Mathematical Constraints (Block Definition Diagram)

Note the use of valueTypes originally defined in Figure D.2.

D.4.7.9 (Non – Normative) Timing Diagram – 100hp Acceleration

Timing diagrams, while included in UML 2, are not directly supported by SysML. For illustration purposes, however, the interaction shown in Figure D.35 was generated based on the constraints and parameters of the StraightLineVehicleDynamics constraintBlock, as described in the Figure D.33. It assumes a constant 100hp at the drive wheels, 4000lb gross vehicle weight, and constant values for Cd and Cf.

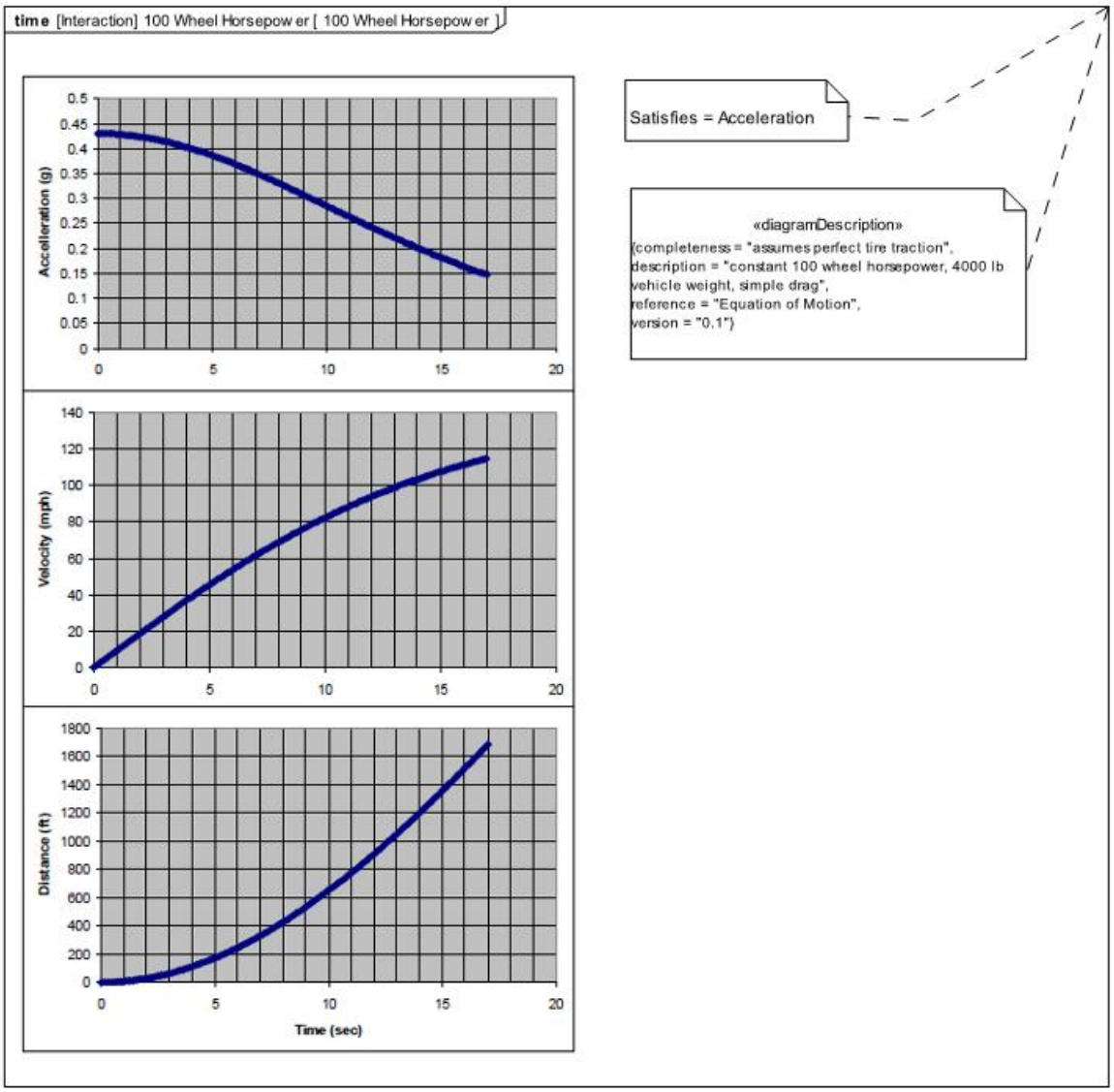


Figure D.35 Results of maximum Acceleration Analysis (Timing Diagram)

D.4.8 Defining, Decomposing, and Allocating Activities

D.4.8.1 Activity Diagram – Acceleration (top level)

Figure D.36 shows the top level behavior of an activity representing acceleration of the HSUV. It is the intent of the systems engineer in this example to allocate this behavior to parts of the PowerSubsystem. It is quickly found, however, that the behavior as depicted cannot be allocated, and must be further decomposed. The stereotypes on the object nodes between actions in the figure apply to parameters of the behaviors or operations called by the actions (see the notation for object nodes described in 11.3.1.4, ObjectNode, Variables, and Parameters).

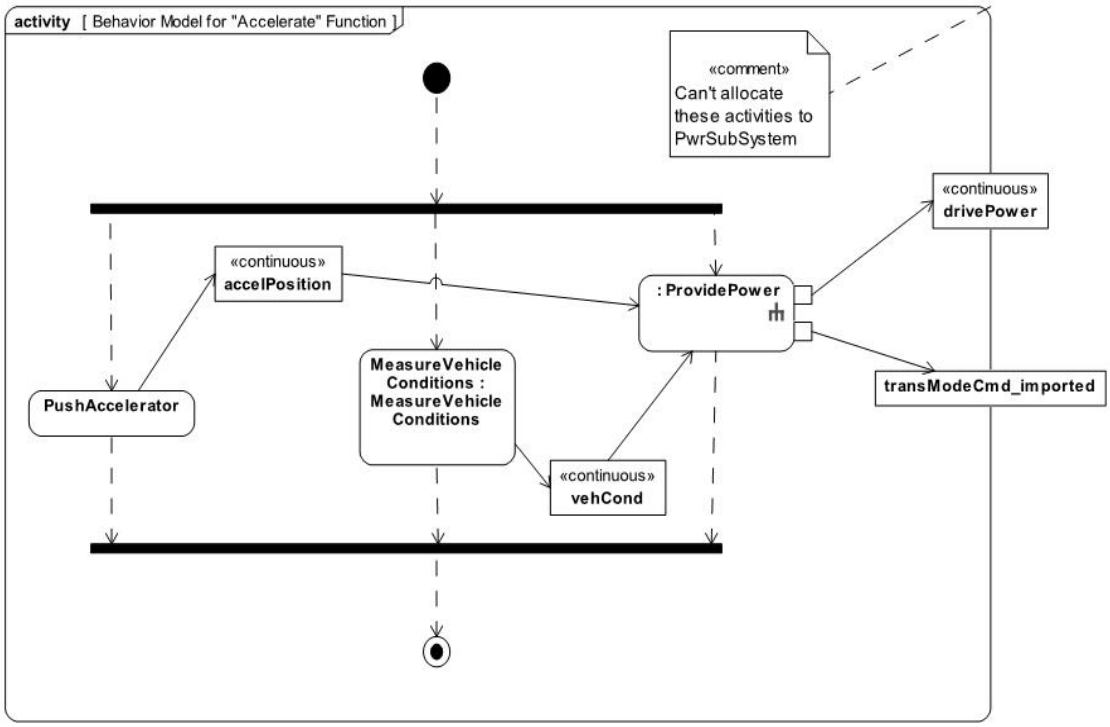


Figure D.36 Behavior Model for "Accelerate" Function (Activity Diagram)

D.4.8.2 Block Definition Diagram – Acceleration

Figure D.37 defines a decomposition of the activities and objectFlows from the activity diagram in Figure D.36.

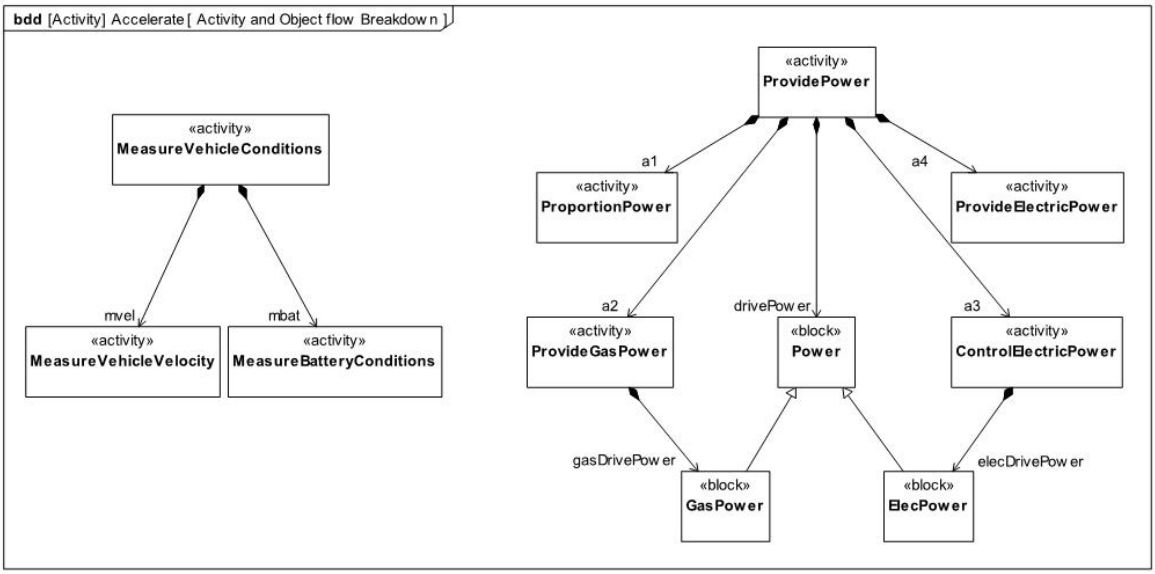


Figure D.37 Decomposition of "Accelerate" Function (Block Definitions diagram)

D.4.8.3 Activity Diagram (EFFBD) – Acceleration (detail)

Figure D.38 shows the ProvidePower activity, which includes Actions invoking the decomposed Activities and ObjectNodes from Figure D.37. It also uses AllocateActivityPartitions and an allocation callout to explicitly allocate activities and an object flow to parts in the PowerSubsystem block.

Note that the incoming and outgoing object flows for the ProvidePower activity have been decomposed. This was done to distinguish the flow of electrically generated mechanical power and gas generated mechanical power, and to provide further insight into the specific vehicle conditions being monitored.

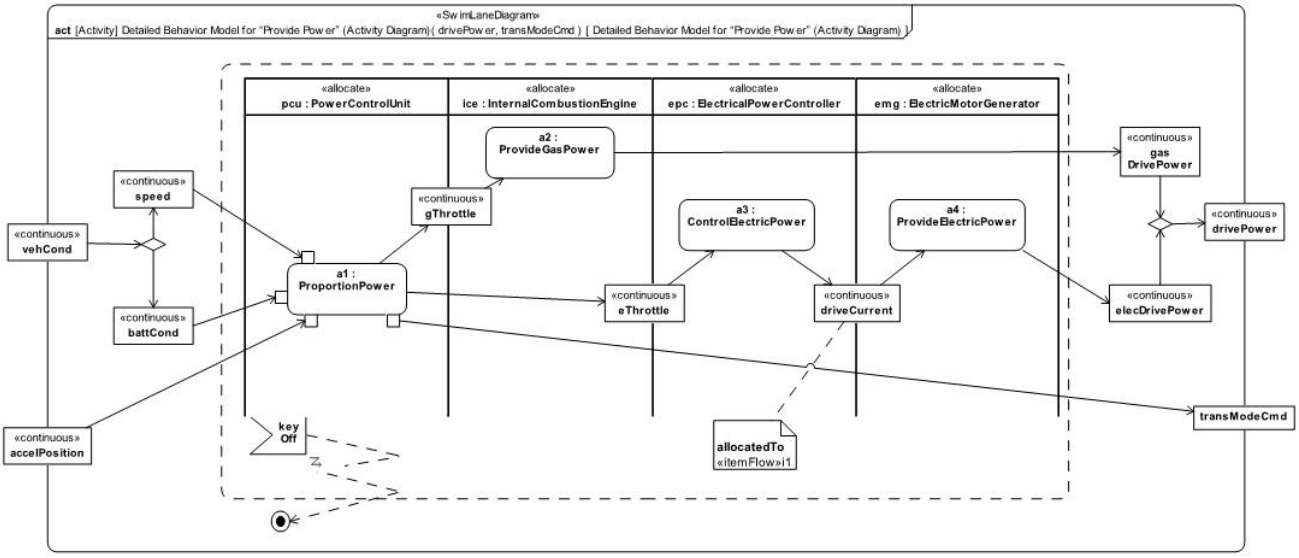


Figure D.38 Detailed Behavior Model for "Provide Power" (Activity Diagram)
Note hierarchical consistency with Figure D.36.

D.4.8.4 Internal Block Diagram – Power Subsystem Behavioral and Flow Allocation

Figure D.39 depicts a subset of the PowerSubsystem, specifically showing the allocation relationships generated in Figure D.38.

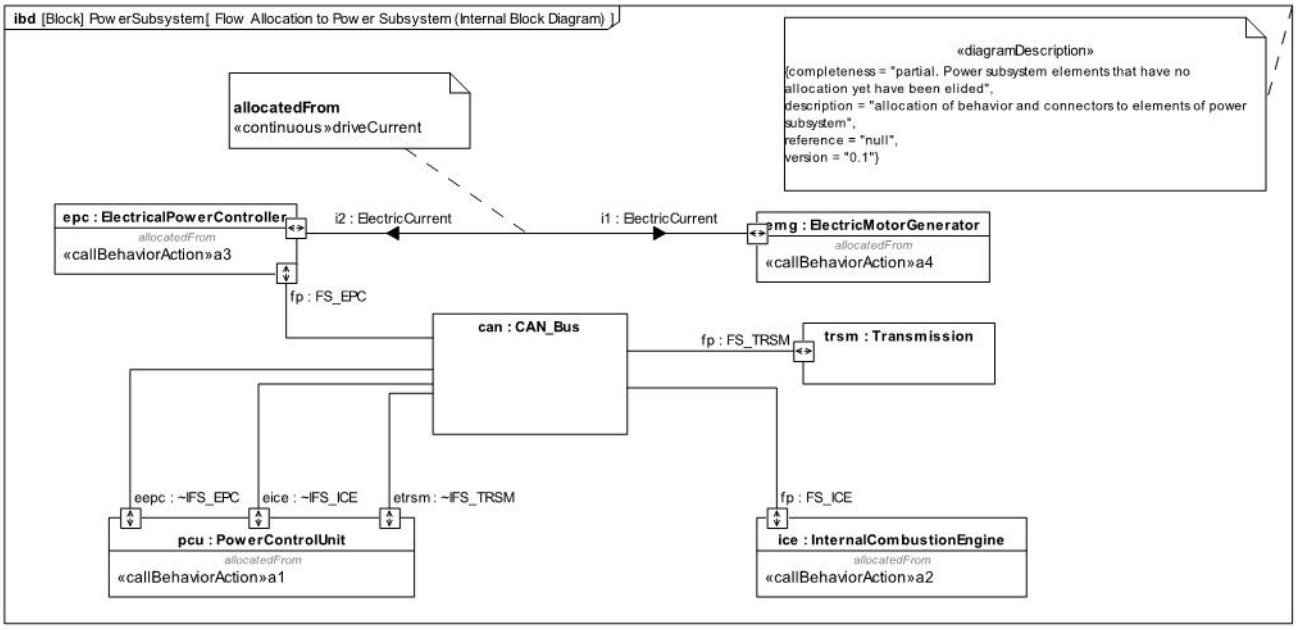


Figure D.39 Flow Allocation to Power Subsystem (Internal Block Diagram)

D.4.8.5 Table – Acceleration Allocation

Figure D.40 shows the same allocation relationships shown in Figure D.38, but in a more compact tabular representation.

D.4.8.6 Internal Block Diagram: Property Values – EPA Fuel Economy Test

bdd [Package] HSUV Behavior [Figure B.37 Tabular Representation of Allocation from"Accelerate" Behavior Model to Power Subsystem]

Type	Name	End	Relation	End	Type	Name
action	a1 : ProportionPower	from	allocate	to	part	ecu : PowerControlUnit
action	a2 : ProvideGasPower	from	allocate	to	part	ice : InternalCombustionEngine
action	a3 : ControlElectricPower	from	allocate	to	part	epc : ElectricPowerController
action	a4 : ProvideElectricPower	from	allocate	to	part	emg : ElectricMotorGenerator
objectFlow	o6	from	allocate	to	connector	epc-emg.1

Figure D.40 Tabular Representation of Allocation from "Accelerate" Behavior Model to Power Subsystem (Table)

Figure D.41shows a particular Hybrid SUV (VIN number) satisfying the EPA fuel economy test. Serial numbers of specific relevant parts are indicated.

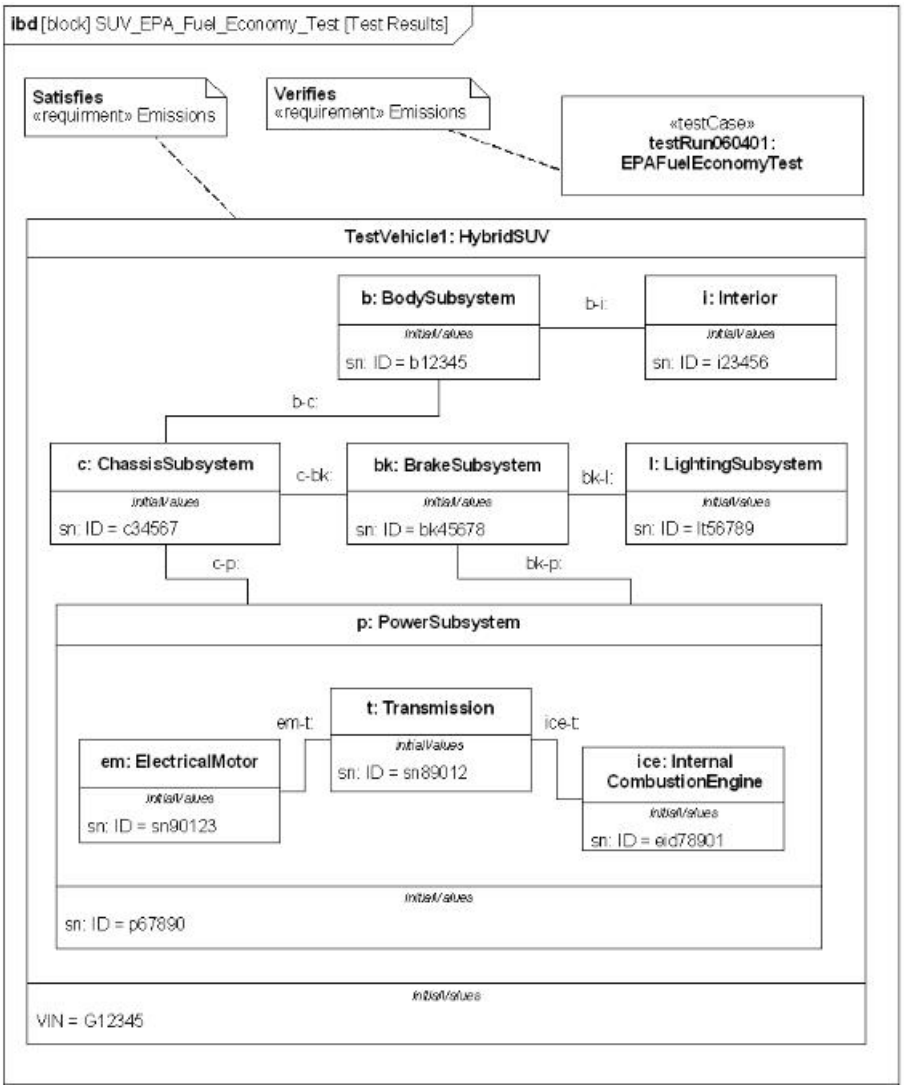


Figure D.41 Special Case of Internal Block Diagram Showing Reference to Specific Properties (serial numbers)

This page intentionally left blank.

Annex E: Non-normative Extensions

(Informative)

E.1 Overview

This annex describes useful non-normative extensions to SysML that may be considered for standardization in future versions of the language.

Non-normative extensions consist of stereotypes and model libraries and are organized by major diagram type, consistent with how the main body of this International Standard is organized. Stereotypes in this sub clause are specified using a tabular format, consistent with how non-normative stereotypes are specified in the UML 2 standard. Model libraries are specified using the guidelines provided in the Profiles & Model Libraries clause of this International Standard.

E.2 Activity Diagram Extensions

E.2.1 Overview

Two non-normative extensions to activities are described for:

- Enhanced Functional Flow Block Diagrams.
- Streaming activities that accept inputs and/or provide outputs while they are active.

More information on these extensions and the standard SysML extensions is available at [Bock. C., "SysML and UML 2.0 Support for Activity Modeling," vol. 9, no. 2, pp. 160-186, Journal of the International Council of Systems Engineering, 2006].

E.2.2 Stereotypes

Enhanced Functional Flow Block Diagrams (EFFBD) are a widely-used systems engineering diagram, also called a behavior diagram. Most of its functionality is a constrained use of UML activities, as described below. This extension does not address replication, resources, or kill branches. Kill branches can be translated to activities using interruptible regions and join specifications.

Table E-1: Addition stereotypes for EFFBDs

Stereotype	Base class	Properties	Constraints	Description
«effbd»	UML4SysML::Activity (or subtype of «nonStreaming» below)	N/A	See below.	Specifies that the activity conforms to the constraints necessary for EFFBD.

When the «effbd» stereotype is applied to an activity, its contents shall conform to the following constraints:

[1] (On Activity) Activities shall not have partitions.

[2] (On Activity) All decisions, merges, joins, and forks shall be well-nested. In particular, each decision and merge shall be matched one-to-one, as are forks and joins, accounting for the output parameter sets acting as decisions, and input parameters and control acting as a join.

[3] (On Action) All actions shall have exactly one control edge coming into them, and exactly one control edge coming out,except when using parameter sets.

[4] (Execution constraint) All control shall be enabling.

[5] (On ControlFlow) All control flows into an action target a pin on the action that shall have isControl = true.

[6] (On ObjectNode) Ordering shall be first-in first out, ordering = FIFO.

[7] (On ObjectNode) Object flow shall be never used for control, isControlType = false, except for pins of parameters in parameter sets.

[8] (On Parameter) Parameters shall take and produce no more than one item, multiplicity.upper =1.

[9] (On Parameter) Output parameters shall produce exactly one value, multiplicity.lower = 1. The «optional» stereotype cannot be applied to parameters.

[10] (On Parameter) Parameters shall not be streaming or exception.

[11] (On ParameterSet) Parameter sets shall only apply to output parameters.

[12] (On ParameterSet) Parameter sets shall only apply to control. Parameters in parameter sets shall have pins with is ControlType = true.

[13] (On ParameterSet) Parameter sets shall have exactly one parameter, and it shall not be shared with other parameter sets.\

[14] (On ParameterSet) If one output parameter is in a parameter set, then all output parameters of the behavior or operation shall be in parameter sets.

[15] (On ActivityEdge) Edges shall not have time constraints.

[16] The following SysML stereotypes shall not be applied: «rate», «controlOperator», «noBuffer», «overwrite».

A second extension distinguishes activities based on whether they can accept inputs or provide outputs after they start and before they finish (streaming), or only accept inputs when they start and provide outputs when they are finished (nonstreaming). EFFBD activities are nonstreaming. Streaming activities are often terminated by other activities, while nonstreaming activities usually terminate themselves.

Table E-2: Streaming options for activities

Stereotype	Base class	Properties	Constraints	Description
«streaming»	UML4SysML::Activity	N/A	The activity has at least one streaming parameter.	Used for activities that can accept inputs or provide outputs after they start and before they finish.
«nonStreaming»	UML4SysML::Activity	N/A	The activity has no streaming parameters.	Used for activities that accept inputs only when they start, and provide outputs only when they finish.

E.2.3 Stereotype Examples

Figure E.1 shows an example activity diagram with the «effbd» stereotype applied, translated from [Long. J., "Relationships between common graphical representations in system engineering," 2002]. The stereotype applies the constraints specified in E.2.2 Stereotypes, for example, that the data outputs on all functions are required and that queuing is FIF.

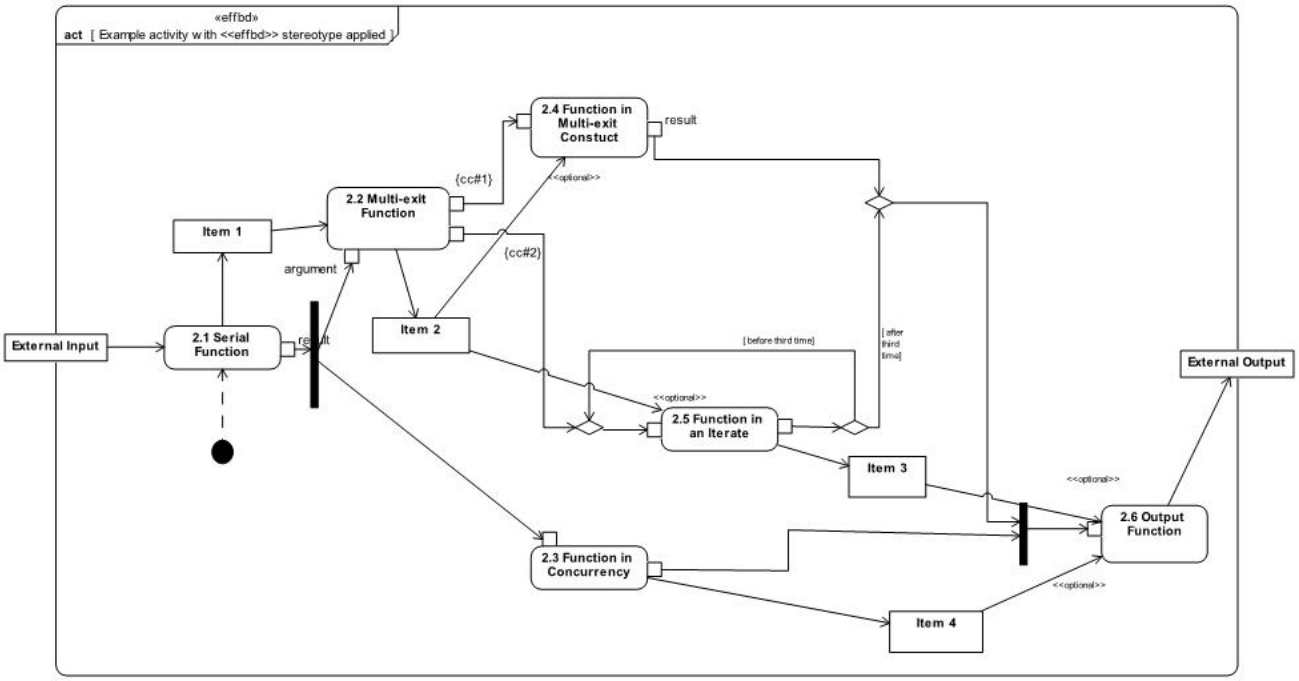


Figure E.1 Example activity with «effbd» stereotype applied

Figure E.2 shows an example activity diagram with the «streaming» and «nonStreaming» stereotypes applied, adapted from [MathWorks, "Using Simulink," 2004]. It is a numerical solution for the differential equation x'(t) = -2x(t) + u(t). Item types are omitted brevity. The «streaming» and «nonStreaming» stereotypes indicate which subactivities take inputs and produce outputs while they are executing. They are simpler to use than the {stream} notation on streaming inputs and outputs.

The example assumes a default of zero for the lower input to Add, and that the entire activity is executed with clocked token flow, to ensure that actions with multiple inputs receive as many of them as possible before proceeding. See the article referenced in E.2.1 Overview.

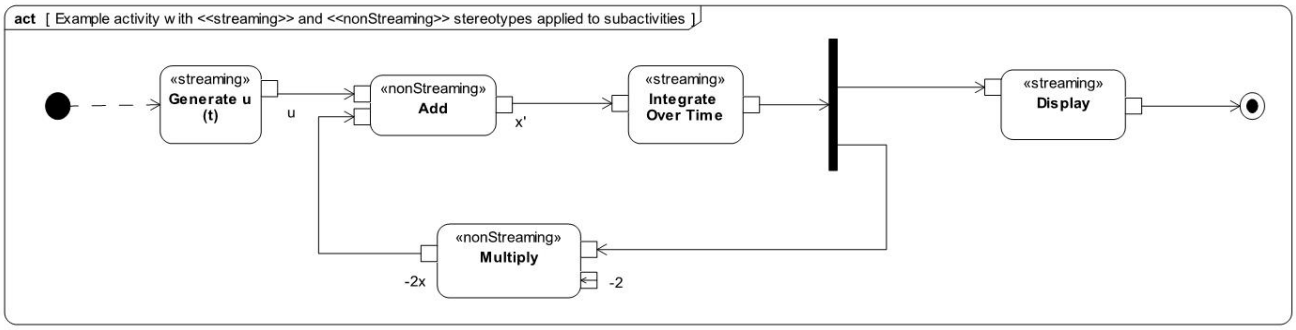


Figure E.2 Example activity with «streaming» and «nonStreaming» stereotypes applied to subactivities

E.3 Requirements Diagram Extensions

E.3.1 Overview

This sub clause describes an example of a non-normative extension for a requirements profile.

E.3.2 Stereotypes

This non-normative extension includes stereotypes for a simplified requirements taxonomy that is intended to be further adapted as required to support the particular needs of the application or organization. The requirements categories in this example include functional, interface, performance, physical requirements, and design constraints as shown in Table E-3. As shown in the table, each category is represented as a stereotype of the generic SysML «requirement». The table also includes a brief description of the category. The table does not include any stereotype properties or constraints, although they can be added as deemed appropriate for the application. For example, a constraint that could be applied to a functional requirement is that only SysML activities and operations can satisfy this category of requirement. Other examples of requirements categories may include operational, specialized requirements for reliability and maintainability, store requirements, activation, deactivation, and a high level category for stakeholder needs.

Some general guidance for applying a requirements profile is as follows:

- The categories should be adapted for the specific application or organization and reflected in the table. This includes agreement on the categories and their associated descriptions, stereotype properties, and constraints. Additional categories can be added by further subclassing the categories in the table below, or adding additional categories at the pier level of these categories.
- The default requirement category should be the generic «requirement».
- Apply the more specialized requirement stereotype (functional, interface, performance, physical, design constraint) as applicable and ensure consistency with the description, stereotype properties, and constraints.
- A specific text requirement can include the application of more than one requirement category, in which case, each stereotype should be shown in guillemets.

Table E-3: Additional Requirement Stereotypes

Stereotype	Base class	Properties	Constraints	Description
«extendedRequirement»	«requirement»	source: String risk: RiskKind verifyMethod: VerifyMethodKind	N/A	A mix-in stereotype that contains generally useful attributes for requirements.
«functionalRequirement»	«extendedrequirement»	N/A	satisfied by an operation or behavior	Requirement that specifies an operation or behavior that a system, or part of a system, must perform.
«interfaceRequirement»	«extendedrequirement»	N/A	satisfied by a port, conector, item flow, and/or constraint property	Requirement that specifies the ports for connecting systems and system parts and the optionally may include the item flows across the connector and/or interface constraints.
«performanceRequirement»	«extendedrequirement»	N/A	satisfied by a value property	Requirement that quantitavely measures the extent to which a system, or a system part, satisfies a required capability or condition.
«physicalRequirement»	«extendedrequirement»	N/A	satisfied by a structural element	Requirement that specifies physical characteristics and/or physical constraints of the system, or a system part.
«designConstraint»	«extendedrequirement»	N/A	satisfied by a block or part	Requirement that specifies a constraint on the implementation of the system or system part, such as the system must use a commercial off the shelf component.

Table E-4 provides the definition of the non-normative enumerations that are used to type properties of "extendedRequirement" stereotype of Figure E.3.

Table E-4: Requirement property enumeration types

Enumeration	Enumeration Literals	Example Description
RiskKind	High	High indicates an unacceptable level of risk
	Medium	Medium indicates an acceptable level of risk
	Low	Low indicates a minimal level of risk or no risk
VerificationMethodKind	Analysis	Analysis indicates that verification will be performed by technical evaluation using mathematical representations, charts, graphs, circuit diagrams, data reduction, or representative data. Analysis also includes the verification of requirements under conditions, which are simulated or modeled; where the results are derived from the analysis of the results produced by the model.
	Demonstration	Demonstration indicates that verification will be performed by operation, movement or adjustment of the item under specific conditions to perform the design functions without recording of quantitative data. Demonstration is typically considered the least restrictive of the verification types.
	Inspection	Inspection indicates that verification will be performed by examination of the item, reviewing descriptive documentation, and comparing the appropriate characteristics with a predetermined standard to determine conformance to requirements without the use of special laboratory equipment or procedures.
	Test	Test indicates that verification will be performed through systematic exercising of the applicable item under appropriate conditions with instrumentation to measure required parameters and the collection, analysis, and evaluation of quantitative data to show that measured parameters equal or exceed specified requirements.

E.3.3 Stereotype Examples

Figure E.3 shows the use of several subtypes of requirements extended to include the properties risk:RiskKind, verifyMethod:VerficationMethodKind, and a text attribute source:String, used to capture the source of the requirement.

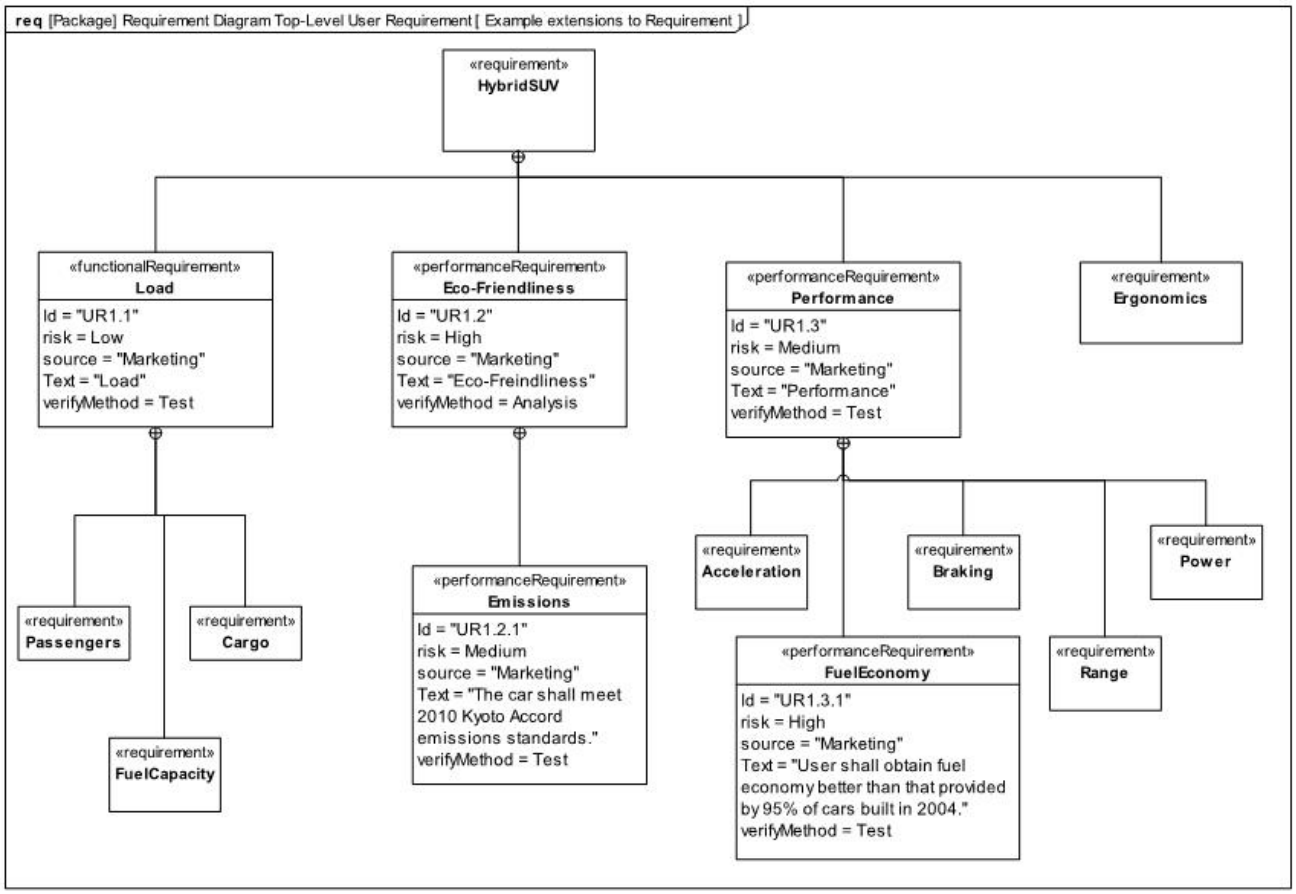


Figure E.3 Example extensions to Requirement

E.4 Parametric Diagram Extensions for Trade Studies

E.4.1 Overview

This sub clause describes a non-normative extension of a parametric diagram (refer to the Constraint Blocks clause) to support trade studies and analysis, which are an essential aspect of any systems engineering effort. In particular, a trade study is used to evaluate a set of alternatives based on a defined set of criteria. The criteria may have a weighting to reflect their relative importance. An objective function (aka optimization or cost function) can be used to represent the weighted criteria and determine the overall value of each alternative. The objective function can be more complex than a simple linear weighting of the criteria and can include probability distribution functions and utility functions associated with each criteria. However, for this example, we will assume the simpler case.

A measure of effectiveness (moe) represents a parameter whose value is critical for achieving the desired mission cost effectiveness. It will also be assumed that the overall mission cost effectiveness can be determined by applying an objective function to a set of criteria, each of which is represented by a measure of effectiveness.

This non-normative extension includes stereotypes for an objective function and a measure of effectiveness. The objective function is a stereotype of a ConstraintBlock and the measure of effectiveness is a stereotype of a block property.

E.4.2 Stereotypes

Table E-5: Stereotypes for Measures of Effectiveness

Stereotype	Base class	Properties	Constraints	Description
«objectiveFunction»	«ConstraintBlock»	N/A	N/A	An objective function (aka optimization or cost function) is used to determine the overall value of an alternative in terms of weighted criteria and/or moe's.
«moe»	UML4SysML::Property	N/A	N/A	A measure of effectiveness (moe) represents a parameter whose value is critical for achieving the desired mission cost effectiveness.

E.4.3 Stereotype Examples

In this example, operational availability, mission response time, and security effectiveness each represent moes along with life cycle cost. The overall cost effectiveness for each alternative may be defined by an objective function that represents a weighted sum of their moe values. For each moe, there is a separate parametric model to estimate the value of operational availability, mission response time, security effectiveness, and life cycle cost to determine an overall cost effectiveness for each alternative. It is assumed that the moes refer to the values for system alternative j (sj).

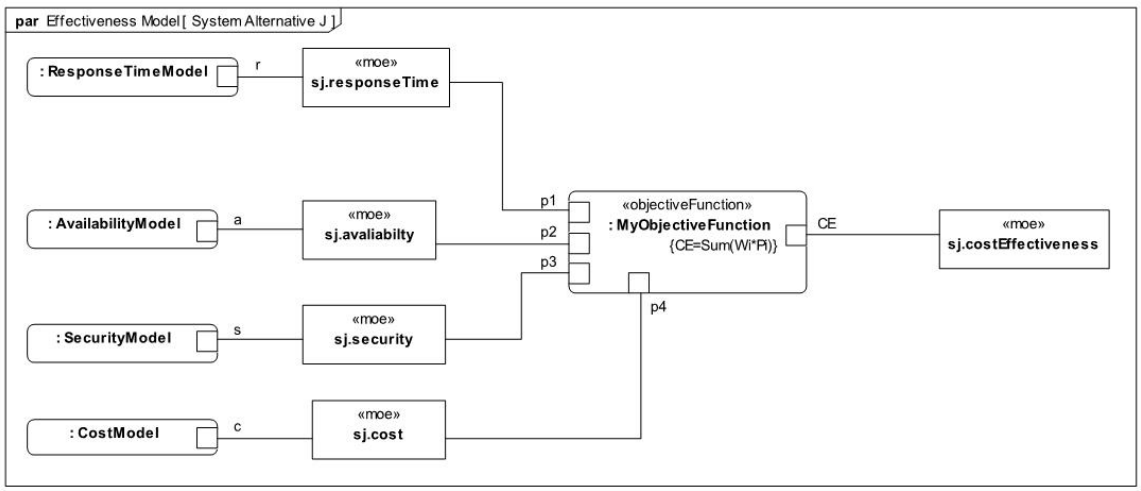


Figure E.4 Example Parametric Diagram using Stereotypes for Measures of Effectiveness

E.5 Model Library for Quantities, Units, Dimensions, and Values (QUDV)

E.5.1 Overview

For any system model, a solid foundation of well-defined quantities, units, and dimensions system is very important. Properties that describe many aspects of a system depend on it. At the same time, such a foundation should be a shareable resource that can be reused in many models within and across organizations and projects.

The most widely accepted, scrutinized, and globally used system of quantities and system of units are the International System of Quantities (ISQ) and the International System of Units (SI). They are formally standardized through [ISO31] and [IEC60027]. The harmonization of these two sets of standards into one new set [ISO/IEC80000] has been published by ISO in 2009 and 2010. The present QUDV model in SysML is based on ISO/IEC 80000-1:2009, which refers normatively to the ISO/IEC Guide 99:2007. The ISO/IEC 80000-1:2009 document is also the baseline for the 2010 revision of the IEEE/ASTM American National Standards for Metric Practice SI-10. All the relevant concepts underlying ISQ and SI are publicly available in [VIM]. See E.5.3, References for references to these documents.

At a minimum, SysML should provide the means to support the imminent international standard [ISO/IEC80000]. In addition, many other systems of quantities and units are still in use for particular applications and for historical reasons. A prime example is the system based on UK Imperial units, which is still widely used in North America. SysML should provide the means to support all such specific systems of quantities and units, including precise definitions of the relationships between different systems of units, and with explicit and unambiguous unit conversions to and from SI as well as other systems.

To provide a solid and stable foundation, the model for defining quantities, units, dimensions, and values in SysML is explicitly based on the concepts defined in [VIM], which have been written by the authoritative Working Group 2 of the Joint Committee for Guides in Metrology (JCGM/WG 2), in which the JCGM member organizations are represented: BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, and OIML. At the same time, the model library is designed in such a way that extensions to the ISQ and SI can be represented, as well as any alternative systems of quantities and units.

The model library can be used to support SysML user models in various ways. A simple approach is to define and document libraries of reusable systems of units and quantities for reuse across multiple projects, and to link units and quantity kinds from these libraries to Unit and QuantityKind stereotypes defined in SysML user models. The name of a Unit or QuantityKind stereotype, its definitionURI, or other means may be used to link it with definitions made using this library. Instances of blocks conforming to this model library may be created by instance specifications, as shown in E.5.4 Usage Examples, or by other means.

Even though this model library is specified in terms of SysML blocks, its contents could equally be specified by UML classes without dependencies on any SysML extensions. This annex specifies the model library using SysML blocks to maintain compatibility with the SysML standard. UML and other forms of this same conceptual model are important and useful to align different standards with each other and with those of [VIM].

Separate forms of this model library, including a UML class model generated as a simple transformation from the model library specified in this annex, together with additional mappings and resources, example applications, and reference libraries of systems of units and quantities built using this model, are expected to be published via the SysML Project Portal wiki at https://www.omgwiki.org/OMGSysML/.

E.5.2 Abstract Syntax

Figures E.5 - E.7 present the QUDV model library in a series of block definition diagrams.

The QUDV Concepts diagram in Figure E.5 presents the core concepts of System of Units, Unit, SystemOfQuantities, and QuantityKind. The QUDV concepts of Unit and QuantityKind are specialized by restriction from their respective SysML concepts shown in gray in Figure E.5. The QUDV concepts form the basis of the QUDV subset of the Vocabulary of International Metrology (VIM) from ISO 80000-1 and JCGM 200:2012. In SysML, a value property typed by a given ValueType, with stereotype properties that refer to a SysML Unit and/or QuantityKind, defines a quantity in the sense of ISO 80000-1, Sub clause 3.1. If specified, the unit of the ValueType designates the measurement unit assumed for the numerical value of such a quantity.

In the QUDV Unit diagram in Figure E.6, SimpleUnit provides the basis for defining other units via conversion or derivation. Additionally, QUDV provides support for specifying a coherent derived unit as a product of the baseUnit(s) of a given SystemOfUnits. In a coherent SystemOfUnits, there is only one base unit for each base quantity kind.

In the QUDV QuantityKind diagram in Figure E.7, SimpleQuantityKind provides the basis for defining other quantity kinds via specialization or derivation. QUDV provides a declarative specification of dimensional analysis to assign to each QuantityKind an expression of its dependence on the baseQuantityKind(s) of a SystemOfQuantities. This dependence is expressed as a list of QuantityKindFactor(s) corresponding to a product of powers of the base quantities. E.5.2.15 SystemOfQuantities, specifies the derivation of quantity dimensions using an algorithm specified in OCL.

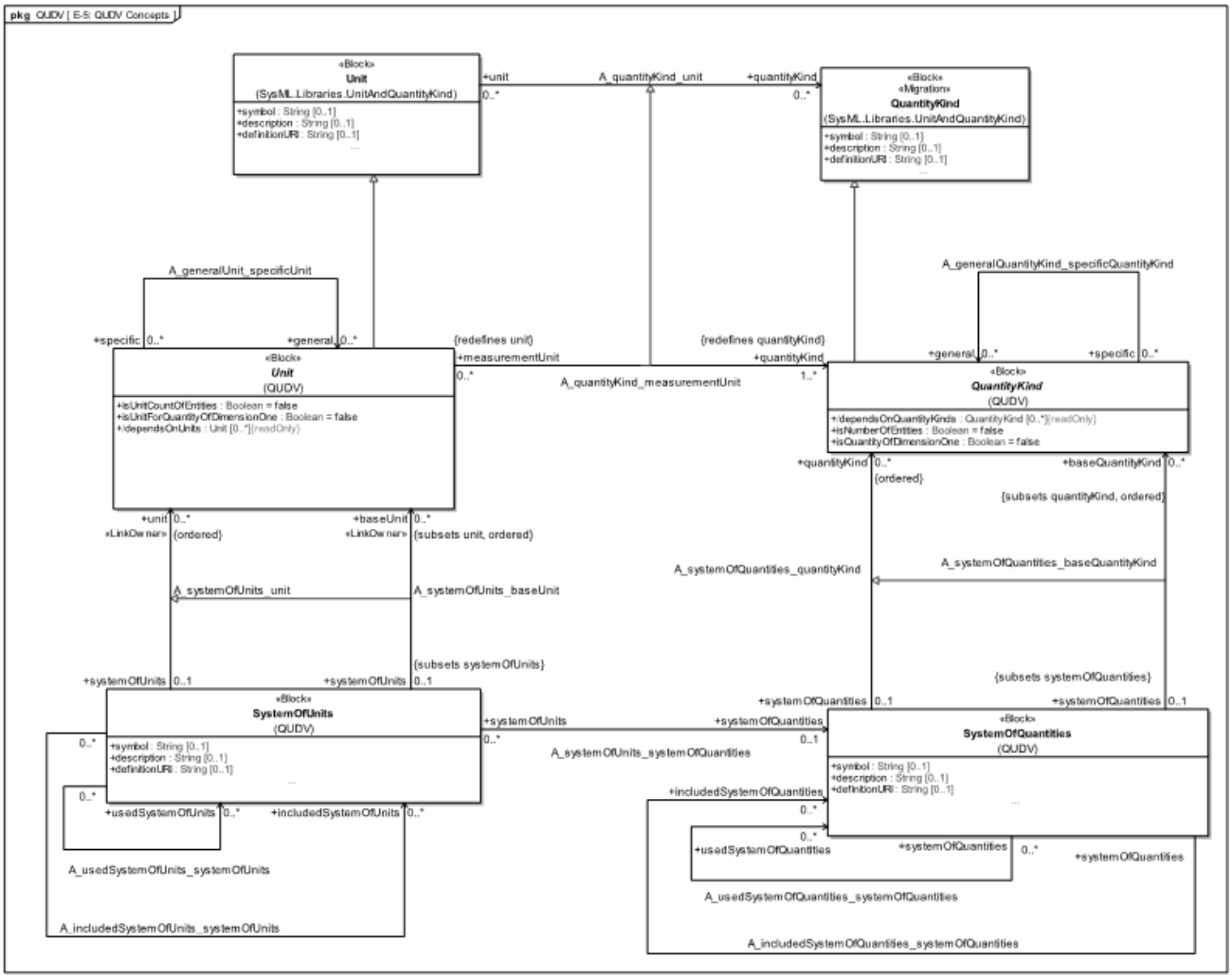


Figure E.5 QUDV Concepts Diagram

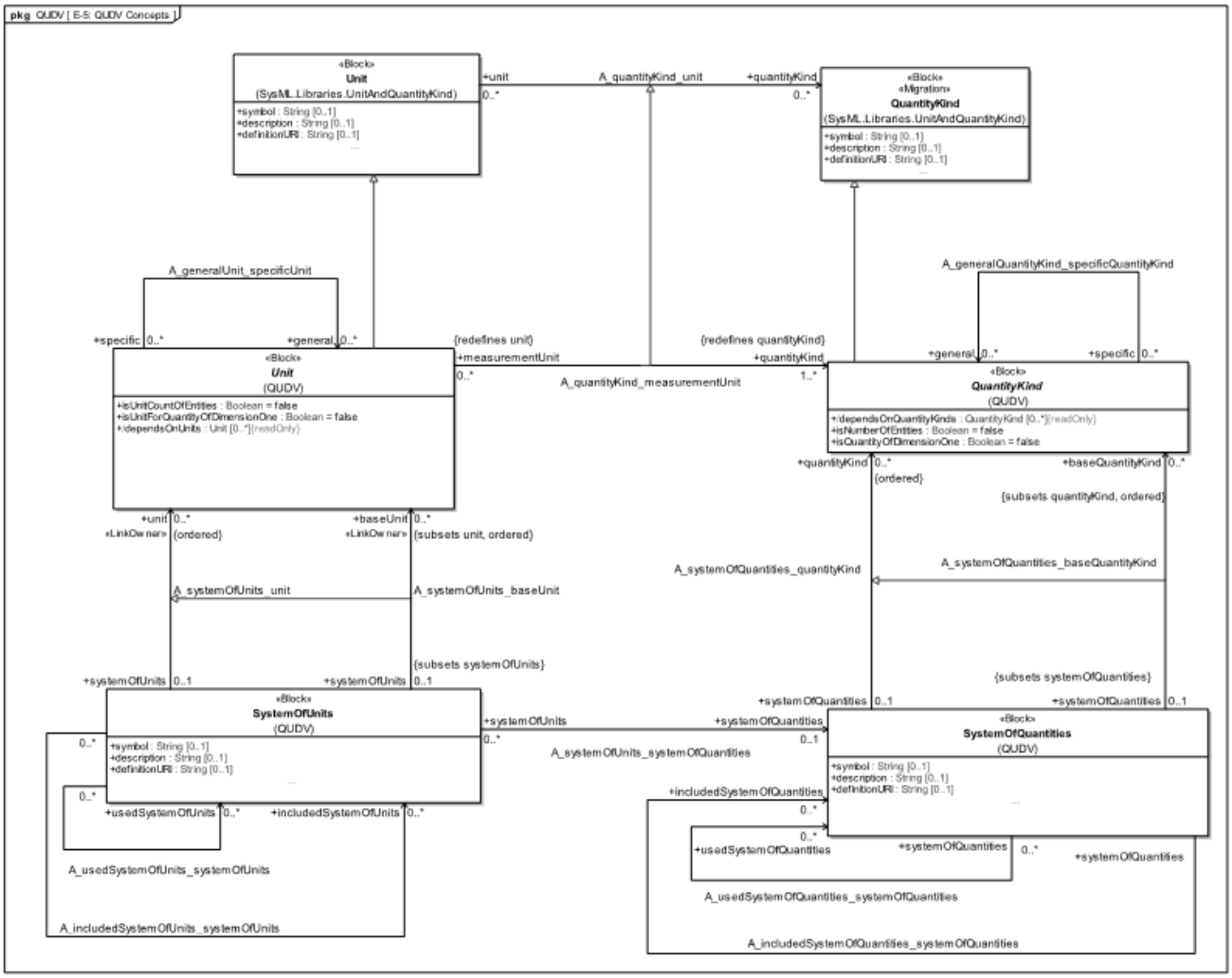


Figure E.6 QUDV Units Diagram

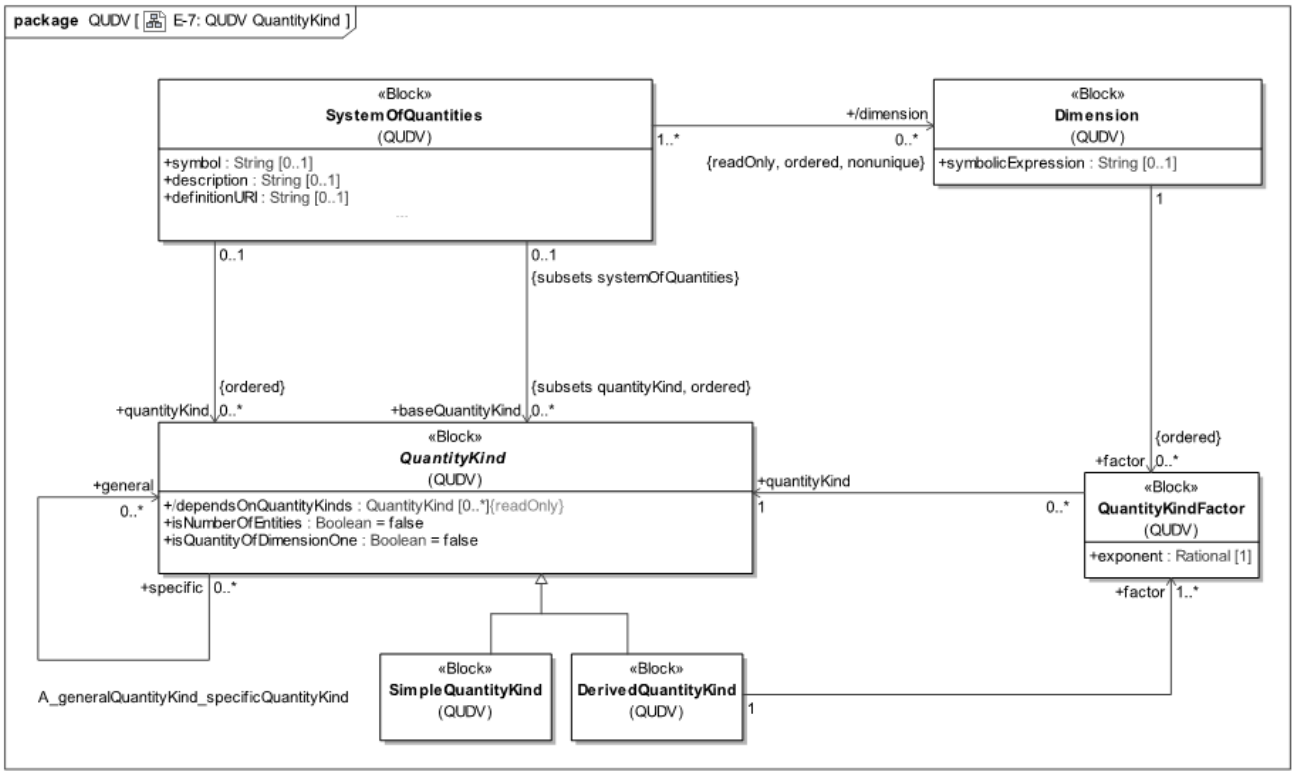


Figure E.7 QUDV QuantityKinds Diagram

E.5.2.1 AffineConversionUnit

Description

An AffineConversionUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another reference measurement unit through an affine conversion relationship with a conversion factor and offset.

The unit conversion relationship is defined by the following equation:

$value_{RU} = factor \cdot value_{CU} + offset$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit, and,

$value_{CU}$ is the quantity value expressed in the AffineConversionUnit.

For example, in the definition of the AffineConversionUnit for “degree Fahrenheit” with respect to the referenceUnit “degree Celsius,” the factor would be 5/9 and the offset would be -160/9, because

$T_{Celsius} = 5/9 \cdot T_{Fahrenheit} - 160/9$ which is equivalent with $T_{Fahrenheit} = 9/5 \cdot T_{Celsius} + 32/1$

Properties

- factor: Number
  Number that specifies the factor in the unit conversion relationship.
- offset: Number
  Number that specifies the offset in the unit conversion relationship.

E.5.2.2 ConversionBasedUnit

Description

A ConversionBasedUnit is an abstract classifier that is a Unit that represents a measurement unit that is defined with respect to another reference unit through an explicit conversion relationship.

Properties

- referenceUnit: Unit
  Specifies the unit with respect to which the ConversionBasedUnit is defined.
- inInvertible: Boolean
  Specifies whether the unit conversion relationship is invertible. For LinearConversionUnit and AffineConversionUnit this is always true.

Operations

[1] A ConversionBasedUnit transitively depends on its referenceUnit and all of the Units that its referenceUnit depends on.

```
dependsOnUnits() : Unit[0..*] {unique}
body: referenceUnit.dependsOnUnits()->including(referenceUnit)->asSet()
```

E.5.2.3 DerivedQuantityKind

Description

A DerivedQuantityKind is a QuantityKind that represents a kind of quantity that is defined as a product of powers of one or more other kinds of quantity. A DerivedQuantityKind may also be used to define a synonym kind of quantity for another kind of quantity.

For example “velocity” can be specified as the product of “length” to the power one times “time” to the power minus one, and subsequently “speed” can be specified as “velocity” to the power one.

Properties

- factor: QuantityKindFactor [1..*]
  Set of QuantityKindFactor that specifies the product of powers of other kind(s) of quantity that define the DerivedQuantityKind.

Operations

[1] A DerivedQuantityKind transitively depends on its factors' QuantityKinds and all of the QuantityKinds that its factors' QuantityKinds depend on.

```
dependsOnQuantityKinds() : QuantityKind[0..*] {unique}
body: factor.quantityKind.dependsOnQuantityKinds()->flatten()->asSet()
->union(factor.quantityKind->flatten()->asSet())->asSet()
```

E.5.2.4 DerivedUnit

Description

A DerivedUnit is a Unit that represents a measurement unit that is defined as a product of powers of one or more other measurement units.

For example the measurement unit “metre per second” for “velocity” is specified as the product of “metre” to the power one times “second” to the power minus one.

Properties

- factor: UnitFactor [1..*]
  Set of UnitFactor that specifies the product of powers of other measurement units that define the DerivedUnit.
- hasReducedFactors : Boolean[1] = true
  If true, the UnitFactors specifying the product of powers of other measurement units that define the DerivedUnit cannot be simplified. If false, the DerivedUnit is non-reduced; some UnitFactors can be simplified. A non-reduced DerivedUnit can have as a general unit other DerivedUnits defined in terms of simplified UnitFactors, possibly in reduced form.

Operations

[1] A DerivedUnit transitively depends on its factors' Units and all of the Units that its factors' Units depend on.

```
dependsOnUnits() : Unit[0..*] {unique}
body: factor.unit.dependsOnUnits()->flatten()->asSet()->union(factor.unit->flatten()
->asSet())->asSet()
```

[1] The query accessibleQuantityKinds() gives all the QuantityKinds directly defined in the SystemOfQuantities or transitively in any included or used SystemOfQuantities.

```
allAccessibleQuantityKinds() : QuantityKind[0..*] {unique}
body: allAccessibleSystemOfQuantities()->collect(quantityKind)->flatten()->asSet()
inv SoU3_3:
getEffectiveSystemOfQuantities() = null or let aqk : Set(QuantityKind) =
getEffectiveSystemOfQuantities().allQuantityKinds() in ->allUnits()
->forAll(u | aqk>includesAll (getKindOfQuantitiesForMeasurementUnit(u)))
```

E.5.2.5 Dimension

A Dimension represents the [VIM] concept of “quantity dimension” that is defined as “expression of the dependence of a quantity on the base quantities of a system of quantities as a product of powers of factors corresponding to the base quantities, omitting any numerical factor.”

For example in the ISQ the quantity dimension of “force” is denoted by dim F = L·M·T-2, where “F” is the symbol for “force,” and “L,” “M,” and “T” are the symbols for the ISQ base quantities “length,” “mass,” and “time” respectively.

The Dimension of any QuantityKind can be derived through the algorithm that is defined in E.5.2.15 SystemOfQuantities with SystemOfQuantities. The actual Dimension for a given QuantityKind depends on the choice of baseQuantityKind specified in a SystemOfQuantities.

Properties

- symbolicExpression: String [0..1]
  Symbolic expression of the quantity dimension's product of powers, in terms of symbols of the kinds of quantity

that represent the base kinds of quantity and their exponents. In tool implementations, the symbolicExpression may automatically derived from the associated factor set.

- factor: QuantityKindFactor [0..*] {ordered}
  If true Ordered set of QuantityKindFactor that specifies the product of powers of base dimensions that define the Dimension. The possible base dimensions are represented by the ordered set of baseQuantityKind defined in the SystemOfQuantities for which the Dimension is specified. The order of the factors should follow the ordered set of baseQuantityKind in SystemOfQuantities.

E.5.2.6 GeneralConversionUnit

Description

A GeneralConversionUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another reference measurement unit through a conversion relationship expressed in some syntax through a general mathematical expression.

The unit conversion relationship is defined by the following equation:

$value_{RU} / value_{CU} = f(value_{RU}, value_{CU})$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit and

$value_{CU}$ is the quantity value expressed in the GeneralConversionUnit and

$f(value_{RU}, value_{CU})$ is a mathematical expression that includes $value_{RU}$ and $value_{CU}$

Properties

- expression: String
  Specifies the unit conversion relationship in some expression syntax.
- expressionLanguageURI: String [0..1]
  URI that specifies the language for the expression syntax.

E.5.2.7 LinearConversionUnit

Description

A LinearConversionUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another measurement reference unit through a linear conversion relationship with a conversion factor.

The unit conversion relationship is defined by the following equation:

$value_{RU} = factor \cdot value_{CU}$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit, and,

$value_{CU}$ is the quantity value expressed in the LinearConversionUnit.

For example, in the definition of the LinearConversionUnit for “inch” with respect to the referenceUnit “metre,” the factor would be 254/10000, because 0.0254 metre = 1 inch.

Properties

- factor: Number
  Number that specifies the factor in the unit conversion relationship.

E.5.2.8 Prefix

Description

A Prefix represents a named multiple or submultiple multiplication factor used in the specification of a PrefixedUnit. A SystemOfUnits may specify a set of prefixes.

Properties

- symbol: String [0..1]
  Short symbolic name of the prefix.
- factor: Rational [1]
  Specifies the multiple or submultiple multiplication factor.

E.5.2.9 PrefixedUnit

Description

A PrefixedUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another measurement reference unit through a linear conversion relationship with a named prefix that represents a multiple or submultiple of a unit.

[VIM] defines “multiple of a unit” as “measurement obtained by multiplying a given measurement unit by an integer greater than one” and “submultiple of a unit” as “measurement unit obtained by dividing a given measurement unit by an integer greater than one.”

The unit conversion relationship is defined by the following equation:

$value_{RU} = factor \cdot value_{CU}$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit and

$value_{CU}$ is the quantity value expressed in the PrefixedUnit.

For example, in the definition of the PrefixedUnit for “megabyte” with respect to the referenceUnit “byte,” the prefix would be the Prefix for “mega” with a factor 106, because 106 byte = 1 megabyte.

See [VIM] for all decimal and binary multiples and decimal submultiples defined in SI.

Properties

- prefix: Prefix
  Specifies the prefix that defines the name, symbol, and factor of the multiple or submultiple.

Constraints

[1] The referenceUnit shall not be a PrefixedUnit, i.e., it is not allowed to prefix an already prefixed measurement unit. In general the referenceUnit should be a SimpleUnit.

```
package QUDV
context PrefixedUnit
inv: not referenceUnit.oclIsTypeOf(PrefixedUnit)
endpackage
```

E.5.2.10 QuantityKind

Description

In QUDV, the concept of QuantityKind is an abstract specialization of SysML QuantityKind to support designating a primary QuantityKind for a given Unit within the scope of a system of units and quantities and to support a richer vocabulary for defining QuantityKinds.

Properties

- /dependsOnQuantityKinds : QuantityKind[0..*] {readOnly, unique}
  The set of all QuantityKinds that this QuantityKind directly or indirectly depends on according to its definition.
- general: QuantityKind[0..*] {unique}
  A quantity can be defined to represent a combination of specific characteristics from one or more aspects defined by general QuantityKinds (see ISO 80000-1, 3.2).
- isNumberOfEntities: Boolean = false
  If true, indicates that the QuantityKind represents a number of entities (see ISO 80000-1, 3.8, Note 4).
- isQuantityOfDimensionOne: Boolean = false
  If true, indicates that the QuantityKind has dimension one (see ISO 80000-1, 3.8).

Constraints

[1] A QuantityKind cannot be defined in terms of itself. This follows from the quantity calculus used for expressing a derived QuantityKind in terms of base QuantityKinds chosen for a SystemOfQuantities by means of non- contradictory equations (See ISO 80000-1, 4.3).

```
inv acyclic_quantity_kind_dependencies:
dependsOnQuantityKinds()->excludes(self)
```

Operations

[1] Abstract operation specified in SimpleQuantityKind and DerivedQuantityKind to calculate the value of the derived property QuantityKind:/dependsOnQuantityKinds.

```
dependsOnQuantityKinds() : QuantityKind[0..*] {unique}
```

E.5.2.11 QuantityKindFactor

Description

A QuantityKindFactor represents a factor in the product of powers that defines a DerivedQuantityKind.

Properties

- exponent: Rational
  Rational number that specifies the exponent of the power to which the quantityKind is raised.
- QuantityKind: QuantityKind
  Reference to the QuantityKind that participates in the factor.

E.5.2.12 Rational

Description

A Rational value type represents the mathematical concept of a number that can be expressed as a quotient of two integers. It may be used to express the exact value of such values, without issues of rounding or other approximations if the result of the division were used instead.

Properties

- numerator: Integer
  An integer number used to express the numerator of a rational number.
- denominator: Integer
  An integer number used to express the denominator of a rational number.

Operations

```
package QUDV
context Rational
def: plus(r : Rational[1]) : Rational[1]
= result.numerator = self.numerator * r.demonimator
+ r.numerator * self.denominator
and result.denominator = self.denominator * r.denominator



context Rational
def: equivalent(r : Rational[1]) : Boolean[1]
= result = ( self.numerator * r.demonimator
= r.numerator * self.denominator)



context Rational
def: times(r : Rational[1]) : Rational[1]
= result.numerator = self.numerator * r.numerator
and result.denominator = self.denominator * r.denominator
endpackage
```

Constraints

[1] The denominator of a rational number shall not be zero.

```
package QUDV
context Rational
```

```
inv: denominator <> 0
endpackage
```

E.5.2.13 SimpleQuantityKind

Description

A SimpleQuantityKind is a QuantityKind that represents a kind of quantity that does not depend on any other QuantityKind. Typically a base quantity would be specified as a SimpleQuantityKind.

Operations

[1] A SimpleQuantityKind does not depend on any other QuantityKind.

dependsOnQuantityKinds() : QuantityKind[0..*] {unique}
body: Set{}

E.5.2.14 SimpleUnit

Description

A SimpleUnit is a Unit that represents a measurement unit that does not depend on any other Unit. Typically, a base unit would be specified as a SimpleUnit.

Operations

[1] A SimpleUnit is a Unit that represents a measurement unit that does not depend on any other Unit. Typically, a base unit would be specified as a SimpleUnit.

dependsOnUnits() : Unit[0..*] {unique}
body: Set{}

E.5.2.15 SystemofQuantities

Description

A SystemOfQuantities represents the [VIM] concept of “system of quantities” that is defined as a “set of quantities together with a set of non-contradictory equations relating those quantities.” It collects a list of QuantityKind that specifies the kinds of quantity that are known in the system.

The International System of Quantities (ISQ) is an example of a SystemOfQuantities, defined in [ISO31] and [ISO/IEC80000].

Properties

- symbol: String [0..1]
  Short symbolic name of the system of quantities.
- description: String [0..1]
  Textual description of the system of quantities.
- definitionURI: String [0..1]
  URI that references an external definition of the system of quantities. Note that as part of [ISO/IEC80000] normative URIs for each of the ISQ quantities and SI units are being defined.

- quantityKind: QuantityKind [0..*] {ordered}
  Ordered set of QuantityKind that specifies the kinds of quantity that are known in the system.
- baseQuantityKind: QuantityKind [0..*] {ordered, subsets quantityKind}
  Ordered set of QuantityKind that specifies the base quantities of the system of quantities. This is a subset of the complete quantityKind list. The base quantities define the basis for the quantity dimension of a kind of quantity.
- /dimension: Dimension [0..*] {ordered, readOnly, nonunique}
  Derived ordered set of Dimension. The actual dimension of a QuantityKind depends on the list of baseQuantityKind that are specified in an actual SystemOfQuantities, see the DerivedDimensions constraint.
- includedSystemOfQuantities: SystemOfQuantities[0..*] {unique}
  Including a SystemOfQuantities means including all of the QuantityKind it defines and includes from other SystemOfQuantities.

- usedSystemOfQuantities: SystemOfQuantities[0..*] {unique}
  A QuantityKind can be defined in a SystemOfQuantities in terms of QuantityKinds defined in that SystemOfQuantities or from other SystemOfQuantities it uses or includes. See for example the units used with the SI in ISO 80000-1, Table 5.

Constraints

[1] All quantity dimensions are derived through the following algorithm specified in OCL.

```
package QUDV
-- get the set of units, if any, that a given unit directly depends on
context Unit
def: directUnitDependencies : Set(Unit) =
if oclIsKindOf(ConversionBasedUnit)
then oclAsType(ConversionBasedUnit).referenceUnit
else
if oclIsKindOf(DerivedUnit)
then oclAsType(DerivedUnit).factor->collect(unit)->asSet()
else Set{}
endif
endif


-- get the set of units, if any, that a given unit directly or indirectly depends on
context Unit
def: allUnitDependencies : Set(Unit)
= self->closure(directUnitDependencies)


context Unit
```

```
inv acyclic_unit_dependencies
: not allUnitDependencies->excludes(self)

-- get the set of quantityKinds, if any, that a given quantityKind directly depends on
context QuantityKind
def: directQKindDependencies : Set(QuantityKind)
= if oclIsKindOf(DerivedQuantityKind)
then oclAsType(DerivedQuantityKind).factor
->collect(quantityKind)->asSet()
else
if oclIsKindOf(SpecializedQuantityKind)
then oclAsType(SpecializedQuantityKind).general
else Set{}
endif
endif
context QuantityKind
def: allQuantityKindDependencies : Set(QuantityKind)
= self->closure(directQKindDependencies)

context QuantityKind
inv acyclic_quantity_kind_dependencies
: allQuantityKindDependencies->excludes(self)

--context SystemOfQuantities::deriveQuantityKindDimensions() :
--post: quantityKind->forAll(qK|qK.hasProperDimension(self))
-- The derived dimension of a simple quantity kind must
-- have exactly one factor
-- whose numerator and denominator are equal to 1.

context SimpleQuantityKind
def: hasProperDimension(sq:SystemOfQuantities) : Boolean
= let d:Dimension=sq.getDimension(self)
in d.factor->size()=1
```

```
and d.factor->forAll(exponent->forAll(numerator=1 and denominator=1))
-- The derived dimension of a specialized quantity kind is
-- the dimension of its general quantity kind.
context SpecializedQuantityKind
def: hasProperDimension(sq:SystemOfQuantities) : Boolean
= sq.getDimension(self) = sq.getDimension(general)
-- A helper function to produce the factor/quantityKind tuples
-- for a given Dimension.
context Dimension
def: dimFactors : Bag(Tuple(factor:Rational,qKind:QuantityKind))
= self.factor->collect(qkf | Tuple{factor=qkf.exponent,qKind=qkf.quantityKind})
-- A helper function to get all the factor/quantityKind tuples
-- for the dimension factors of a derived quantity kind.
context DerivedQuantityKind
def: derQFactors(sq:SystemOfQuantities) : Bag(Tuple(factor:Rational,qKind:QuantityKind))
= self.factor->collect(qkf |
let qd:Dimension = sq.getDimension(qkf.quantityKind) in
qd.factor->collect(qkf |
Tuple{factor=qkf.exponent.plus(df.exponent),qKind=qkf.quantityKind}))
-- Reduce a bag of factor/quantityKind tuples by combining
-- all factors for the same quantity kind
-- and eliminating the zero-factor/quantityKind tuples
context DerivedQuantityKind
def: reducetoNonZeroUniqueFactors(
qFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind)),
qKinds:Set(QuantityKind))
: Bag(Tuple(factor:Rational,qKind:QuantityKind))
= let uqFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
= qKinds->collect(
-- for each unique quantity kind, qKind1,
-- from the set of unique quantity kinds, qKinds...
qKind1:QuantityKind|
-- get the sequence of factors from the set of
-- qFactors tuples whose quantity kind is qKind1...
let factor1s:Sequence(Rational)
```

```
= qFactors->select(qKind=qKind1)
->collect(factor)->asSequence()
-- start with the first factor, factor1,
-- from all the factor1s associated to qKind1...
in let factor1:Rational=factor1s->first()
-- construct the factor/quantityKind tuple
-- for qKind1 where
-- the factor is the product of factor1 with
-- all remaining factors1s
in Tuple{
factor=factor1s->excluding(factor1)->iterate(
factorI:Rational;
factorN:Rational=factorI |
factorN.plus(factorI)),
qKind=qKind1})
-- eliminate the factor/quantityKind tuples where
-- the factor is zero
in let nqFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
= uqFactors->select(factor.numerator<>0)
in nqFactors


-- The derived dimension of a derived quantity kind is
-- the simplified set of factor/quantityKind tuples
-- for the derived quantity kind. The simplified set
-- of factor/quantityKind tuples has
-- one factor/quantityKind tuple for each quantityKind where
-- the simplified factor is a non-zero product of
-- all the factors in the factor/quantityKind tuples.
context DerivedQuantityKind
def: hasProperDimension(sq:SystemOfQuantities) : Boolean
= let d:Dimension = sq.getDimension(self)
in let resFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
= d.dimFactors
-- the unique quantityKinds from the result...
```

```
in let resKinds:Set(QuantityKind)
=resFactors->collect(qKind)->asSet()
-- the factor/quantityKind tuples from the derived quantity...
in let qFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
= self.derQFactors(sq)
-- the unique quantityKinds from the derived quantity...
in let qKinds:Set(QuantityKind)=qFactors->collect(qKind)->asSet()
-- get the reduced non-zero factor/quantityKinds...
in let nqFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
= self.reducetoNonZeroUniqueFactors(qFactors, qKinds)
-- condition1: there should be the same number
-- of factor/quantityKind tuples in the result
-- compared to the non-zero unique factor/quantityKind
-- tuples for the derivedQuantityKind
in nqFactors->size() = resFactors->size()
-- condition2: there should be the same set of
-- quantity kinds in the result
-- and in the non-zero unique factor/quantityKind tuples
-- and qKinds->symmetricDifference(resKinds)->isEmpty()
-- condition3: for each quantity kind,
-- the factors in the result and
-- in the reduced non-zero unique factor/quantityKind
-- tuples should be equivalent rationals
and qKinds->forAll(qk:QuantityKind|
let nFactor:Rational
=nqFactors->select(qKind=qk)
->collect(factor)->asSequence()->first()
in let rFactor:Rational
=resFactors->select(qKind=qk)
->collect(factor)->asSequence()->first()
in nFactor.equivalent(rFactor))
endpackage
```

[2] For a QuantityKind to have a provenance to a single SystemOfQuantities, all included systems of quantities shall be transitively disjoint with all used systems of quantities.

```
inv includedSystemOfQuantities_transitivelyDisjoint_usedSystemOfQuantities:
allIncludedSystemOfQuantities()->intersection(self.oclAsSet()
->closure(usedSystemOfQuantities))->isEmpty()
```

[3] The set of all QuantityKinds in a given SystemOfQuantities shall be partitioned into two disjoint, covering subsets: the set of base QuantityKinds (typically chosen to be mutually independent) and its complement, the set of derived QuantityKinds, each of which can be expressed in terms of the base QuantityKinds (See ISO 80000-1, 4.3).

```
inv allBaseQuantitiesAreQuantities:
allQuantityKinds()->includesAll(allBaseQuantityKinds())
```

[4] Every QuantityKind shall be defined in only one SystemOfQuantities but it can be in the scope of several SystemOfQuantities. A given QuantityKind is in scope of a SystemOfQuantities either because it is defined or used in a SystemOfQuantities or because it is included from the scope of another SystemOfQuantities.

```
inv singleProvenance:
includedSystemOfQuantities->collect(allQuantityKinds())
->intersection(quantityKind)->isEmpty()
```

[5] For a QuantityKind to have a provenance to a single SystemOfQuantities, the use and includes relationships among SystemOfQuantities shall be acyclic.

```
inv acyclicProvenance:
allAccessibleSystemOfQuantities()->excludes(self)
```

Operations

[1] The query accessibleQuantityKinds() gives all the QuantityKinds directly defined in the SystemOfQuantities or transitively in any included or used SystemOfQuantities.

```
allAccessibleQuantityKinds() : QuantityKind[0..*] {unique}
body: allAccessibleSystemOfQuantities()->collect(quantityKind)->flatten()->asSet()
```

[2] The query allAccessibleSystemOfQuantities() gives all the SystemOfQuantities directly or transitively included or used.

```
allAccessibleSystemOfQuantities() : SystemOfQuantities[0..*] {unique}
body: self->closure(includedSystemOfQuantities->union(usedSystemOfQuantities))
->asSet()
```

[3] The query allBaseQuantityKinds() gives all the QuantityKinds directly adopted or transitively adopted from any included SystemOfQuantities as base QuantityKinds.

```
allBaseQuantityKinds(): QuantityKind[0..*] {unique}
body: allIncludedSystemOfQuantities()->collect(baseQuantityKind)->flatten()->asSet()
->union(baseQuantityKind)->asSet()
```

[4] The query allIncludedSystemOfQuantities() gives all the SystemOfQuantities directly or transitively included.

```
allIncludedSystemOfQuantities () : SystemOfQuantities[0..*] {unique}
body: self->closure(includedSystemOfQuantities)->asSet()
```

[5] The query allQuantityKinds() gives all the QuantityKinds in scope of a SystemOfQuantities; that is, each QuantityKind is either directly defined in the SystemOfQuantities, selectively used from another SystemOfQuantities or part of the scope of all the QuantityKinds included from another SystemOfQuantities.

```
allQuantityKinds(): QuantityKind[0..*] {unique}
body: allIncludedSystemOfQuantities()->collect(quantityKind)->flatten()->asSet()
->union(quantityKind)->asSet())
```

E.5.2.16 SystemofUnits

Description

A SystemOfUnits represents the [VIM] concept of "system of units" that is defined as "set of base units and derived units, together with their multiples and submultiples, defined in accordance with given rules, for a given system of quantities." It collects a list of Units that are known in the system. A QUDV SystemOfUnits only optionally defines multiples and submultiples.

Properties

- symbol: String [0..1]
  Short symbolic name of the system of units.
- description: String [0..1]
  Textual description of the system of units.
- definitionURI: String [0..1]
  A URI that references an external definition of the system of units. Note that as part of [ISO/IEC80000] normative URIs for each of the quantities in the ISQ and units in the SI are being defined.
- unit: Unit [0..*] {ordered}
  Ordered set of Unit that specifies the units that are known in the system.
- baseUnit: Unit [0..*] {ordered, subsets unit}

  Ordered set of Unit that specifies the base units of the system of units. A "base unit" is defined in [VIM] as a "measurement unit that is adopted by convention for a base quantity." It is the (preferred) unit in which base quantities of the associated systemOfQuantities are expressed.
- prefix: Prefix [0..*] {ordered}

Ordered set of Prefix that specifies the prefixes for multiples and submultiples of units in the system.

- systemOfQuantities: SystemOfQuantities [0..1]
  Reference to the SystemOfQuantities for which the units are specified.
- includedSystemOfUnits: SystemOfUnits[0..*] {unique}

Including a SystemOfQuantities means including all of the QuantityKind it defines and includes from other SystemOfQuantities.

- usedSystemOfUnits: SystemOfUnits[0..*] {unique}
  A Unit can be defined in a SystemOfUnits in terms of Units defined in that SystemOfUnits or from other SystemOfUnits it uses or includes. See for example the units used with the SI in ISO 80000-1, Table 5.

Constraints

[1] In a coherent system of units, there shall be only one base unit for each base quantity.

```
package QUDV
context SystemOfUnits
def: isCoherent() : Boolean =
baseUnit->size() = systemOfQuantities.baseQuantityKind->size()
and baseUnit
->forAll(bU|systemOfQuantities.baseQuantityKind
->one(bQK|bU.primaryQuantityKind=bQK))
and systemOfQuantities.baseQuantityKind
->forAll(bQK|baseUnit->one(bU|bQK=bU.primaryQuantityKind))
Endpackage
```

[2] A coherent derived unit shall be a derived unit that, for a given system of quantities and for a chosen set of base units, is a product of powers of base units with no other proportionality factor than one.

```
package QUDV
context SystemOfUnits
def: isCoherent(du : DerivedUnit) : Boolean =
baseUnit->includesAll(du.factor->collect(unit))
and du.factor->collect(exponent)
->forAll(numerator=1 and denominator=1)
Endpackage
```

[3] In a well-formed SystemOfUnits, all of the prefixes of PrefixedUnits shall be defined in the SystemOfUnits.

```
inv SoU3_1:
allPrefixes()->includesAll(allUnits()->select(oclIsTypeOf(PrefixedUnit))
->collect(oclAsType(PrefixedUnit).prefix))
```

[4] All the dependent Units of a SystemOfUnits shall be in the scope of that SystemOfUnits.

```
inv SoU3_2:
```

```
allUnits()->includesAll(allUnits()->collect(dependsOnUnits())->flatten()->asSet())
```

[5] All of the quantityKinds that are measurementUnits of Units in the SystemOfUnits shall be defined in the systemOfQuantities of that SystemOfUnits.

```
inv SoU3_3:

getEffectiveSystemOfQuantities() = null or let aqk : Set(QuantityKind) =

getEffectiveSystemOfQuantities().allQuantityKinds() in ->allUnits()

->forAll(u | aqk
->includesAll(getKindOfQuantitiesForMeasurementUnit(u)))
```

[6] For a Unit to have a provenance to a single SystemOfUnits, all included systems of units shall be transitively disjoint with all used systems of units.

```
inv includedSystemOfUnits_transitivelyDisjoint_usedSystemOfUnits:

allIncludedSystemOfUnits()->intersection(self.oclAsSet()

->closure(usedSystemOfUnits))->isEmpty()
```

[7] The set of all Units in a given SystemOfUnits shall be capable of being partitioned into two disjoint, covering subsets: the set of base Units (typically chosen to be mutually independent) and all its complement, the set of derived Units, each of which can be expressed in terms of the base Units (See ISO 80000-1, 6.4).

```
inv allBaseUnitsAreUnits:

allUnits()->includesAll(allBaseUnits())
```

[8] Every Unit shall be defined in only one SystemOfUnits but it can be in the scope of several SystemOfUnits. A given Unit is in scope of a SystemOfUnits either because it is defined or used in a SystemOfUnits or because it is included from the scope of another SystemOfUnits.

```
inv singleProvenance:

includedSystemOfUnits->collect(allUnits())->intersection(unit)->isEmpty())
```

[9] For a Unit to have a provenance to a single SystemOfUnits, the use and includes relationships among SystemOfUnits shall be acyclic.

```
inv acyclicProvenance:

allAccessibleSystemOfUnits()->excludes(self)
```

Operations

[1] The query accessibleQuantityKinds() gives all the QuantityKinds directly defined in the SystemOfQuantities or transitively in any included or used SystemOfQuantities.

```
allAccessibleQuantityKinds() : QuantityKind[0..*] {unique}
body: allAccessibleSystemOfQuantities()->collect(quantityKind)->flatten()->asSet()
inv SoU3_3:
getEffectiveSystemOfQuantities() = null or let aqk : Set(QuantityKind) =
getEffectiveSystemOfQuantities().allQuantityKinds() in ->allUnits()
->forAll(u | aqk
->includesAll(getKindOfQuantitiesForMeasurementUnit(u)))
```

[2] The query allAccessibleSystemOfUnits() gives all the SystemOfUnits directly or transitively included or used.

```
allAccessibleSystemOfUnits(): SystemOfUnits[0..*] {unique}
body: self->closure(includedSystemOfUnits->union(usedSystemOfUnits))->asSet()
```

[3] The query accessibleUnits () gives all the units directly defined in a system of units or transitively in any included or used system of units.

```
allAccessibleUnits(): Unit[0..*] {unique}
body: allAccessibleSystemOfUnits()->collect(unit)->flatten()->asSet()
```

[4] The query allBaseQuantityKinds() gives all the QuantityKinds directly adopted or transitively adopted from any included SystemOfQuantities as base QuantityKinds in the effective SystemOfQuantities associated to a SystemOfUnits.

```
allBaseQuantityKinds(): QuantityKind[0..*] {unique}
body: getEffectiveSystemOfQuantities()->allBaseQuantityKinds()->flatten()->asSet()
```

[5] The query allBaseUnits() gives all the Units directly adopted or transitively adopted from any included SystemOfUnits as base Units.

```
allBaseUnits(): Unit[0..*] {unique}
body: allIncludedSystemOfUnits()->collect(baseUnit)->flatten()->asSet()
->union(baseUnit)->asset
```

[6] The query allIncludedSystemOfUnits() gives all the SystemOfUnits directly or transitively included.

```
allIncludedSystemOfUnits(): SystemOfUnits[0..*] {unique}
body: self->closure(includedSystemOfUnits->union(usedSystemOfUnits))->asSet()
```

[7] The predicate allMeasurementUnitsDefinedForSomeQuantityKind() determines whether, in a SystemOfUnits, every Unit shall be defined, by convention, as a multiplicable reference for at least one QuantityKind (see ISO 80000-1, 3.9).

```
allMeasurementUnitsDefinedForSomeQuantityKind(): Boolean
body: allUnits()->select(quantityKind <> null)
```

[8] The query allPrefixes() gives all the Prefixes in scope of a SystemOfUnits; that is, each Prefix is either directly defined in the SystemOfUnits or in any accessible SystemOfUnits.

```
allPrefixes(): Prefix[0..*] {unique}

body: allAccessibleSystemOfUnits()->including(self)->collect(prefix)->flatten()
->asSet()
```

[9] The query allUnits() gives all the Units in scope of a SystemOfUnits; that is, each Unit is either directly defined in the SystemOfUnits, selectively used from another SystemOfUnits or part of the scope of all the Units included from another SystemOfUnits.

```
allUnits(): Unit[0..*] {unique}

body: allIncludedSystemOfUnits()->collect(unit)->flatten()->asSet()->union(unit)
->asSet()
```

[10] The query getAdoptedBaseUnitForMeasurementUnit() determines for a Unit u in scope of a SystemOfUnits the base Unit, if any, corresponding to u, which can be u itself if it is a baseUnit in that SystemOfUnits or its reference Unit if it is a base Unit and u is a PrefixUnit.

```
getAdoptedBaseUnitForMeasurementUnit(u : Unit) : Unit[0..1]

body: let abu : Set(Unit) = allBaseUnits() in

if (abu->includes(u)) then u

else if (u.oclIsKindOf(PrefixedUnit))

then abu->intersection(u.oclAsType(PrefixedUnit).referenceUnit->asSet())
->any(true)

else null endif

endif
```

[11] The query getAdoptedQuantityKindForAdoptedBaseUnitOfMeasurementUnit() determines for a Unit u in scope of a SystemOfUnits the base QuantityKind, if any, corresponding to the base Unit of u.

```
getAdoptedQuantityKindForAdoptedBaseUnitOfMeasurementUnit (u : Unit) :
QuantityKind[0..1]

body: let bu : Unit = getAdoptedBaseUnitForMeasurementUnit(u) in

if (bu = null) then Set{}

else let qks : Set(QuantityKind) = getKindOfQuantitiesForMeasurementUnit(bu) in

allBaseQuantityKinds()->intersection(qks)

endif
```

[12] The query getEffectiveSystemOfQuantities() determines for a SystemOfUnits the SystemOfQuantities, if any, that it is directly or indirectly associated with via included SystemOfUnits.

```
getEffectiveSystemOfQuantities () : SystemOfQuantities[0..1]
```

```
body: if systemOfQuantities = null then includedSystemOfUnits->
collect(getEffectiveSystemOfQuantities())->flatten()->asSet()->any(true)
else systemOfQuantities endif
```

[13] The query getKindOfQuantitiesForMeasurementUnit() determines for a Unit u in scope of a SystemOfUnits the set of QuantityKinds corresponding to u, if specified, or to the Units that u is defined in terms of, if any.

```
getKindOfQuantitiesForMeasurementUnit(u : Unit) : QuantityKind[0..*] {unique}
body: let bu : Unit = getAdoptedBaseUnitForMeasurementUnit(u) in
if (bu = null) then Set{}
else let qks : Set(QuantityKind) = getKindOfQuantitiesForMeasurementUnit(bu) in
allBaseQuantityKinds()->intersection(qks)
endif
```

E.5.2.17 Unit

Description

In QUDV, the concept of Unit is an abstract specialization of SysML Unit to support designating a primary QuantityKind for a given Unit within the scope of a system of units and quantities and to support a richer vocabulary for defining Units.

Properties

- /dependsOnUnits : Unit[0..*] {readOnly, unique}
  The set of all Units that this Unit directly or indirectly depends on according to its definition.
- general: Unit[0..*] {unique}
  A Unit can be defined as a specialization of zero or more Units. This capability is important for specifying the meaning of a unit for a quantity of dimension one (see ISO 80000-1, 3.8 and 3.10).
- isUnitCountOfEntities: Boolean = false
  If true, indicates that the measurement unit represents a number of entities (see ISO 80000-1, 3.10, Note 3).
- isUnitForQuantityOfDimensionOne: Boolean = false
  If true, indicates that the corresponding QuantityKind has dimension one (see ISO 80000-1, 3.8).

Constraints

[1] A Unit cannot be defined in terms of itself. This follows from the requirement that, in a coherent SystemOfUnits, the Units of all derived QuantityKinds are expressed in terms of the base Units in accordance with the equations in the SystemOfQuantities (see ISO 80000-1, 6.4).

```
inv acyclic_unit_dependencies:
dependsOnUnits()->excludes(self)
```

Operations

[1] Abstract operation specified in SimpleQuantityKind and DerivedQuantityKind to calculate the value of the derived property QuantityKind:/dependsOnQuantityKinds.

```
dependsOnQuantityKinds() : QuantityKind[0..*] {unique}
```

E.5.2.18 UnitFactor

Description

A UnitFactor represents a factor in the product of powers that defines a DerivedUnit.

Properties

- exponent: Rational
  Rational number that specifies the exponent of the power to which the unit is raised.
- unit: Unit
  Reference to the Unit that participates in the factor.

E.5.3 References

[VIM]

JCGM 200:2012, International vocabulary of metrology - Basic and general concepts and associated terms (VIM), 3rd edition (JCGM 200:2008 with minor corrections), 2012, BIPM, Paris, France. http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf.

[ISO/IEC80000]

ISO/IEC 80000, Quantities and units. 15 parts, some published, some still in progress, harmonized replacement of [ISO31] and [IEC60027], the new international system of quantities and units.

[ISO31]

ISO 31, Quantities and units (Third edition 1992-08-01). Specifies the international system of units - SI - in 14 parts.

[IEC60027]

IEC 60027-2:2005, Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics (Third edition 2005-08).

[SI-Brochure]

Le Système international dunités (SI) / The International System of Units (SI), 8th edition 2006, BIPM, (French and English). Available for download in PDF format from http://www.bipm.org/en/si/si_brochure.

[NIST330]

The International System of Units (SI), NIST Special Publication 330, 2008 Edition. NOTE: U.S. version of the English language text of [SI-Brochure].

Available for download in PDF format from http://physics.nist.gov/cuu/Units/bibliography.html.

[NIST822]

Guide for the Use of the International System of Units (SI), NIST Special Publication 811, 2008 Edition.

Available for download in PDF format from http://physics.nist.gov/cuu/Units/bibliography.html.

[Dybkaer-2010] Rene Dybkaer, “ISO terminological analysis of the VIM3 concepts of ‘quantity’ and ‘kind-of-quantity’”, Metrologia 47, (2010) 127-143, http://dx.doi.org/10.1088/0026-1394/47/3/003. See also: http://www.bipm.org/en/publications/guides/rationale_vim3.html.

E.5.4 Usage Examples

E.5.4.1 SI Unit and QuantityKind examples

Figure E.8 shows an approach for defining base units of the System International of Units defined in http://www.bipm.org/en/si/si_brochure/chapter2/2-1/ and http://physics.nist.gov/cuu/Units/units.html. This approach involves instantiating the concrete classes of Unit shown in Figure E.6.

Figure E.9diagram shows the definition of “newton” as a DerivedUnit (E.5.2.4 DerivedUnit) corresponding to the “force” DerivedQuantityKind (E.5.2.3 DerivedQuantityKind). Derived units and quantity kinds are defined as products of factors on other units and quantity kinds respectively. In the QUDV, the product factors of a DerivedUnit (resp. DerivedQuantityKind) are all of the UnitFactor (resp. DerivedUnitFactor) at the “factor” ends of association link instances.

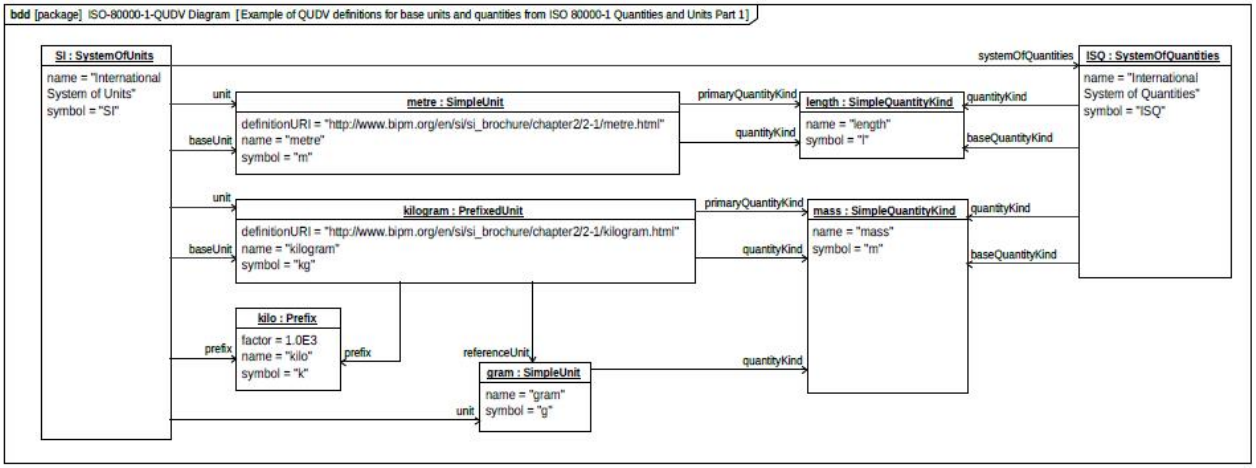


Figure E.8 Base Unit and Quantity Kinds of the SI and ISQ respectively

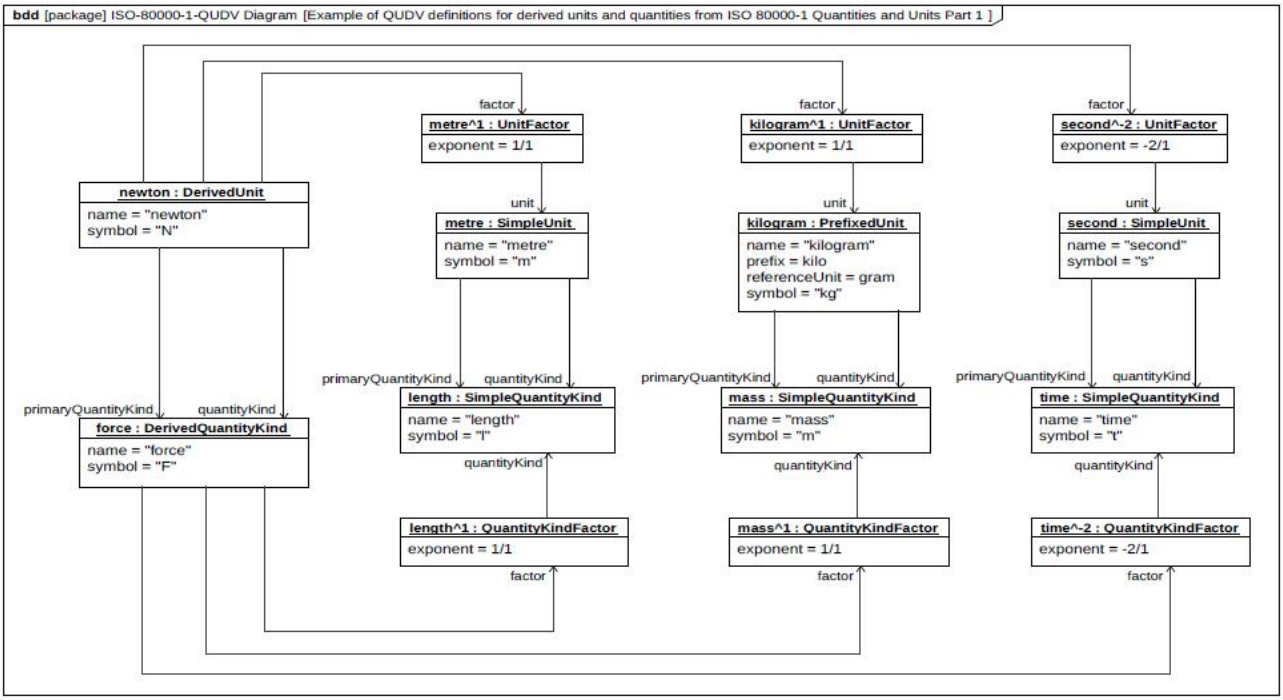


Figure E.9 Example of a derived unit and derived quantity kind

E.5.4.2 Spring Example

Figure E.10 shows a simple model of the length of a spring defined as the linear distance between the linear position of its two flange ends. QUDV supports defining arbitrary systems of units and quantities. Although this example uses only one unit, “metre” and one quantity kind, “lengthQK;” this example illustrates specialized value types to make additional distinctions such as “LinearPosition” vs. “LinearDistance,” two distinct quantities that have the same unit and quantity kind. This example illustrates an instance of a spring and uses the dot pathname property notation defined for IBDs (8.3.1.2, Internal Block Diagram) to clearly indicate the role of each instance specification.

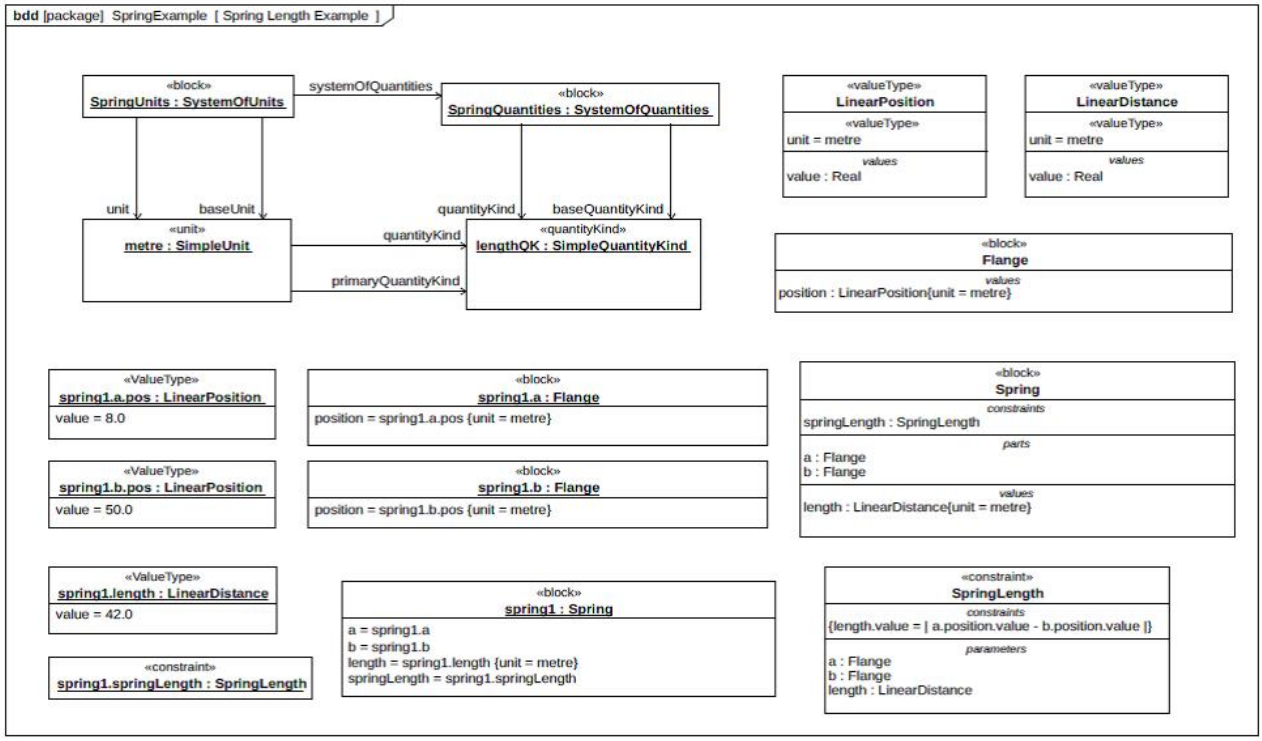


Figure E.10 Spring Length Example

E.6 Model Library of SysML Quantity Kinds and Units for ISO 8000

E.6.1 Overview

This non-normative extension defines a model library of SysML QuantityKind and Unit definitions for a subset of quantities and units defined by the International System of Quantities (ISQ) and the International System of Units (SI). The specific quantities and units in this library are defined by ISO 80000-1 Quantities and units - Part1: General. ISO/IEC 80000 currently has fourteen parts that define many quantities and units for use within various fields of science and technology. Part 1 defines base quantities and units used by other parts as well as a starting set of derived quantities and units with special names and symbols.

E.6.2 Units and Quantity Kinds

The model library defined in this sub clause contains SysML QuantityKind and Unit elements as defined by Clause 8, “Blocks.” Each QuantityKind or Unit element may optionally carry a “definitionURI” property to document each quantity kind and unit using additional information available from some external source. One option is for this definitionURI to identify an element of a QUDV model (see E.5, Model Library for Quantities, Units, Dimensions, and Values (QUDV)) that more fully describes the same quantities and units, including the systems of quantities and units they belong to, and the means by which they may be derived from each other.

E.5.4 Usage Examples contains examples of such QUDV definitions that could be referenced by these definitionURIs.

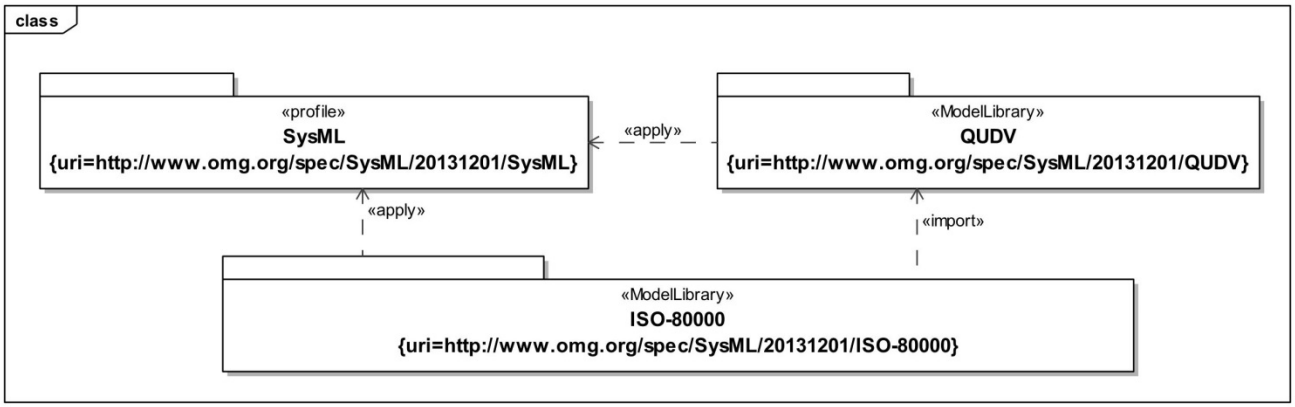


Figure E.11 Model libraries of SysML Quantity Kinds and Units for the covered content of ISO 80000 parts 3,4,5,6,7,9,10 and 13

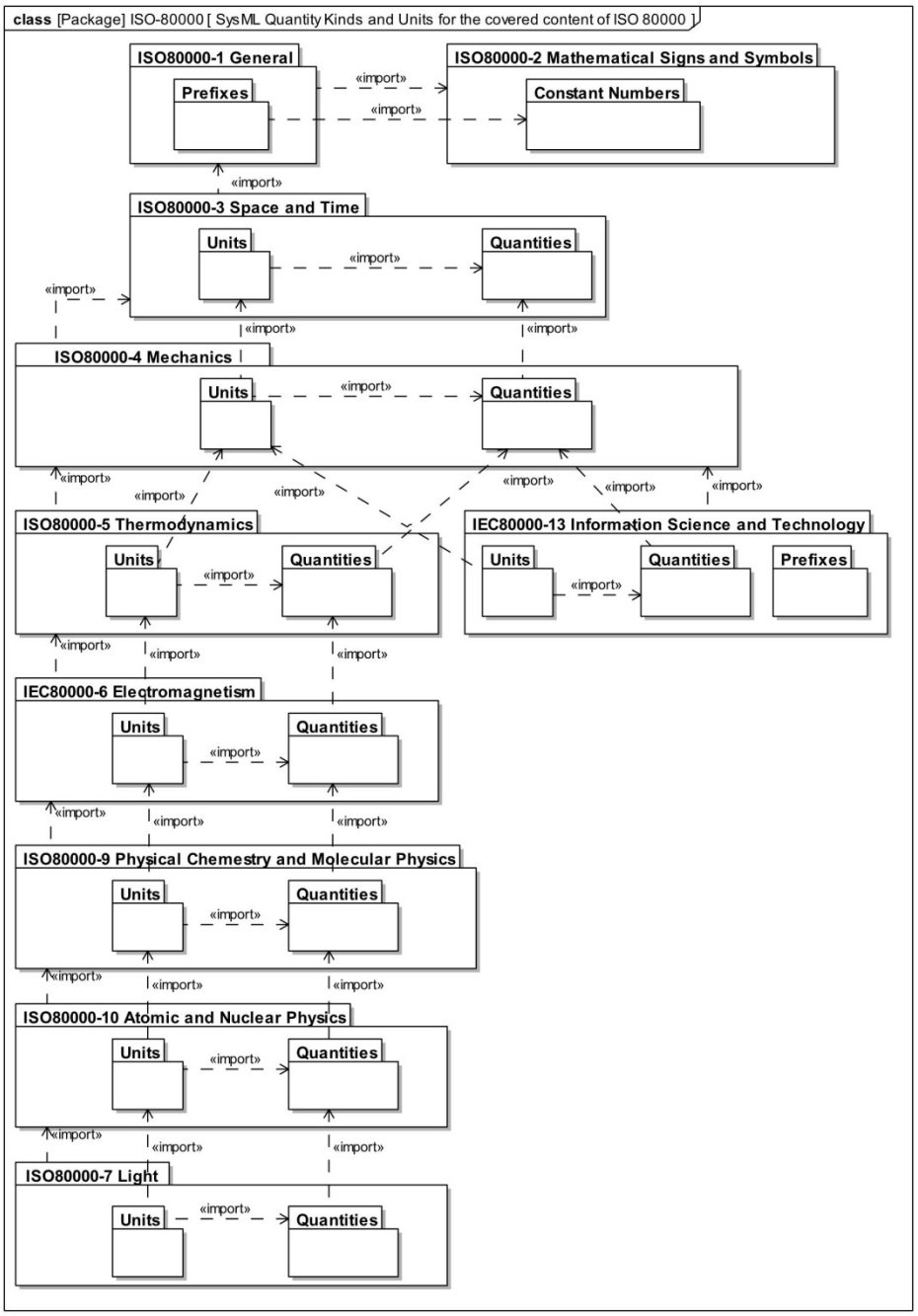


Figure E.12 Organization of the definitions of units and quantities from the normative parts of ISO 80000 covered in SysML 1.4, which includes all the normative content of parts 3,4,5,6; the subset of parts 7,9,10 corresponding to the content from SysML 1.3 and the subset of part 13 pertaining to commonly used units of information. Parts 8,11 and 12 are not covered because none of their units and quantities were referenced in previous versions of SysML nor in the summary tables in ISO 80000-1

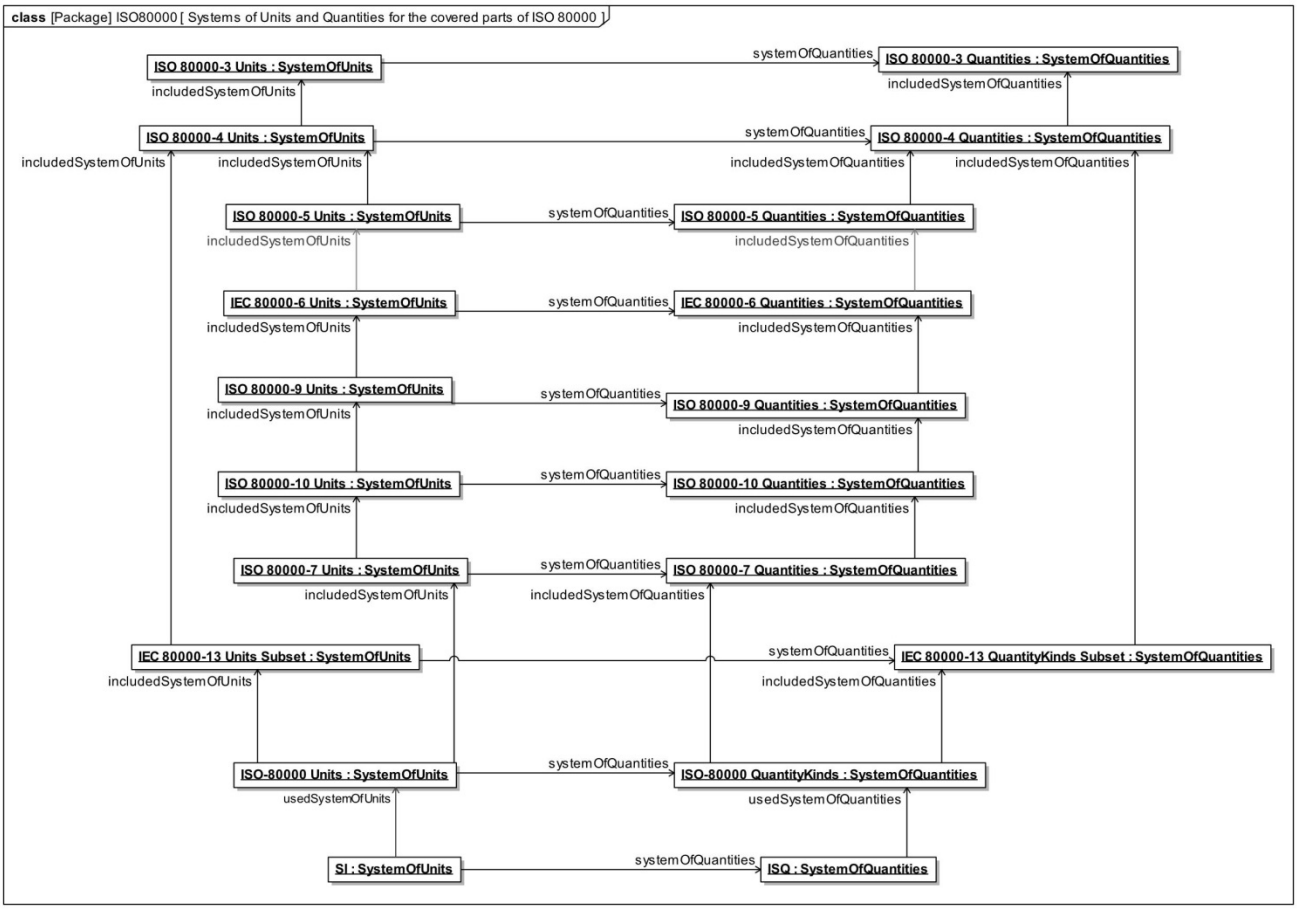


Figure E.13 Content relationships for the systems of units and quantities in from the different parts of ISO 80000 in relation to ISO 80000 as a whole and to the International System of Units (SI) and quantities (ISQ)

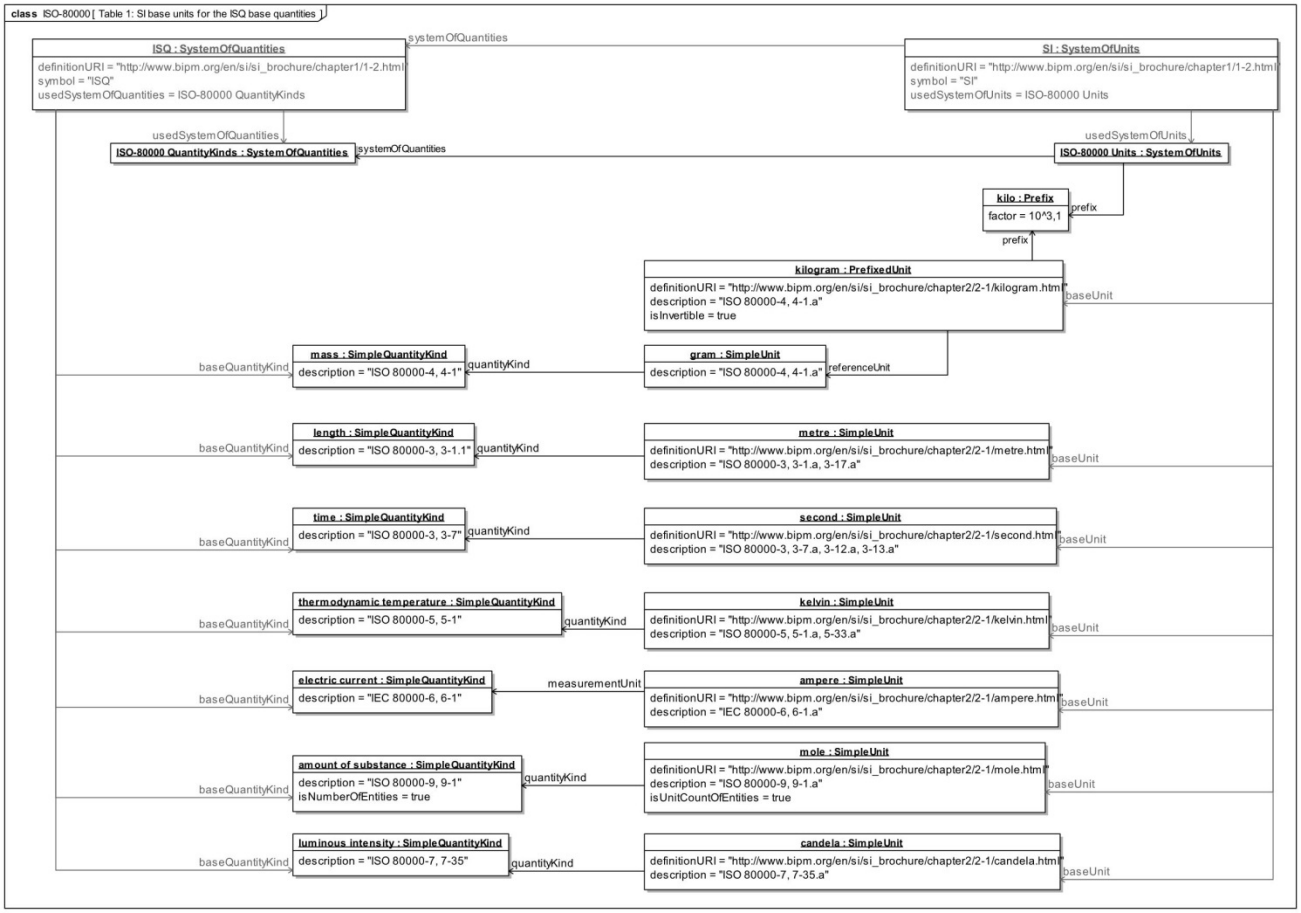


Figure E.14 Table 1 (from ISO 80000-1) SI base units for the ISQ base quantities

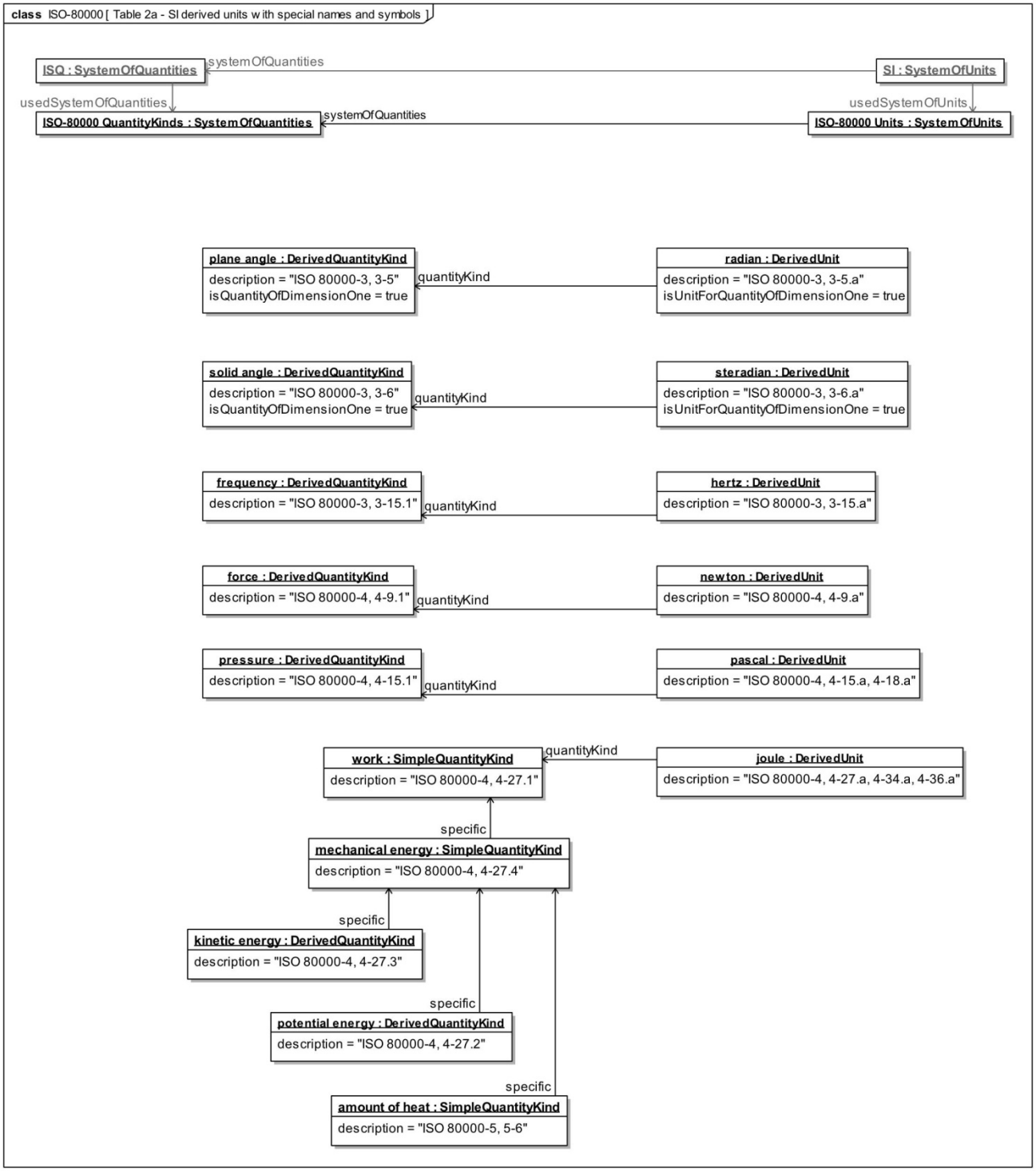


Figure E.15 Table 2 (from ISO 80000-1) ISQ derived quantities and SI derived units with special names (1)

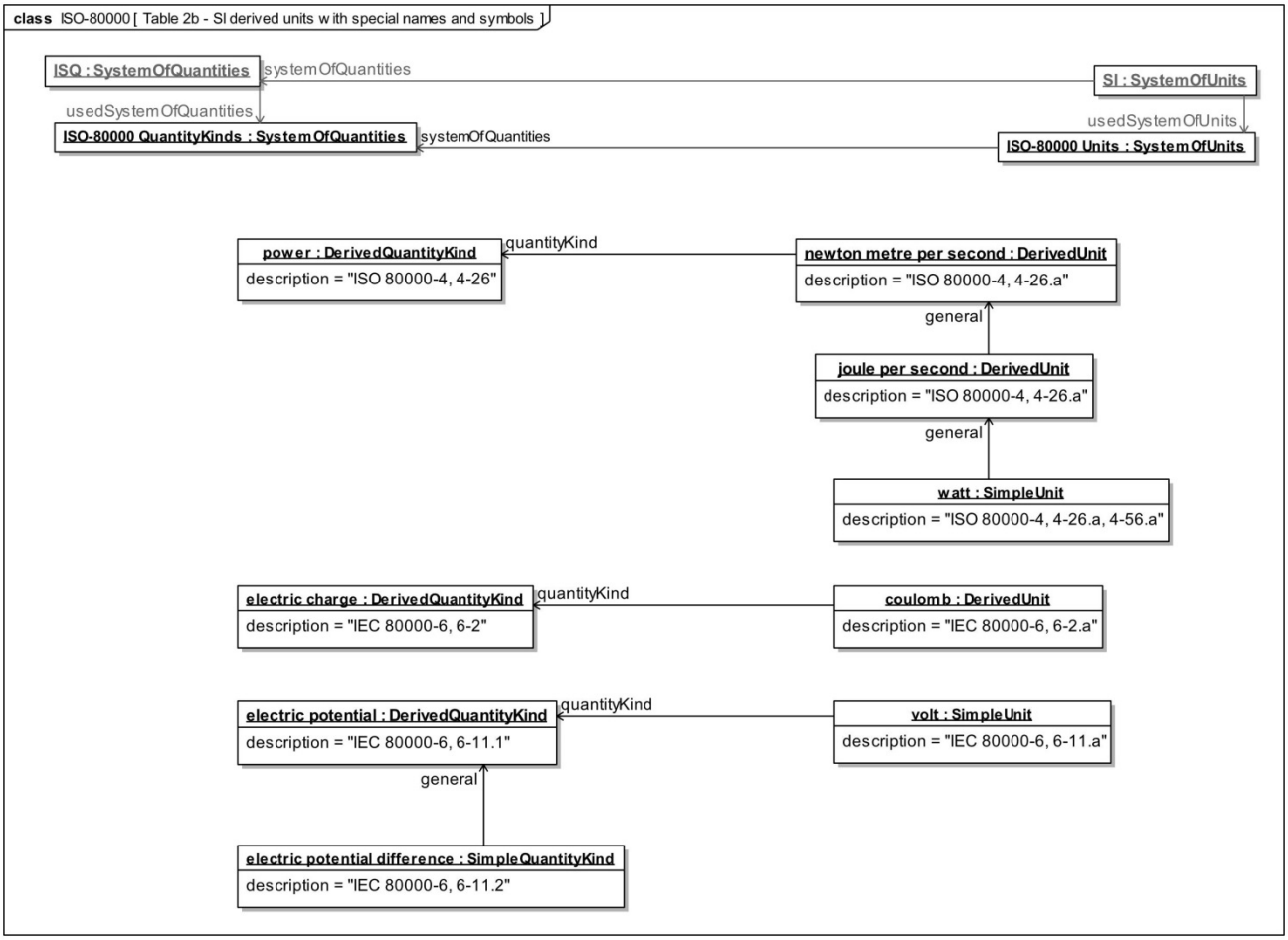


Figure E.16 Table 2 (from ISO 80000-1) ISQ derived quantities and SI derived units with special names (2)

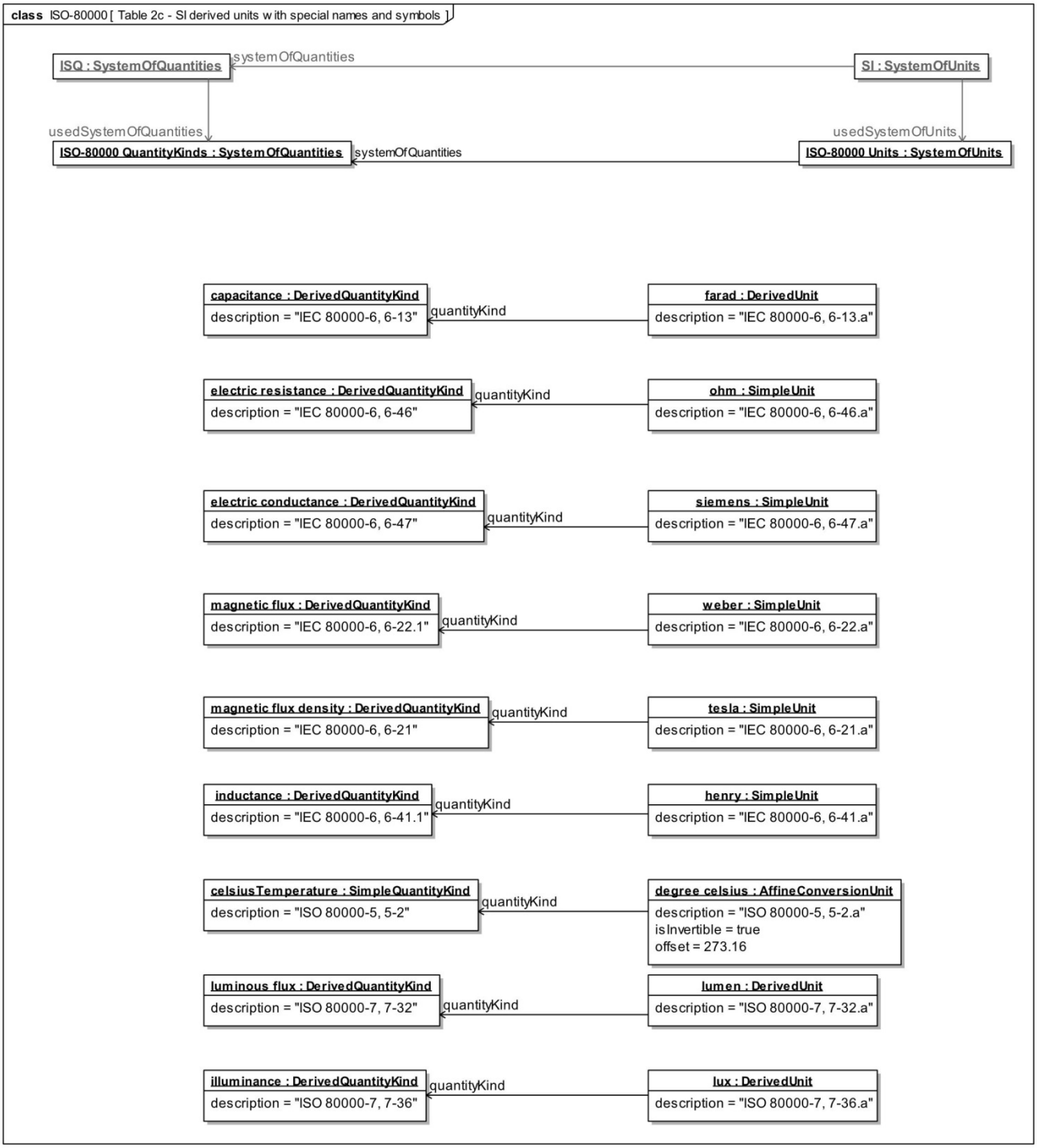


Figure E.17 Table 2 (from ISO 80000-1) ISQ derived quantities and SI derived units with special names (3)

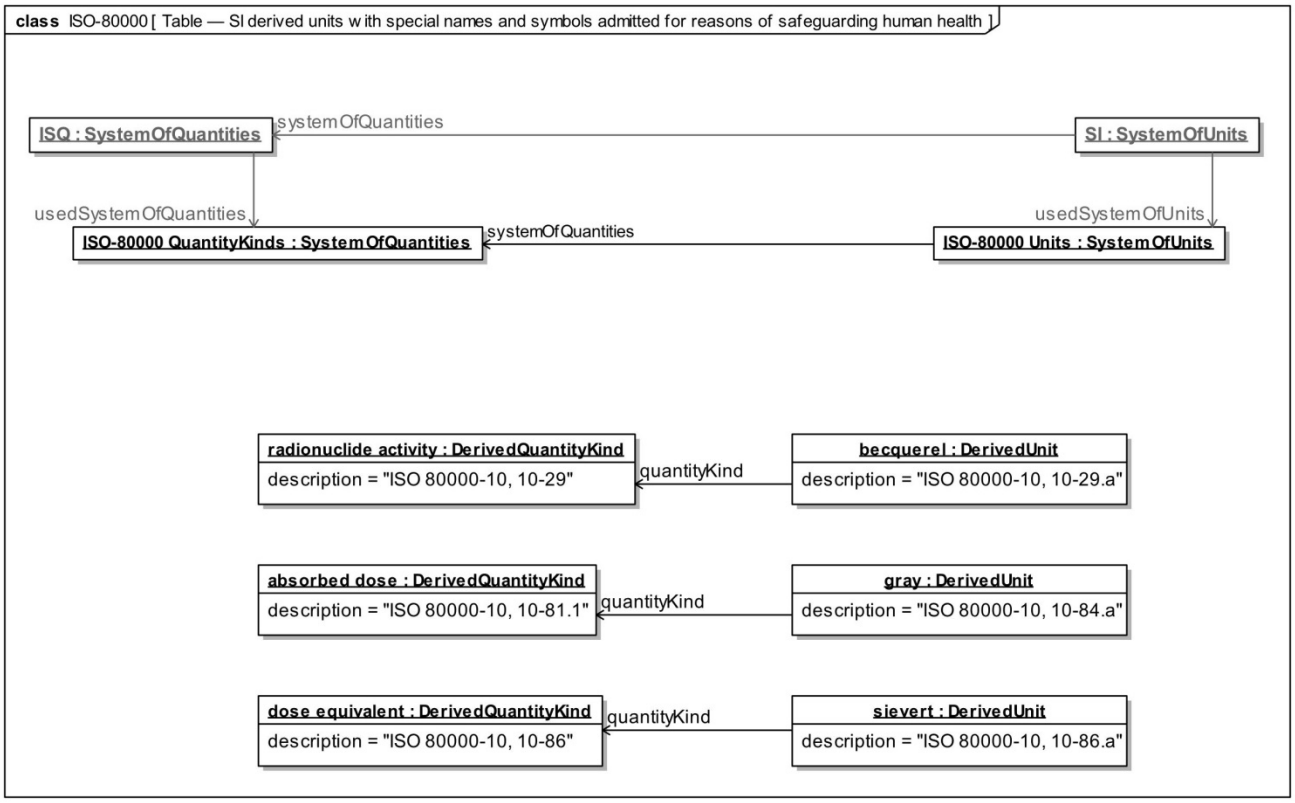


Figure E.18 Table 3 (from the SI brochure) SI derived units with special names and symbols

E.6.3 ISO 80000-1 Prefixes

Table E-6: The decimal and binary prefixes in scope of the International System of Units (SI) which uses the ISO 80000 system of units and its included systems of units such as ISO 80000-13

Prefix name	Prefix Factor (num, den)	Defining Part
yocto	1,10^24	ISO 80000-1 General
zepto	1,10^21	ISO 80000-1 General
atto	1,10^18	ISO 80000-1 General
femto	1,10^15	ISO 80000-1 General
pico	1,10^12	ISO 80000-1 General

Prefix name	Prefix Factor (num, den)	Defining Part
nano	1,10^9	ISO 80000-1 General
micro	1,10^6	ISO 80000-1 General
milli	1,10^3	ISO 80000-1 General
centi	1,10^2	ISO 80000-1 General
deci	1,10^1	ISO 80000-1 General
deca	10^1.1	ISO 80000-1 General
hecto	10^2.1	ISO 80000-1 General
kilo	10^3.1	ISO 80000-1 General
mega	10^6.1	ISO 80000-1 General
giga	10^9.1	ISO 80000-1 General
tera	10^12.1	ISO 80000-1 General
peta	10^15.1	ISO 80000-1 General
exa	10^18.1	ISO 80000-1 General
zetta	10^21.1	ISO 80000-1 General
yotta	10^24.1	ISO 80000-1 General
kibi	(2^10)^1,1	IEC80000-13 Information Science and Technology
mebi	(2^10)^2,1	IEC80000-13 Information Science and Technology
gibi	(2^10)^3,1	IEC80000-13 Information Science and Technology
lebi	(2^10)^4,1	IEC80000-13 Information Science and Technology
pebi	(2^10)^5,1	IEC80000-13 Information Science and Technology
exbi	(2^10)^6,1	IEC80000-13 Information Science and Technology

zebi	(2^10)^7,1	IEC80000-13 Information Science and Technology
yobi	(2^10)^8,1	IEC80000-13 Information Science and Technology

E.6.4 ISO 80000-2 Mathematical Signs and Symbols

ISO 80000 part 2 defines Mathematical Signs and Symbols used in other ISO 80000 parts. In the SysML library, this part contains definitions of constant numbers used across all other parts.

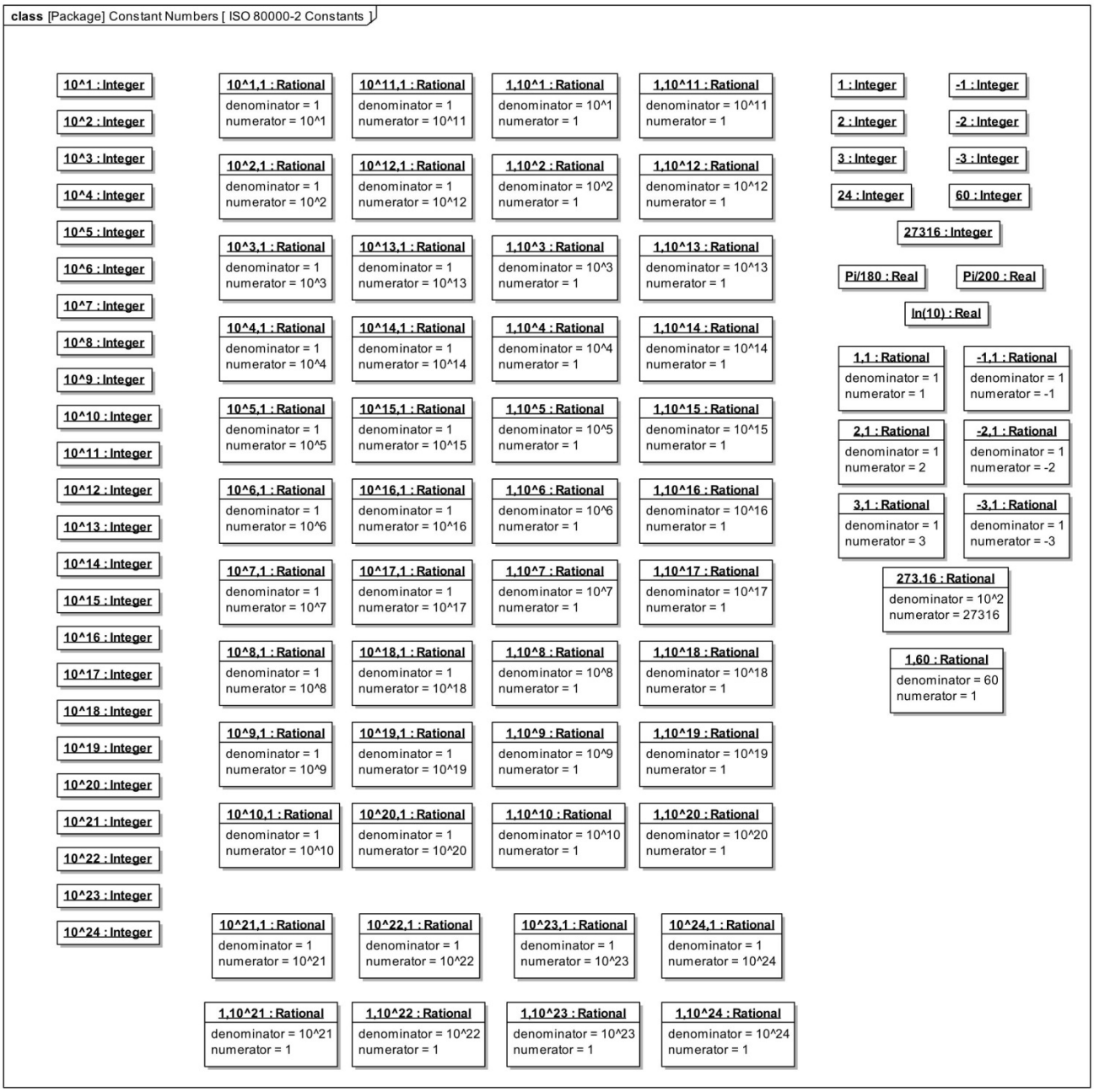


Figure E.19: Constant numbers used throughout the SysML ISO 80000 library

E.6.5 Summary of the covered parts of ISO 80000

The following sub clauses provide a summary overview of all definitions of units and quantity kinds grouped by ISO 80000 part (3,4,5,6,7,9,10,13). Note that “quantities” in the ISO documents correspond to “QuantityKinds” in QUDV. As explained in 8.3.3.2.1, QuantityKind, the type of a SysML value property (i.e., a VIM “quantity”), a SysML ValueType,

specifies the QUDV QuantityKind aspects that this “quantity” has in common with other “quantities” typed by SysML ValueTypes referencing the same QUDV QuantityKind aspect.

The SysML definitions are indexed and ordered according to their corresponding ISO 80000 definition. The ISO 80000 part document provides the authoritative reference for the meaning of the corresponding SysML definitions of units and quantity kinds.

Prefixes apply for all units except for units corresponding to quantities of dimension one or for units in non-reduced form. The 20 decimal prefixes apply to such units in parts 3,4,5,6,7,9,10; the 8 binary prefixes apply to such units in parts 13. For a derived unit defined in terms of N other units, there are 20^N possible prefixed derived units; far too many to create explicitly. This library contains only the combinations for the first factor for each derived unit. Finally, the library includes value type definitions for the possible combinations of quantity kinds and compatible units and prefixed units represented in the library.

All value type definitions follow the same pattern: a toplevel value type is defined with only the quantity kind. This value type is compatible with values typed by specializations of that toplevel value type that specify a particular unit. The following diagram shows the resulting taxonomy for the value types about mass (ISO 80000 -4, 4-1) and all applicable prefixes for the corresponding unit, gram (ISO 80000-4, 4-1.a).

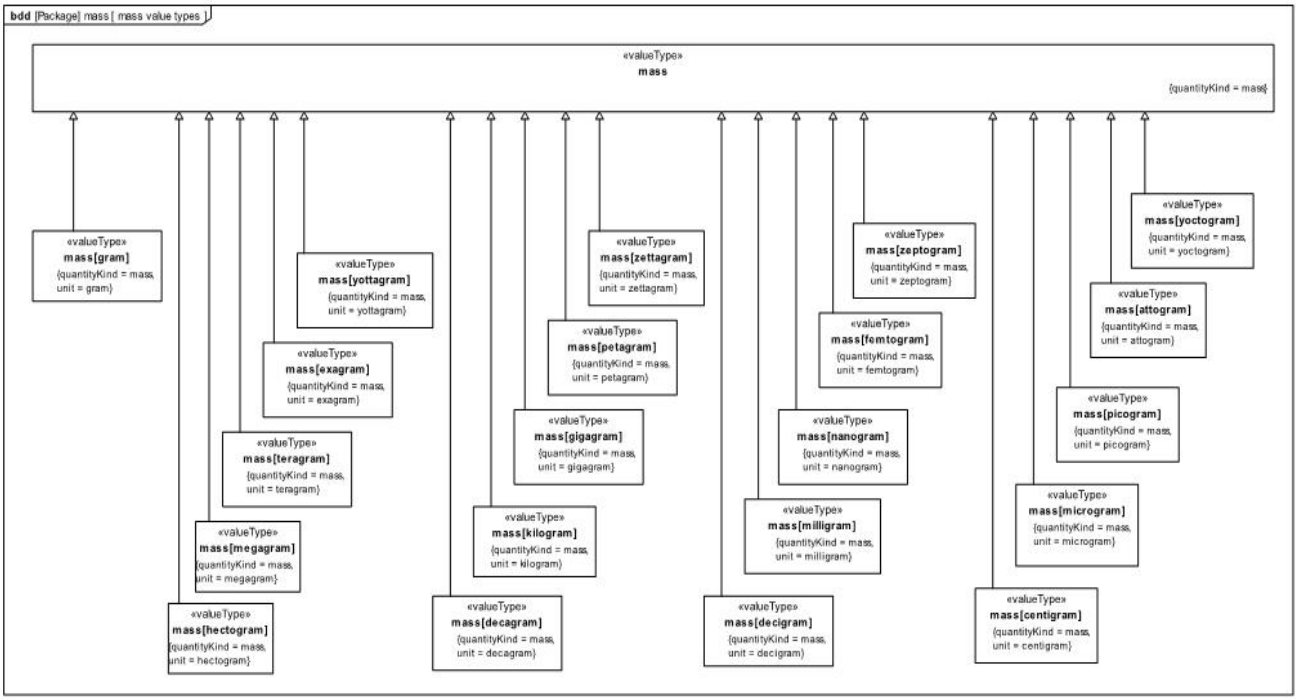


Figure E.20 Example of value type definitions for a quantity and applicable units and prefixed un

E.6.5.1 ISO 80000-3 Space and Time

All 25 entries (including sub-entries) in the normative contents of ISO 80000-3 are modeled as summarized below.

Table E-7: Normative units in ISO 80000-3 (1 of 2)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of	is reduced form?
metre	ISO 80000-3, 3-1.a, 3-17.a	m		ISO 80000-3, 3-1.1		
metre to the power minus one	ISO 80000-3, 3-2.a, 3-18.a, 3-19.a, 3-25.a	m-1		ISO 80000-3, 3-2 [5]		
square metre	ISO 80000-3, 3-3.a	m^2		ISO 80000-3, 3-3		
cubic metre	ISO 80000-3, 3-4.a	m^3		ISO 80000-3, 3-4		
litre	ISO 80000-3, 3-4.b	1				
radian	ISO 80000-3, 3-5.a	rad		ISO 80000-3	TRUE	
degree angle	degree angle	°			TRUE	
minute angle	ISO 80000-3, 3-5.c	'			TRUE	
second angle	ISO 80000-3, 3-5.d	"			TRUE	
gon	ISO 80000-3, 3-5.e	gon			TRUE	
steradian	ISO 80000-3, 3-6.a	srad		ISO 80000-3, 3-6	TRUE	
second	ISO 80000-3, 3-7.a, 3-12.a, 3-13.a	s		ISO 80000-3, 3-7		
minute	ISO 80000-3, 3-7.b	min				
hour	ISO 80000-3, 3-7.c	h				
day	ISO 80000-3, 3-7.d	d				
metre per second	ISO 80000-3, 3-8.a, 3-20.a	m/s		ISO 80000-3, 3-8.1		
metre per second squared	ISO 80000-3, 3-9.a	m/s^2		ISO 80000-3, 3-9.1		
radian per second	ISO 80000-3, 3-10.a,	rad/s	ISO 80000-3,	ISO 80000-3,		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of	is reduced form?
	3-16.a		3-15.b, 3-16.b, 3-23.\a [4\]	3-10 [5]		
radian per second squared	ISO 80000-3, 3-11.a	rad/s^2		ISO 80000-3,3-11		
number of turns	ISO 80000-3, 3-14.a			ISO 80000-3,3-14	TRUE	
revolution	ISO 80000-3, 3-14.a		ISO 80000-3, 3-14.a		TRUE	
hertz	ISO 80000-3, 3-15.a	H_z		ISO 80000-3, 3-15.1		

Table E-8: Normative units in ISO 80000-3 (2 of 2)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension	is reduced form?
number of turns per second	ISO 80000-3, 3-15.b	s-1	ISO 80000-3, 3-15.b, 3-16.b	ISO 80000-3, 3-15.2 [6]		
second to the power minus one	ISO 80000-3, 3-15.b, 3-16.b, 3-23.a	s-1		ISO 80000-3, 3-23 [6]		
revolution per second	ISO 80000-3, 3-15.b	r/s	ISO 80000-3, 3-15.b			
revolution per minute	ISO 80000-3, 3-15.b	r/min	ISO 80000-3			
neper	ISO 80000-3, 3-21.a, 3-22.a, 3-24.b	Np		ISO 80000-3, 3-24	TRUE	
bel	ISO 80000-3, 3-21.b, 3-22.b	B		ISO 80000-3, 3-24 [6]	TRUE	
bel per second	ISO 80000-3, 3-23.b, 3-24.b	B/s	ISO 80000-3, 3-15.b, 3-16.b, 3-23.a [4]			

neper per second	ISO 80000-3, 3-23.b	Np/s	ISO 80000-3, 3-15.b, 3-16.b, 3-23.a			

Table E-9: Normative quantity kinds in ISO 80000-3 (1 of 2)

Quantity Kind name	Description	Symbol	General	is dimension 1?
length	ISO 80000-3, 3-1.1	l, L		
breadth	ISO 80000-3, 3-1.2	b, B	ISO 80000-3, 3-1.1 [5]	
height	ISO 80000-3, 3-1.3	h, H	ISO 80000-3, 3-1.1 [5]	
thickness	ISO 80000-3, 3-1.4	d, δ	ISO 80000-3, 3-1.1 [5]	
radius	ISO 80000-3, 3-1.5	r, R	ISO 80000-3, 3-1.1 [5]	
radial distance	ISO 80000-3, 3-1.6	$\mathbf{r}_Q, \rho$	ISO 80000-3, 3-1.1 [5]	
diameter	ISO 80000-3, 3-1.7	d, D	ISO 80000-3, 3-1.1 [5]	
length of path	ISO 80000-3, 3-1.8	s	ISO 80000-3, 3-1.1 [5]	
distance	ISO 80000-3, 3-1.9	d, r	ISO 80000-3, 3-1.1 [5]	
cartesian coordinates	ISO 80000-3, 3-1.10	x, y, z	ISO 80000-3, 3-1.1 [5]	
position vector	ISO 80000-3, 3-1.11	$\mathbf{r}$	ISO 80000-3, 3-1.1 [5]	
displacement	ISO 80000-3, 3-1.12	Δr	ISO 80000-3, 3-1.1 [5]	
radius of curvature	ISO 80000-3, 3-1.13	ρ	ISO 80000-3, 3-1.1 [5]	
curvature	ISO 80000-3, 3-2	χ		
area	ISO 80000-3, 3-3	$A, (S)$		
volums	ISO 80000-3, 3-4	V		
plane angle	ISO 80000-3, 3-5	$\alpha, \beta, \gamma, \vartheta, \phi$		TRUE
solid angle	ISO 80000-3, 3-6	Ω		TRUE

Quantity Kind name	Description	Symbol	General	is dimension 1?
time	ISO 80000-3, 3-7	t		
speed	ISO 80000-3, 3-8.1	u,v,w	ISO 80000-3, 3-8.1 [5]	
velocity	ISO 80000-3, 3-8.1	$\boldsymbol{v}$		
speed of propagation of waves	ISO 80000-3, 3-8.2	c	ISO 80000-3, 3-8.1 [5]	
acceleration	ISO 80000-3, 3-9.1	a		
acceleration of free fall	ISO 80000-3, 3-9.2	g	ISO 80000-3, 3-9.1 [5]	
angular velocity	ISO 80000-3, 3-10	ω,ω		
angular acceleration	ISO 80000-3, 3-11	α		
period duration	ISO 80000-3, 3-12	T	ISO 80000-3, 3-7 [5]	
time constant for an exponentially varying quantity	ISO 80000-3, 3-13	$\tau,(T)$	ISO 80000-3, 3-7 [5]	
rotation	ISO 80000-3, 3-14	N		TRUE
frequency	ISO 80000-3, 3-15.1	f,ν		

Table E-10: Normative quantity kinds in ISO 80000-3 (2 of 2)

Quantity Kind name	Description	Symbol	General	is dimension 1?
rational frequency	ISO 80000-3, 3-15.2	n		
angular frequency	ISO 80000-3, 3-16	ω		
wavelength	ISO 80000-3, 3-17	λ	ISO 80000-3, 3-1.1 [5]	
linear repetency8	ISO 80000-3, 3-18			
angular repetency	ISO 80000-3, 3-19	k		

Quantity Kind name	Description	Symbol	General	is dimension 1?
phase velocity	ISO 80000-3, 3-20.1	c, v, c_ϕ, v_ϕ	ISO 80000-3, 3-8.1 [5]	
group velocity	ISO 80000-3, 3-20.2	c_g, v_g	ISO 80000-3, 3-8.1 [5]	
level of a field quantity	ISO 80000-3, 3-21	L_F		TRUE
level of a power quantity	ISO 80000-3, 3-22	L_P		TRUE
damping coefficient for an exponentially varying quantity	ISO 80000-3, 3-23	δ		
logarithmic decrement for an exponentially varying quantity	ISO 80000-3, 3-24	Λ	ISO 80000-3, 3-23 [6]	TRUE
attenuation coefficient for an exponentially varying quantity	ISO 80000-3, 3-25.1	α	ISO 80000-3, 3-25.3 [6]	
phase coefficient for an exponentially varying quantity	ISO 80000-3, 3-25.2	β	ISO 80000-3, 3-25.3 [6]	
propagation coefficient for an exponentially varying quantity	ISO 80000-3, 3-25.3	γ		

E.6.5.2 ISO 80000-4 Mechanics

All 37 entries (including sub-entries) in the normative contents of ISO 80000-4 are modeled as summarized below.

Table E-11: Normative units in ISO 80000-4 (1 of 2)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
gram	ISO 80000-4, 4-1.a	g		ISO 80000-4, 4-1 [10]		
tonne	ISO 80000-4, 4-1.b	t				
kilogram per cubic metre	ISO 80000-4, 4-2.a	kg/m^3		ISO 80000-4, 4-2 [10]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
mass density ratio	ISO 80000-4, 4-3.a			ISO 80000-4, 4-3 [10]	TRUE	
cubic metre per kilogram	ISO 80000-4, 4-4.a	m^3/kg		ISO 80000-4, 4-4 [10]		
kilogram per square metre	ISO 80000-4, 4-5.a	kg/m^2		ISO 80000-4, 4-5 [10]		
kilogram per metre	ISO 80000-4, 4-6.a	kg/m		ISO 80000-4, 4-6 [10]		
kilogram metre squared	ISO 80000-4, 4-7.a	$kg.m^2$		ISO 80000-4, 4-7 [10]		
kilogram metre per second	ISO 80000-4, 4-8.a	kg.m/s		ISO 80000-4, 4-8 [10]		
newton	ISO 80000-4, 4-9.a	N		ISO 80000-4, 4-9.1 [10]		
newton metre squared per kilogram squared	ISO 80000-4, 4-10.a	$N \cdot m^2/kg^2$		ISO 80000-4, 4-10 [10]		
newton second	ISO 80000-4, 4-11.a	N.s		ISO 80000-4, 4-11 [10]		
kilogram metre squared per second	ISO 80000-4, 4-12.a	$k_g \cdot m^2/s$		ISO 80000-4, 4-12 [10]		
newton metre	ISO 80000-4, 4-13.a	N.m		ISO 80000-4, 4-13 [10]		
newton metre second	ISO 80000-4, 4-14.a	N.m.s		ISO 80000-4, 4-14 [10]		
pascal	ISO 80000-4, 4-15.a 4-18.a	Pa		ISO 80000-4, 4-15.1 [10]		
cubic metre strain factor	ISO 80000-4, 4-16.a		ISO 80000-4, 4-16.a [7]	ISO 80000-4, 4-16.3 [11]	TRUE	
strain factor	ISO 80000-4, 4-16.a			ISO 80000-4, 4-16.1.2.3 [10]	TRUE	

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
metre strain factor	ISO 80000-4, 4-16.a		ISO 80000-4, 4-16.a [7]	ISO 80000-4, 4-16.1 [11]	TRUE	

Table E-12: Normative units in ISO 80000-4 (2 of 2)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
contraction to elongation metre ratio	ISO 80000-4, 4-17.a			ISO 80000-4, 4-17 [11]	TRUE	
cubic metre strain factor per pascal	ISO 80000-4, 4-19.a	Pa^{-1}	ISO 80000-4, 4-19.a [8]	ISO 80000-4, 4-19 [11]		
pascal to the power minus one	ISO 80000-4, 4-19.a	Pa^{-1}				
metre to the power of four	ISO 80000-4, 4-20.a	m^4		ISO 80000-4, 4-20.1 [11]		
newton ratio	ISO 80000-4, 4-22.a			ISO 80000-4, 4-22.1 [13]	TRUE	
pascal second	ISO 80000-4, 4-23.a	Pa.s		ISO 80000-4, 4-23 [13]		
metre per second per metre	ISO 80000-4, 4-23.a		ISO 80000-3, 3-15b, 3-16b, 3-23.a [4]	ISO 80000-4, 4-23 [13]		FALSE
square metre per second	ISO 80000-4, 4-24.a	m^2/s				
pascal second kilogram per cubic metre	ISO 80000-4, 4-24.a	m^2/s	ISO 80000-4, 4-24.a [8]	ISO 80000-4, 4-24 [13]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
newton per metre	ISO 80000-4, 4-25.a	N/m		ISO 80000-4, 4-25 [13]		
watt	ISO 80000-4, 4-26.a, 4-56.a	W	ISO 80000-4, 4-26.a [8]			
joule per second	ISO 80000-4, 4-26.a	J/s	ISO 80000-4, 4-26.a [8]			
newton metre per second	ISO 80000-4, 4-26.a	N.m/s		ISO 80000-4, 4-26 [13]		
joule	ISO 80000-4, 4-27.a, 4.34.a, 4-36.a	J		ISO 80000-4, 4-34 [14]		
output watt	ISO 80000-4, 4-28.a	W_{out}	ISO 80000-4, 4-26.a, 4-56.a [8]	ISO 80000-4, 4-28 [13]		
output input watt ratio	ISO 80000-4, 4-28.a			ISO 80000-4, 4-28 [13]		
input watt	ISO 80000-4, 4-28.a	W_{out}	ISO 80000-4, 4-26.a, 4-56.a [8]	ISO 80000-4, 4-28 [13]		
kilogram per second	ISO 80000-4, 4-29.a	kg/s		ISO 80000-4, 4-29 [14]		
cubic metre per second	ISO 80000-4, 4-30.a	m^3/s		ISO 80000-4, 4-30 [14]		
joule second	ISO 80000-4, 4-37.a	J.s		ISO 80000-4, 4-37 [14]		

Table E-13: Normative quantity kinds in ISO 80000-4 (1 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
mass	ISO 80000-4, 4-1	m		
density	ISO 80000-4, 4-2		ISO 80000-4, 4-24 [10]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
mass density of a reference substance	ISO 80000-4, 4-2, 4-3	ρ_0	ISO 80000-4, 4-24 [10]	
mass density	ISO 80000-4, 4-2	ρ		
relative mass density	ISO 80000-4, 4-3	d		TRUE
specificVolume	ISO 80000-4, 4-4	v		
surface density	ISO 80000-4, 4-5	ρ_A		
linear density	ISO 80000-4, 4-6	ρ_l		
mass moment of inertia	ISO 80000-4, 4-7	I, J		
momentum	ISO 80000-4, 4-8	$\boldsymbol{p}$		
force	ISO 80000-4, 4-9.1	$\boldsymbol{F}$		
weight	ISO 80000-4, 4-9.2	$\boldsymbol{F}_g, \boldsymbol{G}$		
gravitational constant between two mass particles	ISO 80000-4, 4-10	G		
impulse	ISO 80000-4, 4-11	$\boldsymbol{I}$		
moment of momentum	ISO 80000-4, 4-12	$\boldsymbol{L}$		
moment of force	ISO 80000-4, 4-13.1	M		
torque	ISO 80000-4, 4-13.2	T	ISO 80000-4, 4-13.1 [10]	
bending moment of force	ISO 80000-4, 4-13.3	M_b	ISO 80000-4, 4-13.1 [10]	
angular impulse	ISO 80000-4, 4-14	H		
pressure	ISO 80000-4, 4-15.1	p		
normal stress	ISO 80000-4, 4-15.2	σ	ISO 80000-4, 4-15.1 [10]	
sheer stress	ISO 80000-4, 4-15.3	τ	ISO 80000-4, 4-15.1 [10]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
length of item in a reference state	ISO 80000-4, 4-16	l_0	ISO 80000-3, 3-1.1 [5]	
increase in length	ISO 80000-4, 4-16	Δl	ISO 80000-3, 3-1.1 [5]	
strain	ISO 80000-4, 4-16.1.2.3			TRUE

Table E-14: Normative quantity kinds in ISO 80000-4 (2 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
linear strain	ISO 80000-4, 4-16.1	$\varepsilon, (e)$	ISO 80000-4, 4-16.1.2.3 [10]	TRUE
thickness of a layer between two surfaces	ISO 80000-4, 4-16.2	d	ISO 80000-3, 3-1.4 [5]	
sheer strain	ISO 80000-4, 4-16.2	γ	ISO 80000-4, 4-16.1.2.3 [10]	TRUE
parallel displacement between two surfaces of a layer	ISO 80000-4, 4-16.2	Δx	ISO 80000-3, 3-1.12 [5]	
increase in volume	ISO 80000-4, 4-16.3	ΔV	ISO 80000-3, 3-4 [5]	
volume strain	ISO 80000-4, 4-16.3	ϑ	ISO 80000-4, 4-16.1.2.3 [10]	TRUE
volume in a reference state	ISO 80000-4, 4-16.3	V_0	ISO 80000-3, 3-4 [5]	
elongation	ISO 80000-4, 4-17	Δl	ISO 80000-3, 3-1.1 [5]	
lateral contraction	ISO 80000-4, 4-17	$\Delta\delta$	ISO 80000-3, 3-1.1 [5]	
poisson number	ISO 80000-4, 4-17	$\mu, (\nu)$		TRUE
modulus of elasticity	ISO 80000-4, 4-18.1	E	ISO 80000-4, 4-18.1.2.3 [11]	
modulus	ISO 80000-4, 4-18.1.2.3		ISO 80000-4, 4-15.1 [10]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
modulus of rigidity	ISO 80000-4, 4-18.2	G	ISO 80000-4, 4-18.1.2.3 [11]	
modulus of compression	ISO 80000-4, 4-18.3	K	ISO 80000-4, 4-18.1.2.3 [11]	
compressibility	ISO 80000-4, 4-19	χ		
increase in pressure	ISO 80000-4, 4-19		ISO 80000-4, 4-15.1 [10]	
surface considered	ISO 80000-4, 4-20		ISO 80000-3, 3-3 [5]	
second axial moment of area	ISO 80000-4, 4-20.1	I_a		
radial distance from a Q-axis in the plane of the surface considered	ISO 80000-4, 4-20.1	r_Q	ISO 80000-3, 3-1.6 [5]	
second polar moment of area	ISO 80000-4, 4-20.2	I_p		
radial distance from a Q-axis perpendicular to the plane of the surface considered	ISO 80000-4, 4-20.2	r_Q	ISO 80000-3, 3-1.6 [5]	

Table E-15: Normative quantity kinds in ISO 80000-4 (3 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
section modules	ISO 80000-4, 4-21	$Z, (W)$		
maximum radial distance from a Q-axis in the plane of the surface considered	ISO 80000-4, 4-21	$I_{Q,\max}$	ISO 80000-4, 4-20.1 [11]	
maximum tangential component of the contact force between two bodies at rest	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
tangential component of the contact force between two sliding bodies	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	
contact force between two sliding bodies	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	
tangential component of the contact force between two bodies at rest	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	
tangential component of the contact force between two bodies	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	
contact force between two bodies	ISO 80000-4, 4-22	F	ISO 80000-4, 4-9.1 [10]	
normal component of the contact force between two sliding bodies	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [13]	
maximum contact force between two bodies	ISO 80000-4, 4-22	$F_{\max}$	ISO 80000-4, 4-22 [12]	
contact force between two bodies at rest	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	
normal component of the contact force between two bodies at rest	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [13]	
normal component of the contact force between two bodies	ISO 80000-4, 4-22		ISO 80000-4, 4-22 [12]	

Table E-16: Normative quantity kinds in ISO 80000-4 (4 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
dynamic friction factor	ISO 80000-4, 4-22.1	$\mu_c(f)$		TRUE

Quantity Kind name	Description	Symbol	General	is dimension 1?
static friction factor	ISO 80000-4, 4-22.2	$\mu_s, (f_s)$		TRUE
velocity gradient	ISO 80000-4, 4-23			
dynamic viscosity	ISO 80000-4, 4-23			
kinematic viscosity	ISO 80000-4, 4-24	ν		
surface tension	ISO 80000-4, 4-25	γ, σ		
force component perpendicular to a line element in a surface	ISO 80000-4, 4-25		ISO 80000-4, 4-9.1 [10]	
length of line element in a surface	ISO 80000-4, 4-25		ISO 80000-3, 3-1.1 [5]	
power	ISO 80000-4, 4-26	P		
work	ISO 80000-4, 4-27.1	W		
potential energy	ISO 80000-4, 4-27.2	${}^{E}p$	ISO 80000-4, 4-27.4 [13]	
kinetic energy	ISO 80000-4, 4-27.3	${}^{E}k$	ISO 80000-4, 4-27.4 [13]	
mechanical energy	ISO 80000-4, 4-27.4	E	ISO 80000-4, 4-27.1 [13]	
power efficiency	ISO 80000-4, 4-28	η		TRUE
output power	ISO 80000-4, 4-28	P_{out}	ISO 80000-4, 4-26 [13]	
input power	ISO 80000-4, 4-28	P_{in}	ISO 80000-4, 4-26 [13]	
mass flow rate	ISO 80000-4, 4-29	q_{m}		
volume flow rate	ISO 80000-4, 4-30	q_{V}		
generalized coordinate	ISO 80000-4, 4-31	q_i		
generalized velocity	ISO 80000-4, 4-32	q_i		
generalized force	ISO 80000-4, 4-33	Q_i		

Quantity Kind name	Description	Symbol	General	is dimension 1?
generalized potential energy	ISO 80000-4, 4-34	$V\left(q_i, \dot{q}_i\right)$		
generalized kinetic energy	ISO 80000-4, 4-34	$T\left(q_i, \dot{q}_i\right)$		
Lagrange function	ISO 80000-4, 4-34	$L\left(q_i, \dot{q}_i\right)$	ISO 80000-4, 4-34 [14]	
generalized momentum	ISO 80000-4, 4-35	p_i		
generalized momentum of velocity	ISO 80000-4, 4-36	$p_i \dot{q}_i$	ISO 80000-4, 4-36 [14]	
Hamilton function	ISO 80000-4, 4-36	H	ISO 80000-4, 4-36 [14]	
action functional	ISO 80000-4, 4-37	S		

Contact force between two bodies is an example of a taxonomy of specialized quantity kinds induced by different measurement procedures.

Per ISO 80000-4, 4-31, 4-32, 4-33 and 4-35, there are no measurement units defined for these generalized quantity kinds; the unit of a particular quantity (i.e., SysML value property) typed by a SysML ValueType referencing a generalized quantity kind depends on the dimension of that particular quantity.

E.6.5.3 ISO 80000-5 Thermodynamics

All 33 entries (including sub-entries) in the normative contents of ISO 80000-5 are modeled as summarized below.

Table E-17: Normative units in ISO 80000-5 (1 of 2)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
kelvin	ISO 80000-5, 5-1.a 5-33.a	K		ISO 80000-5, 5-1 [17]		
degree celsius	ISO 80000-5, 5-2.a	°C		ISO 80000-5, 5-2 [17]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
cubic metre coefficient per kelvin	ISO 80000-5, 5-3.2	K^{-1}	ISO 80000-5, 5-3.a [15]	ISO 80000-5, 5-3.2 [17]		
pascal ratio per kelvin	ISO 80000-5, 5-3.3	K^{-1}	ISO 80000-5, 5-3.a [15]	ISO 80000-5, 5-3.1 [17]		
kelvin to the power minus one	ISO 80000-5, 5-3.a	K^{-1}				
metre coefficient per kelvin	ISO 80000-5, 5-3.a	K^{-1}	ISO 80000-5, 5-3.a [15]	ISO 80000-5, 5-3.1 [17]		
pascal ratio	ISO 80000-5, 5-3.a			ISO 80000-5, 5-3.3 [17]	TRUE	
pascal per kelvin	ISO 80000-5, 5-4.a	Pa / K		ISO 80000-5, 5-4 [17]		
cubic metre ratio per pascal	ISO 80000-5, 5-5.a	Pa^{-1}	ISO 80000-4, 4-19.a [8]	ISO 80000-5, 5-5.1 [17]		
watt per square metre	ISO 80000-5, 5-8.a	W / m^2		ISO 80000-5, 5-8 [18]		
watt per metre kelvin	ISO 80000-5, 5-9.a	$W / (m \cdot K)$		ISO 80000-5, 5-9 [18]		
kelvin per metre	ISO 80000-5, 5-9.a	K / m		ISO 80000-5, 5-9 [18]		
watt per square metre per kelvin	ISO 80000-5, 5-10.a	$W / m^2 \cdot K$		ISO 80000-5, 5-10.1 [18]		
square metre kelvin per watt	ISO 80000-5, 5-11.a	$m^2 \cdot K / W$		ISO 80000-5, 5-11 [18]		
kelvin per watt	ISO 80000-5, 5-12.a	K / W		ISO 80000-5, 5-12 [18]		
watt per kelvin	ISO 80000-5, 5-13.a	W / K		ISO 80000-5, 5-13 [18]		
watt square metre per joule	ISO 80000-5, 5-14.a	$W \cdot m^2 / J$	ISO 80000-4, 4-24.a [8]	ISO 80000-5, 5-14 [18]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
joule per kelvin	ISO 80000-5, 5-15.a, 5-18.a, 5-21.a, 5-22.a, 5-23.a	$\mathrm{J / K}$		ISO 80000-5, 5-18 [19]		

Table E-18: Normative units in ISO 80000-5 (2 of 2)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
joule per kilogram kelvin	ISO 80000-5, 5-16.a	$\mathrm{J/(kg \cdot K)}$		ISO 80000-5, 5-16.1 [18]		
cubic metre per pascal ratio	ISO 80000-5, 5-17.a			ISO 80000-5, 5-17.2 [19]	TRUE	
cubic metre per pascal	ISO 80000-5, 5-17.a	$\mathrm{m^3 / Pa}$		ISO 80000-5, 5-17.2 [19]		
joule per kilogram kelvin ratio	ISO 80000-5, 5-17.a			ISO 80000-5, 5-17.1 [19]	TRUE	
pascal per cubic metre	ISO 80000-5, 5-17.a	$\mathrm{Pa / m^3}$		ISO 80000-5, 5-17.2 [19]		
kelvin joule per kelvin	ISO 80000-5, 5-20.a	J	ISO 80000-4, 4-27.a, 4-34.a, 4-36.a [8]	ISO 80000-5, 5-20. [45] [19]		FALSE
pascal cubic metre	ISO 80000-5, 5-20.a	$\mathrm{Pa \cdot m^3}$	ISO 80000-4, 4-27.a, 4-34.a, 4-36.a [8]	ISO 80000-5, 5-20.3 [19]		
kelvin joule per kelvin kilogram	ISO 80000-5, 5-21.a	$\mathrm{J / K}$	ISO 80000-5, 5-21.a [16]	ISO 80000-5, 5-21.5 [20]		FALSE
joule per kilogram	ISO 80000-5, 5-21.a	$\mathrm{J / K}$		ISO 80000-5, 5-21.1 [19]		
kilogram ratio	ISO 80000-5, 5-26.a, 5-27.a, 5-28.a, 5-29.a			ISO 80000-5, 5-26 [20]	TRUE	

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
kilogram ratio fraction	ISO 80000-5, 5-28.a, 5-32.a			ISO 80000-5, 5-28 [21]	TRUE	
kilogram per cubic metre ratio	ISO 80000-5, 5-31.a			ISO 80000-5, 5-31 [21]	TRUE	

Table E-19: Normative quantity kinds in ISO 80000-5 (1 of 5)

Quantity Kind name	Description	Symbol	General	is dimension 1?
thermodynamic temperature	ISO 80000-5, 5-1	$T, (\Theta)$		
celcius Temperature	ISO 80000-5, 5-2	t, ϑ	ISO 80000-5, 5-1 [17]	
linear expansion coefficient	ISO 80000-5, 5-3.1	α_l		
increase in temperature	ISO 80000-5, 5-3.1.2.3.4	$\partial T, \mathrm{d}T$	ISO 80000-5, 5-1 [17]	
cubic expansion coefficient	ISO 80000-5, 5-3.2	α_V, α, γ		
pressure in a reference state	ISO 80000-5, 5-3.3		ISO 80000-4, 4-15.1 [10]	
relative pressure coefficient	ISO 80000-5, 5-3.3	α_p		
pressure ratio	ISO 80000-5, 5-3.3			TRUE
increase in pressure at constant volums	ISO 80000-5, 5-3.3	$(\partial P)_V$	ISO 80000-4, 4-19 [11]	
increase in temperature at constant volume	ISO 80000-5, 5-3.3	$(\partial T)_V$	ISO 80000-5, 5-3.1.2.3.4 [17]	
pressure coefficient	ISO 80000-5, 5-4	β		
isothermal compressibility	ISO 80000-5, 5-5.1	χ_T		
increase in pressure at constant temperature	ISO 80000-5, 5-5.1	$(\partial P)_T$	ISO 80000-4, 4-19 [11]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
increase in volume at constant temperature	ISO 80000-5, 5-5.1	$(\partial V)_T$	ISO 80000-4, 4-16.3 [11]	
increase in pressure at constant entropy	ISO 80000-5, 5-5.2	$(\partial P)_S$	ISO 80000-4, 4-19 [11]	
isentropic compressibility	ISO 80000-5, 5-5.2	χ_S		
increase in volume at constant entropy	ISO 80000-5, 5-5.2	$(\partial V)_S$	ISO 80000-4, 4-16.3 [11]	

Table E-20: Normative quantity kinds in ISO 80000-5 (2 of 5)

Quantity Kind name	Description	Symbol	General	is dimension 1?
amount of heat	ISO 80000-5, 5-6	Q	ISO 80000-4, 4-27 [13]	
heat flow rate	ISO 80000-5, 5-7	Φ	ISO 80000-4, 4-26 [13]	
surface density of heat flow rate	ISO 80000-5, 5-8	$q_{,}\phi$	ISO 80000-5, 5-8 [18]	
areic heat flow rate	ISO 80000-5, 5-8	$q_{,}\phi$		
thermodynamic temperature gradient	ISO 80000-5, 5-9			
thermal conductivity	ISO 80000-5, 5-9	$\lambda_{,}(\chi)$		
coefficient of heat transfer	ISO 80000-5, 5-10.1	$K_{,}(k)$		
tnermodynamic temperature difference	ISO 80000-5, 5-10.1		ISO 80000-5, 5-1 [17]	
surface coefficient of heat transfer	ISO 80000-5, 5-10.2	$h_{,}(\alpha)$		
surface thermodynamic temperature difference	ISO 80000-5, 5-10.2	$h_{,}(\alpha)$	ISO 80000-5, 5-2 [18]	
surface thermodynamic temperature	ISO 80000-5, 5-10.2		ISO 80000-5, 5-1 [17]	
reference thermodynamic temperature	ISO 80000-5, 5-10.2		ISO 80000-5, 5-1 [17]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
coefficient of thermal insulance	ISO 80000-5, 5-11	M		
thermal resistance	ISO 80000-5, 5-12	R		
thermal conductance	ISO 80000-5, 5-13	$G, (H)$		
thermal diffusivity	ISO 80000-5, 5-14	α		
heat capacity	ISO 80000-5, 5-15	C		
specific heat capacity	ISO 80000-5, 5-16.1	c		
specific heat capacity at constant pressure	ISO 80000-5, 5-16.2	c_p	ISO 80000-5, 5-16.1 [18]	

Table E-21:Normative quantity kinds in ISO 80000-5 (3 of 5)

Quantity Kind name	Description	Symbol	General	is dimension 1?
specific heat capacity at constant volume	ISO 80000-5, 5-16.3	c_V	ISO 80000-5, 5-16.1 [18]	
specific heat capacity at saturation	ISO 80000-5, 5-16.4	c_{sat}	ISO 80000-5, 5-16.1 [18]	
ratio of the specific heat capacities	ISO 80000-5, 5-17.1	γ		TRUE
pressure per volume increase at constant entropy	ISO 80000-5, 5-17.2	χ		
volume per pressure in a reference state	ISO 80000-5, 5-17.2			
isentropic exponent	ISO 80000-5, 5-17.2			
entropy	ISO 80000-5, 5-18	S		
heat received	ISO 80000-5, 5-18	dQ	ISO 80000-5, 5-6 [18]	
specific entropy	ISO 80000-5, 5-19	s		

Quantity Kind name	Description	Symbol	General	is dimension 1?
energy	ISO 80000-5, 5-20.1	E	ISO 80000-4, 4-27.4 [13]	
internal thermodynamic energy	ISO 80000-5, 5-20.2	U	ISO 80000-5, 5-18 [19]	
volumetric pressure	ISO 80000-5, 5-20.3	pV		
enthalpy	ISO 80000-5, 5-20.3	H	ISO 80000-5, 5-20.2 [19]	
Helmholtz energy	ISO 80000-5, 5-20.4	A, F	ISO 80000-5, 5-20.2 [19]	
Gibbs energy	ISO 80000-5, 5-20.5	G	ISO 80000-5, 5-20.3 [19]	
system enthalpy at thermodynamic temperature	ISO 80000-5, 5-20 [45]	TS		
specific energy	ISO 80000-5, 5-21.1	e		
specific internal thermodynamic energy	ISO 80000-5, 5-21.2	u	ISO 80000-5, 5-21.1 [19]	
specific enthalpy	ISO 80000-5, 5-21.3	h	ISO 80000-5, 5-21.2 [19]	
specific Helmholtz energy	ISO 80000-5, 5-21.4	a, f		

Table E-22: Normative quantity kinds in ISO 80000-5 (4 of 5)

Quantity Kind name	Description	Symbol	General	is dimension 1?
specific Gibs energy	ISO 80000-5, 5-21.5	g		
Massieu function	ISO 80000-5, 5-22	J		
Planck function	ISO 80000-5, 5-23	Y		
mass of water irrespective of the form of aggregation	ISO 80000-5, 5-24	m	ISO 80000-4, 4-1 [10]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
mass concentration of water at saturation	ISO 80000-5, 5-24	w_{sat}	ISO 80000-4, 4-2 [10]	
total volume of water and dry matter	ISO 80000-5, 5-24	V	ISO 80000-3, 3-4 [5]	
mass concentration of water	ISO 80000-5, 5-24	w	ISO 80000-4, 4-2 [10]	
mass of water vapour	ISO 80000-5, 5-24	m	ISO 80000-5, 5-24 [20]	
mass concentration of water vapour	ISO 80000-5, 5-25	v	ISO 80000-4, 4-2 [10]	
mass concentration of water vapour at saturation	ISO 80000-5, 5-25	v_{sat}	ISO 80000-4, 4-2 [10]	
mass of water at saturation	ISO 80000-5, 5-25	m_{sat}	ISO 80000-5, 5-24 [20]	
mass of water vapour at saturation	ISO 80000-5, 5-25	m_{sat}	ISO 80000-5, 5-24 [20]	
mass ratio of water to dry matter	ISO 80000-5, 5-26	u		TRUE
mass of dry matter	ISO 80000-5, 5-26	m_d	ISO 80000-4, 4-1 [10]	TRUE
mass ratio of water to dry gas at saturation	ISO 80000-5, 5-26	u_{sat}	ISO 80000-5, 5-26 [20]	TRUE
mass ratio of water vapour to dry gas	ISO 80000-5, 5-27	w	ISO 80000-5, 5-26 [20]	TRUE
mass ration of water vapour to dry gas at saturation	ISO 80000-5, 5-27	w_{sat}	ISO 80000-5, 5-27 [20]	TRUE

Table E-23: Normative quantity kinds in ISO 80000-5 (5 of 5)

Quantity Kind name	Description	Symbol	General	is dimension 1?
mass of dry gas	ISO 80000-5, 5-27	m_d	ISO 80000-5, 5-26 [20]	
mass fraction of water	ISO 80000-5, 5-28	w_{H_2O}		TRUE
mass fraction of dry matter	ISO 80000-5, 5-29	w_{H_2O}	ISO 80000-5, 5-28 [21]	TRUE
partial pressure of a gas in a mixture at saturation	Quantity Kind name	Description	Symbol	General
partial pressure of a gas in a mixture	ISO 80000-5, 5-30	p	ISO 80000-4, 4-15.1 [10]	
relative partial pressure of a gas	ISO 80000-5, 5-30		ISO 80000-5, 5-3.3 [17]	TRUE
relative mass concentration of water vapour	ISO 80000-5, 5-31	ϕ		TRUE
relative mass ratio of water vapour	ISO 80000-5, 5-32			TRUE
dew point thermodynamic temperature of humid air	ISO 80000-5, 5-33	T_d	ISO 80000-5, 5-33 [21]	
thermodynamic temperature of humid air	ISO 80000-5, 5-33	T	ISO 80000-5, 5-1 [17]	

E.6.5.4 ISO 80000-6 Electromagnetism

All 62 entries (including sub-entries) in the normative contents of ISO 80000-6 are modeled as summarized below.

Table E-24: Normative units in ISO 80000-6 (1 of 5)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
ampere	IEC 80000-6, 6-1.a	A		IEC 80000-6, 6-1 [27]		
coulomb	IEC 80000-6, 6-2.a	C		IEC 80000-6, 6-2 [27]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
coulomb per cubic metre	IEC 80000-6, 6-3.a	C/m^3		IEC 80000-6, 6-3 [27]		
coulomb per square metre	IEC 80000-6, 6-4.a	C/m^2		IEC 80000-6, 6-4 [27]		
coulomb per metre	IEC 80000-6, 6-5.a	C/m		IEC 80000-6, 6-5 [27]		
coulomb metre	IEC 80000-6, 6-6.a	$C \cdot m$		IEC 80000-6, 6-6 [27]		
coulomb per square metre per second	IEC 80000-6, 6-7.a	$C/(m^2 \cdot s)$	IEC 80000-6, 6-8.a [22]	IEC 80000-6, 6-8 [27]		
coulomb per metre squared	IEC 80000-6, 6-7.a	C/m^2	IEC 80000-6, 6-4.a [22]	IEC 80000-6, 6-7 [27]		
ampere per square metre	IEC 80000-6, 6-8.a	A/m^2		IEC 80000-6, 6-8 [27]		
coulomb per metre per second	IEC 80000-6, 6-9.a	$C/(m \cdot s)$	IEC 80000-6, 6-25.a [23]	IEC 80000-6, 6-9 [27]		
volt per metre	IEC 80000-6, 6-10.a	V/m		IEC 80000-6, 6-10 [27]		
newton per coulomb	IEC 80000-6, 6-10.a	N/C	IEC 80000-6, 6-10.a [22]	IEC 80000-6, 6-10 [27]		
volt	IEC 80000-6, 6-11.a	V		IEC 80000-6, 6-11.1 [27]		
volt metre per metre	IEC 80000-6, 6-11.a	$V \cdot m/m$	IEC 80000-6, 6-11.a [22]			FALSE
farad volt per metre squared	IEC 80000-6, 6-12.a	$F \cdot V/m^2$	IEC 80000-6, 6-7.a [22]	IEC 80000-6, 6-12 [27]		
farad	IEC 80000-6, 6-13.a	F		IEC 80000-6, 6-13 [27]		
farad per metre	IEC 80000-6, 6-14.a	F/m	IEC 80000-6, 6-14.a [22]	IEC 80000-6, 6-14.1 [27]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
coulomb per volt per metre	IEC 80000-6, 6-14.a	$C/(V \cdot m)$		IEC 80000-6, 6-14.2 [28]		
coulomb per volt per metre ratio	IEC 80000-6, 6-15.a			IEC 80000-6, 6-15 [28]	TRUE	

Table E-25: Normative units in ISO 80000-6 (2 of 5)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
coulomb per metre squared ratio	IEC 80000-6, 6-16.a			IEC 80000-6, 6-16 [28]	TRUE	
square metre coulomb per metre squared	IEC 80000-6, 6-17.a	C	IEC 80000-6, 6-2.a [22]			FALSE
coulomb per metre squared per second	IEC 80000-6, 6-18.a	$C/(m^2 \cdot s)$	IEC 80000-6, 6-8.a [22]			
square metre ampere per square metre	IEC 80000-6, 6-19.a	A	IEC 80000-6, 6-1.a [22]			FALSE
volt second per metre squared	IEC 80000-6, 6-21	$V \cdot s / A \cdot m^2$	IEC 80000-6, 6-21.a [23]			
newton per ampere per metre	IEC 80000-6, 6-21.a	$N/(A \cdot m)$	IEC 80000-6, 6-21.a [23]			
tesla	IEC 80000-6, 6-21.a	T		IEC 80000-6, 6-21 [28]		
weber	IEC 80000-6, 6-22.a	Wb		IEC 80000-6, 6-21.1 [28]		

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
newton metre per ampere	IEC 80000-6, 6-22.a	$N \cdot m / A$	IEC 80000-6, 6-22.a [23]			
volt second	IEC 80000-6, 6-22.a	$V \cdot s$	IEC 80000-6, 6-22.a [23]			
ampere square metre	IEC 80000-6, 6-23.a	$A \cdot m^2$		IEC 80000-6, 6-23 [28]		
ampere square metre per cubic metre	IEC 80000-6, 6-24.a	$A \cdot m^2 / m^3$	IEC 80000-6, 6-25.a [23]			FALSE
newton per weber	IEC 80000-6, 6-25	N / Wb	IEC 80000-6, 6-25.a [23]			
ampere per metre	IEC 80000-6, 6-25.a	A / m		IEC 80000-6, 6-25 [28]		
ampere metre per metre squared	IEC 80000-6, 6-25.a	$A \cdot m / m^2$	IEC 80000-6, 6-25.a [23]			FALSE
volt second metre squared per ampere per metre cube	IEC 80000-6, 6-26.a	$\frac{A \cdot s \cdot m^2}{A \cdot m^3}$	IEC 80000-6, 6-26.a [24]			FALSE

Table E-26: Normative units in ISO 80000-6 (3 of 5)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
volt second per ampere per metre	IEC 80000-6, 6-26.a	$\frac{V \cdot s}{A \cdot m}$	IEC 80000-6, 6-26.a [24]			
newton weber per ampere per metre per newton	IEC 80000-6, 6-26.a	$\frac{N \cdot Wb}{A \cdot m \cdot N}$	IEC 80000-6, 6-26.a [24]			FALSE

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
henry per metre	IEC 80000-6, 6-26.a	$\mathrm{H/m}$		IEC 80000-6, 6-26.2 [28]		
weber per ampere per metre	IEC 80000-6, 6-26.a	$\frac{\mathrm{Wb}}{\mathrm{A \cdot m}}$	IEC 80000-6, 6-26.a [24]			
henry per metre ratio	IEC 80000-6, 6-27.a			IEC 80000-6, 6-27 [28]	TRUE	
ampere per metre ratio	IEC 80000-6, 6-28.a			IEC 80000-6, 6-28 [29]	TRUE	
weber per metre squared	IEC 80000-6, 6-29.a		IEC 80000-6, 6-21.a [23]			
volt second ampere per ampere per metre squared	IEC 80000-6, 6-29.a	$\frac{\mathrm{V \cdot s \cdot A}}{\mathrm{A \cdot m^2}}$	IEC 80000-6, 6-21 [23]			FALSE
volt second metre	IEC 80000-6, 6-30.a	$\mathrm{V \cdot s \cdot m}$	IEC 80000-6, 6-30.a [24]			
weber metre	IEC 80000-6, 6-30.a	$\mathrm{Wb \cdot m}$		IEC 80000-6, 6-30 [29]		
weber per metre	IEC 80000-6, 6-32.a	$\mathrm{Wb/m}$		IEC 80000-6, 6-32 [29]		
newton per ampere	IEC 80000-6, 6-32.a	$\mathrm{N/A}$	IEC 80000-6, 6-32.a [24]			
volt second per metre	IEC 80000-6, 6-32.a	$\mathrm{V \cdot s/m}$	IEC 80000-6, 6-32.a [24]			
newton ampere per metre squared	IEC 80000-6, 6-33.a	$\mathrm{N \cdot A/m^2}$	IEC 80000-6, 6-33.a [25]			
newton coulomb per metre squared	IEC 80000-6, 6-33.a	$\mathrm{N \cdot C/m^2}$	IEC 80000-6, 6-33.a [25]			
joule per cubic metre	IEC 80000-6, 6-33.a	$\mathrm{J/m^3}$		IEC 80000-6, 6-33 [29]		

Table E-27: Normative units in ISO 80000-6 (4 of 5)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
newton per metre squared	IEC 80000-6, 6-33.a	N/m^2	IEC 80000-6, 6-33.a [24]			
volt ampere per square metre	IEC 80000-6, 6-34.a	$N \cdot A/m^2$	ISO 80000-5, 5-8.a [15]	IEC 80000-6, 6-34 [29]		
ampere metre per metre	IEC 80000-6, 6-37.a	$A \cdot m/m$	IEC 80000-6, 6-1.a [22]			FALSE
turns	IEC 80000-6, 6-38.a			IEC 80000-6, 6-38 [29]		
ampere per volt per second	IEC 80000-6, 6-39.a	$A/(V \cdot s)$	IEC 80000-6, 6-39.a [25]			
henry to the power minus one	IEC 80000-6, 6-39.a	$1/H$		IEC 80000-6, 6-39 [29]		
volt second per ampere	IEC 80000-6, 6-41.a	$V \cdot s/A$	IEC 80000-6, 6-41.a [25]			
weber per ampere	IEC 80000-6, 6-41.a	Wb/A	IEC 80000-6, 6-41.a [25]			
henry	IEC 80000-6, 6-41.a	H		IEC 80000-6, 6-41.1 [29]		
henry factor squared	IEC 80000-6, 6-42.2			IEC 80000-6, 6-42.2 [29]	TRUE	
henry factor	IEC 80000-6, 6-42.a			IEC 80000-6, 6-42.1 [29]	TRUE	
ampere metre per volt per square metre	IEC 80000-6, 6-43.a	$\frac{A \cdot m}{V \cdot m^2}$	IEC 80000-6, 6-43.a [25]			FALSE
siemens per metre	IEC 80000-6, 6-43.a	S/m		IEC 80000-6, 6-43 [29]		
ampere per volt per metre	IEC 80000-6, 6-43.a	$A/(V \cdot m)$	IEC 80000-6, 6-43.a [25]			
metre per siemens	IEC 80000-6, 6-44	m/S	IEC 80000-6, 6-44.a [25]			

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
ohm metre	IEC 80000-6, 6-44.a	$\Omega \cdot m$		IEC 80000-6, 6-44 [29]		
volt ampere	IEC 80000-6, 6-45.a, 6-57.a, 6-59.a, 6-61.a	$V \cdot A$	ISO 80000-4, 4-26.a, 4-56.a [8]	IEC 80000-6, 6-59 [30]		
ohm	IEC 80000-6, 6-46.a	Ω		IEC 80000-6, 6-46 [30]		
volt per ampere	IEC 80000-6, 6-46.a	V / A	IEC 80000-6, 6-46.a [25]			

Table E-28: Normative units in ISO 80000-6 (5 of 5)

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
siemens to the power minus one	IEC 80000-6, 6-46.a	$1/S$	IEC 80000-6, 6-46.a [25]			
siemens	IEC 80000-6, 6-47.a	S		IEC 80000-6, 6-47 [30]		
ampere per volt	IEC 80000-6, 6-47.a	A/V	IEC 80000-6, 6-47.a [26]			
ohm to the power minus one	IEC 80000-6, 6-47.a	$1/\Omega$	IEC 80000-6, 6-47.a [26]			
ohm ratio	IEC 80000-6, 6-53.a			IEC 80000-6, 6-53 [30]	TRUE	
watt per volt per ampere	IEC 80000-6, 6-58.a			IEC 80000-6, 6-58 [30]	TRUE	
var	IEC 80000-6, 6-60.b	var	IEC 80000-6, 6-45.a, 6-57.a, 6-59.a, 6-61.a [25]	IEC 80000-6, 6-60 [30]		
second joule per second	IEC 80000-6, 6-62.a	s.J/s	ISO 80000-4, 4-27.a, 4-34.a, 4-36.a [8]	IEC 80000-6, 6-62 [31]		FALSE

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
watt hour	IEC 80000-6, 6-62.b	W.h		IEC 80000-6, 6-62 [31]		

Table E-29: Normative quantity kinds in ISO 80000-6 (1 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
electric current in a thin conducting loop n	IEC 80000-6, 6-1	I_n	IEC 80000-6, 6-1 [27]	
electirc current	IEC 80000-6, 6-1	I, i		
rms current	IEC 80000-6, 6-1	I	IEC 80000-6, 6-1 [27]	
electric charge	IEC 80000-6, 6-2	Q, q		
volumic electric charge	IEC 80000-6, 6-3	ρ, ρ_V		
areic electric charge	IEC 80000-6, 6-4	ρ_A, σ		
lineic electric charge	IEC 80000-6, 6-5	ρ_l, τ		
electric dipole moment	IEC 80000-6, 6-6	$\boldsymbol{p}$		
electric polarization	IEC 80000-6, 6-7	$\boldsymbol{P}$		
electric current density	IEC 80000-6, 6-8	$\boldsymbol{J}$		
areic electric current	IEC 80000-6, 6-8	$\boldsymbol{J}$	IEC 80000-6, 6-8 [27]	
lineic electric current	IEC 80000-6, 6-9	$\boldsymbol{J}_s$		
electric field strength	IEC 80000-6, 6-10	$\boldsymbol{E}$		
electric potential	IEC 80000-6, 6-11.1	V, ϕ		
electric potential difference	IEC 80000-6, 6-11.2	V_{ab}	IEC 80000-6, 6-11.1 [27]	
electric tension	IEC 80000-6, 6-11.3	U, U_{ab}	IEC 80000-6, 6-11.1 [27]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
voltage	IEC 80000-6, 6-11.3	U, U_{ab}	IEC 80000-6, 6-11.3 [27]	
rms voltage	IEC 80000-6, 6-11.3	U	IEC 80000-6, 6-11.3 [27]	
electric flux density	IEC 80000-6, 6-12	$\boldsymbol{D}$	IEC 80000-6, 6-7 [27]	
electric flus density in vacuum	IEC 80000-6, 6-12		IEC 80000-6, 6-12 [27]	
capacitance	IEC 80000-6, 6-13	C		
permittivity of vaccum	IEC 80000-6, 6-14.1	ε_0	IEC 80000-6, 6-14.2 [28]	
permittivity	IEC 80000-6, 6-14.2	ε		
relative permittivity	IEC 80000-6, 6-15	ε_r		TRUE
electric susceptibility	IEC 80000-6, 6-16	χ		TRUE
electric flux	IEC 80000-6, 6-17	ψ	IEC 80000-6, 6-2 [27]	
displacement current density	IEC 80000-6, 6-18	$\boldsymbol{J}_D$	IEC 80000-6, 6-20 [25]	
displacement current	IEC 80000-6, 6-19.1	I_D	IEC 80000-6, 6-19.2 [28]	
total current	IEC 80000-6, 6-19.2	$\boldsymbol{I}_{tot}, \boldsymbol{I}_t$	IEC 80000-6, 6-1 [27]	
total current density	IEC 80000-6, 6-20	$\boldsymbol{J}_{tot}, \boldsymbol{J}_t$	IEC 80000-6, 6-8 [27]	

Table E-30: Normative quantity kinds in ISO 80000-6 (2 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
magnetic flux density	IEC 80000-6, 6-21	$\boldsymbol{B}$		
magnetic flux	IEC 80000-6, 6-22.1	Φ		
linked flux in a loop caused by an electric current in that loop	IEC 80000-6, 6-22.2	χ_m, χ	IEC 80000-6, 6-22.2 [28]	
linked flux	IEC 80000-6, 6-22.2	χ		

Quantity Kind name	Description	Symbol	General	is dimension 1?
linked flux in a loop m caused by an electric current in another loop n	IEC 80000-6, 6-22.2	χ	IEC 80000-6, 6-22.2 [28]	
magnetic area moment	IEC 80000-6, 6-23	$\boldsymbol{m}$		
magnetization	IEC 80000-6, 6-24	$\boldsymbol{M}, \boldsymbol{H}_{\mathrm{i}}$	IEC 80000-6, 6-25 [28]	
magnetic field strength in vacuum	IEC 80000-6, 6-25	$\boldsymbol{H}_0$	IEC 80000-6, 6-25 [28]	
magnetic field strength	IEC 80000-6, 6-25	$\boldsymbol{H}$		
permeability of vacuum	IEC 80000-6, 6-26.1	μ_0	IEC 80000-6, 6-26.2 [28]	
permeability	IEC 80000-6, 6-26.2	μ	IEC 80000-6, 6-26.2 [28]	
magnetic flux density of magnetic field strength	IEC 80000-6, 6-26.2			
relative permeability	IEC 80000-6, 6-27	μ_{r}		TRUE
magnetic susceptibility	IEC 80000-6, 6-28	$\kappa, \left(\chi_{\mathrm{m}}\right)$		TRUE
magnetic polarization	IEC 80000-6, 6-29	$\boldsymbol{J}_{\mathrm{m}}$		
magnetic dipole moment	IEC 80000-6, 6-30	$\boldsymbol{j}_{\mathrm{m}}\, \boldsymbol{j}$		
coercivity	IEC 80000-6, 6-31		IEC 80000-6, 6-25 [28]	
magnetic vector potential	IEC 80000-6, 6-32	A		
energy density of electric field	IEC 80000-6, 6-33		IEC 80000-6, 6-33 [29]	
energy density of magnetic field	IEC 80000-6, 6-33		IEC 80000-6, 6-33 [29]	
electromagnetic energy density	IEC 80000-6, 6-33	w		
Poynting vector	IEC 80000-6, 6-34	$\boldsymbol{S}$		
phase speed of electromagnetic waves	IEC 80000-6, 6-35.1	c	ISO 80000-3, 3-8.2 [5]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
phase speed of light in vacuum	IEC 80000-6, 6-35.2	c_0	IEC 80000-6, 6-35.1 [29]	
source voltage	IEC 80000-6, 6-36	U_s	IEC 80000-6, 6-11.3 [27]	
scalar magnetic potential	IEC 80000-6, 6-37.1	V_m, ϕ	IEC 80000-6, 6-1 [27]	
magnetic tension	IEC 80000-6, 6-37.2	U_m	IEC 80000-6, 6-1 [27]	
magnetomotive force	IEC 80000-6, 6-37.3	F_m	IEC 80000-6, 6-1 [27]	
current linkage	IEC 80000-6, 6-37.4	Θ	IEC 80000-6, 6-1 [27]	

Table E-31: Normative quantity kinds in ISO 80000-6 (3 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
number of turns in a winding	IEC 80000-6, 6-38	N		
reluctance	IEC 80000-6, 6-39	R_m, R		
permeanance	IEC 80000-6, 6-40	Λ		
inductance	IEC 80000-6, 6-41.1	L, L_m		
mutual inductance	IEC 80000-6, 6-41.1	L_{mn}	IEC 80000-6, 6-41.1 [29]	
self inductance	IEC 80000-6, 6-41.1	L_n	IEC 80000-6, 6-41.1 [29]	
coupling factor	IEC 80000-6, 6-42.1	k		
leakage factor	IEC 80000-6, 6-42.2	σ		
conductivity	IEC 80000-6, 6-43	σ, γ		
resistivity	IEC 80000-6, 6-44	ρ		
electric power	IEC 80000-6, 6-45	p	ISO 80000-4, 4-26 [13]	
electric resistance	IEC 80000-6, 6-46	R		

Quantity Kind name	Description	Symbol	General	is dimension 1?
electric conductance	IEC 80000-6, 6-47	G		
initial phase of electric voltage	IEC 80000-6, 6-48	ϕ_u	ISO 80000-3, 3-5 [5]	
phase difference	IEC 80000-6, 6-48	ϕ	ISO 80000-3, 3-5 [5]	
initial phase of electric current	IEC 80000-6, 6-48	ϕ_u	ISO 80000-3, 3-5 [5]	
electric current phasor	IEC 80000-6, 6-49	$\underline{I}$	IEC 80000-6, 6-1 [27]	
voltage phasor	IEC 80000-6, 6-50	$\underline{U}$	IEC 80000-6, 6-11.3 [27]	
complex impedance	IEC 80000-6, 6-51.1	$\underline{Z}$	IEC 80000-6, 6-46 [30]	
resistance to alternating electric current	IEC 80000-6, 6-51.2	R	IEC 80000-6, 6-51.1 [30]	
reactance to alternative electric current	IEC 80000-6, 6-51.3	X	IEC 80000-6, 6-51.1 [30]	
modules of impedance	IEC 80000-6, 6-51.4	Z	IEC 80000-6, 6-51.1 [30]	

Table E-32: Normative quantity kinds in ISO 80000-6 (4 of 4)

Quantity Kind name	Description	Symbol	General	is dimension 1?
complex admittance	IEC 80000-6, 6-52.1	$\underline{Y}$		
conductance to alternating current	IEC 80000-6, 6-52.2	G	IEC 80000-6, 6-52.1 [30]	
susceptance to alternating current	IEC 80000-6, 6-52.3	B	IEC 80000-6, 6-52.1 [30]	
modules of admittance	IEC 80000-6, 6-52.4	Y	IEC 80000-6, 6-52.1 [30]	
quality factor	IEC 80000-6, 6-53	Q		TRUE
loss factor	IEC 80000-6, 6-54	d		TRUE
loss angle	IEC 80000-6, 6-55	δ	ISO 80000-3, 3-5 [5]	TRUE
active power	IEC 80000-6, 6-56	P	IEC 80000-6, 6-59 [30]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
apparent power	IEC 80000-6, 6-57	S		
power factor	IEC 80000-6, 6-58	λ		TRUE
complex power	IEC 80000-6, 6-59	S		
reactive power	IEC 80000-6, 6-60	Q	IEC 80000-6, 6-59 [30]	
non-active power	IEC 80000-6, 6-61	Q'	IEC 80000-6, 6-56 [30]	
active energy	IEC 80000-6, 6-62	W		

E.6.5.5 ISO 80000-7 Light

The subset of the normative contents of ISO 80000-7 is identical to that of SysML 1.4 as summarized below.

Table E-33: Units in ISO 80000-7

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
refractive index	ISO 80000-7, 7-5.a			ISO 80000-7, 7-5 [33]	TRUE	
lumen	ISO 80000-7, 7-32.a	lm		ISO 80000-7, 7-32 [33]		
candela	ISO 80000-7, 7-35.a	cd		ISO 80000-7, 7-35 [33]		
lux	ISO 80000-7, 7-36.a	lx		ISO 80000-7, 7-36 [33]		
candela per square metre	ISO 80000-7, 7-37.a	cd/m^2		ISO 80000-7, 7-37 [33]		

Table E-34: Quantity kinds in ISO 80000-7

Quantity Kind name	Description	Symbol	General	is dimension 1?
speed of light in vacuum	ISO 80000-7, 7-4.1	c_0	ISO 80000-3, 3-8.1 [5]	

Quantity Kind name	Description	Symbol	General	is dimension 1?
phase speed of light in medium	ISO 80000-7, 7-4.2	c	ISO 80000-3, 3-8.2 [5]	
refractive index	ISO 80000-7, 7-5	n		TRUE
radiant flux	ISO 80000-7, 7-13		ISO 80000-4, 4-26 [13]	
luminous flux	ISO 80000-7, 7-32	$\Phi_v(\Phi)$		
luminous intensity	ISO 80000-7, 7-35	$I_v(I)$		
illuminance	ISO 80000-7, 7-36	$E_v(E)$		
luminance	ISO 80000-7, 7-37	$L_v(L)$		

E.6.5.6 ISO 80000-9 Physical Chemistry and Molecular Physic

The subset of the normative contents of ISO 80000-9 is identical to that of SysML 1.4 as summarized below.

Table E-35: Units in ISO 80000-9

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
mole	ISO 80000-9, 9-1.a	mol		ISO 80000-9, 9-1		
mole per cubic metre	ISO 80000-9, 9-13.a	mol/m3		ISO 80000-9, 9-13		

Table E-36: Quantity kinds in ISO 80000-9

Quantity Kind name	Description	Symbol	General	is dimension 1?
amount of substance	ISO 80000-9, 9-1	n		
amount of substance concentration	ISO 80000-9, 9-13	c_B		

E.6.5.7 ISO 80000-10 Atomic and Nuclear Physics

The 3 units and 3 quantity kind definitions included were in the non-normative ISO 80000-10 library of SysML 1.3.

Table E-37: Units in ISO 80000-10

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
becquerel	ISO 80000-10, 10-29.a	Bq		ISO 80000-10, 10-29		
gray	ISO 80000-10, 10-84.a	Gy		ISO 80000-10, 10-84		
sievert	ISO 80000-10, 10-86.a	Sv		ISO 80000-10, 10-86		

Table E-38: Quantity kinds in ISO 80000-10

Quantity Kind name	Description	Symbol	General	is dimension 1?
radionuclide activity	ISO 80000-10, 10-29	A		
absorbed dose	ISO 80000-10, 10-81.1	D		
dose equivalent	ISO 80000-10, 10-86	H		

E.6.5.8 ISO 80000-13 Information Science and Technology

SysML 1.4 adds commonly used 3 units (bit, byte and octet) of information and 3 of their associated quantity kinds.

Table E-39: Units in ISO 80000-13

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
bit	IEC 80000-13, 13-9.b	bit		IEC 80000-13, 13-9	TRUE	
byte	IEC 80000-13, 13-9.c	B		IEC 80000-13, 13-9	TRUE	

Unit name	Description	Symbol	General units	Quantity Kinds	is unit for quantity of dimension 1?	is reduced form?
octet	IEC 80000-13, 13-9.c	o		IEC 80000-13, 13-9	TRUE	

Table E-40: Quantity kinds in ISO 80000-13

Quantity Kind name	Description	Symbol	General	is dimension 1?
storage capacity	IEC 80000-13, 13-9	M		TRUE
storage size	IEC 80000-13, 13.9	M	IEC 80000-13, 13-9	TRUE
equivalent binary storage capacity	IEC 80000-13, 13-10	M_e		TRUE

E.7 Distribution Extensions

E.7.1 Overview

This sub clause describes a non-normative extension to provide a candidate set of distributions (see 8.3.2.9, DistributedProperty). It consists of a profile containing stereotypes that can be used to specify distributions for properties of blocks.

E.7.2 Stereotypes

E.7.2.1 Package Distributions

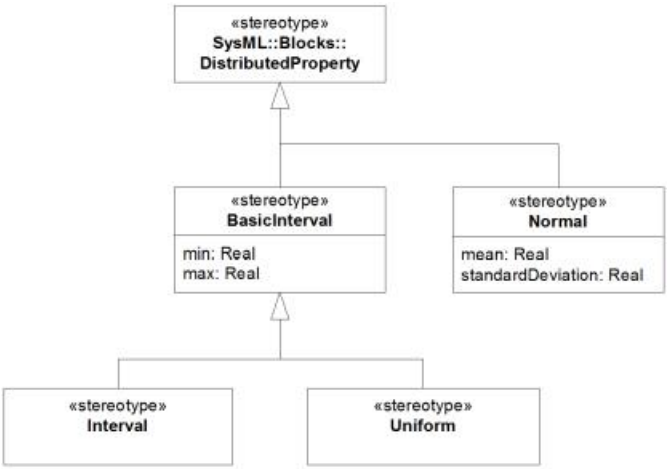


Figure E.21 Basic distribution stereotypes

Table E-41: Distribution Stereotypes

Stereotype	Base Class	Properties	Constraints	Description
«BasicInterval»	«DistributedProperty»	min:Real max:Real	N/A	Basic Interval distribution - value between min and max inclusive
«Interval»	«BasicInterval»	N/A	N/A	Interval distribution - unknown probability between min and max
«Uniform»	«BasicInterval»	N/A	N/A	Uniform distribution - constant probability between min and max
«Normal»	«DistributedProperty»	mean:Real standard Deviation:Real	N/A	Normal distribution - constant probability between min and max

E.7.3 Usage Example

Figure E.22 shows a simple example of using distributions; the force of the Cannon is specified using a Normal distribution with parameters mean and standard deviation. Whereas the use of a Normal distribution can be inferred from

the names of its parameters, an Interval distribution shares parameters with a Uniform distribution, hence the stereotype keyword «interval» is used to distinguish it.

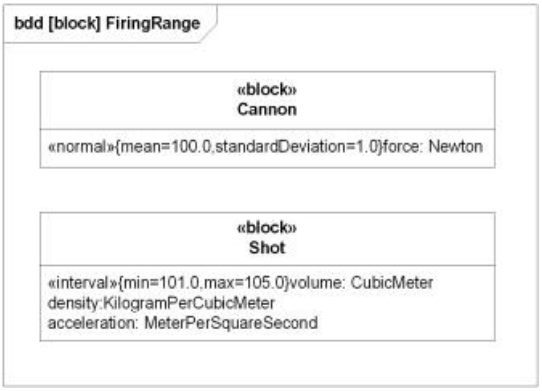


Figure E.22 Distribution Example

E.8 Building Non-normative Extension for Property-based Requirements

E.8.1 Overview

Annex E.3 addresses extending requirements that are fundamentally textual in nature. They may be extended with various enumerations (for example RiskKind or VerifyMethodKind), and they may have different modeling constraints applied to the requirements relationships, but the requirements are only expressed as text strings.

Expressing requirements as text strings alone fundamentally limits their ability to be evaluated and verified. This Annex addresses a more formal expression of requirements generally referred to as property based requirements (PBR); one that includes quantitative specification of numerical parameters, relationships, equations and/or constraints.

Current users of text-based requirements have frequently expressed a basic need to represent numerical requirements more precisely, both to reduce ambiguity and facilitate verification by analysis and other methods. This basic need can be decomposed into three primary needs: 1) Requirements shall have numerical properties (properties capable of representing numerical values), 2) these numerical properties shall be typeable (preferably by ValueType) to account for quantity kind and units, and 3) these numerical properties shall be bindable (preferably using BindingConnector) to other model elements (e.g., ConstraintParameters) so they can be evaluated using analysis tools. For the purpose of this discussion, a requirement that meets these three conditions is said to be a property-based requirement.

This kind of property-based requirement is intended to be used with the overall system model to assist in specifying and architecting systems. More generally, the system model may be used as a model-based specification, such as when block instances with specific property values represent the requirement. In this latter case, the model-based specification can further refine the property-based requirement.

Users of property-based requirements may desire a more elaborate capability than the primary need described above. For example, it may be desirable for the requirement to contain a constraint or mathematical expression that formally states an acceptance condition, threshold, or goal. This may alternatively need to be expressed as a set of valued pairs, elaborating both the conditions and the acceptance thresholds for each condition, or by an arbitrary graphical

relationship. Some users may want the property-based requirement to formally own a behavior representing the functionality of the requirement, or the behavior by which it is satisfied or verified.

The need for this kind of property-based requirement is illustrated in the simple example of specifying a vehicle's required stopping distance for various initial speeds and road conditions. The requirement can be expressed in a table as follows:

The Vehicle stopping distance shall not exceed the values in Table E-42 as a function of initial speed and pavement condition.

Table E-42: Example of Requirement in Tabular Form

Initial Speed (mph)	Pavement Condition (wet/dry)	Req'd Stopping Distance -Dry (feet)	Initial Speed (mph)	Pavement Condition (wet/dry)	Req'd Stopping Distance- Wet (feet)
0	dry	0	0	wet	0
10	dry	4	10	wet	6
20	dry	17	20	wet	22
30	dry	38	30	wet	50
40	dry	67	40	wet	89
50	dry	104	50	wet	139
60	dry	150	60	wet	201
70	dry	205	70	wet	273
80	dry	267	80	wet	357
90	dry	338	90	wet	451
100	dry	418	100	wet	557

An alternative expression in plot format can be:

The Vehicle stopping distance shall not exceed the values in Table E-42 as a function of initial speed and pavement condition.

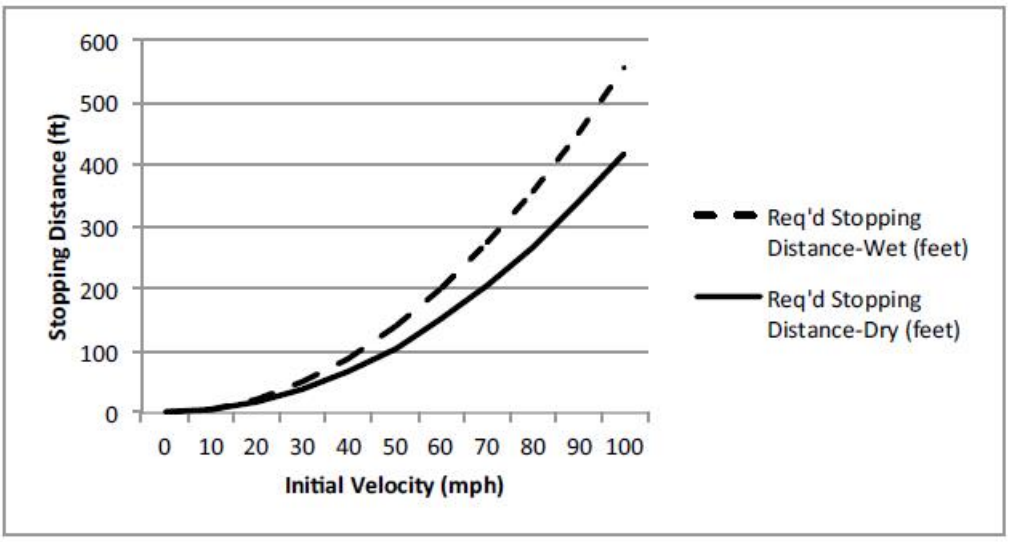


Figure E.23 Example of Requirement in Graphical Form

The input/output parameter relationship or constraint can be specified in equation form, such as in the following example:

Stopping distance <= (1/(2*32.174* alpha)*(580*Initial Speed/3600^2)

State Speed = 0 ...100

	alpha
dry	0.8
wet	0.6

More generally, the input and output parameter values may be complex functions of other parameters, and may have probability distributions associated with them.

This annex addresses mechanisms and approaches for building SysML profiles to enable property-based requirements. While examples of property-based requirement profiles are provided in this annex, these are not to be considered normative or even authoritative. Instead, they are intended to be illustrative of the kind of extensions that some users may find desirable. Ultimate responsibility for the compatibility of any property-based requirement profile with a particular requirements management process or toolset rests fully with the user.

E.8.2 An Example PBR Profile Based on ConstraintBlock

E.8.2.1 Profile/Stereotypes of PBR Based on ConstraintBlock

Table E-24 shows use of both «abstractRequirement» and «constraintBlock» to define a new PBR stereotype, named RequirementConstraintBlock in this example for clarity.

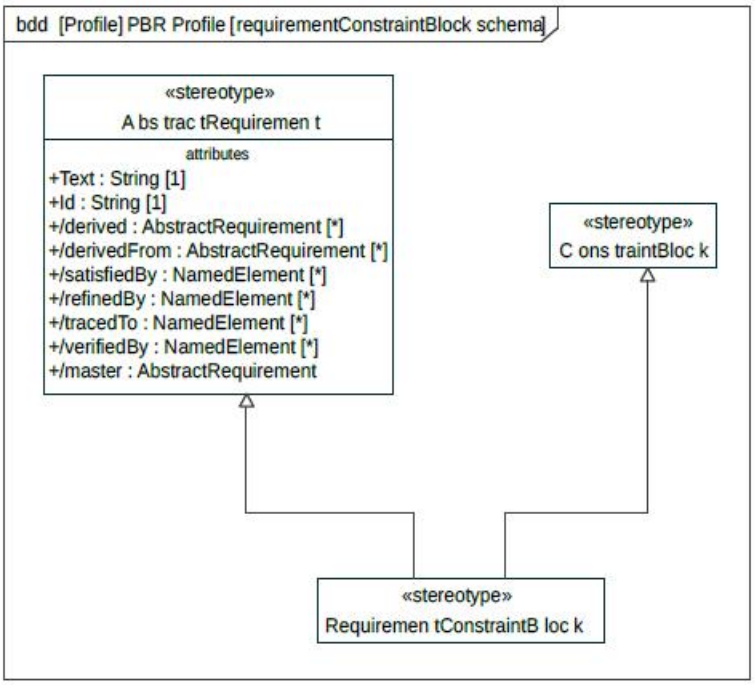


Figure E.24 Example of a PBR Profile Based on ConstraintBlock

Basing PBR on ConstraintBlock provides flexibility in expressing the name of required numerical values as ConstraintParameters, which can be typed by ValueTypes and related to properites or parameters of other model elements using binding connectors. Textual requirement statements may be restated as constraint expressions that reference these ConstraintParameters. The value bindings can then be used to evaluate the constraint expression and determine compliance with the requirement.

The numerical required value may then be stored as a DefaultValue of the ConstraintParameter. It may alternatively be specified directly in a constraint expression, rather than a default value, e.g., {requiredWeight = 1450} where requiredWeight is defined as a constraint parameter typed by a value type. Complex requirement criteria may be represented by a series of constraint expressions.

It is also noted that constraint blocks can have owned behavior, and that a constraint expression can be a value expression (with opaque behavior).

E.8.2.2 Usage Example using PBR Based on ConstraintBlock

The following example leverages the above PBR user profile based on ConstraintBlock to specify and evaluate the weight of a vehicle.

The requirement is captured via a PBR (RequirementConstraintBlock), which includes a constraint expression that reflects the textual requirements statement in terms of two defined parameters, actualMass and requiredMass. Both of these parameters are typed by the kilogram value type from the SI value types library. The required value for mass is expressed as a default value of the requiredMass parameter. Note that the required value may have alternatively been expressed as a second constraint expression, e.g., {requiredMass = 1450}. The vehicle itself is represented in the model by a block with a value property for mass, also typed by the kilogram SI value type.

As shown in Table E-25, the context for evaluating if the requirement has been met is established using a Requirement Context block. This method of context setting is a best practice that is not essential to this example. Both the Vehicle and the Vehicle Mass Requirement are used in this Requirement Context.

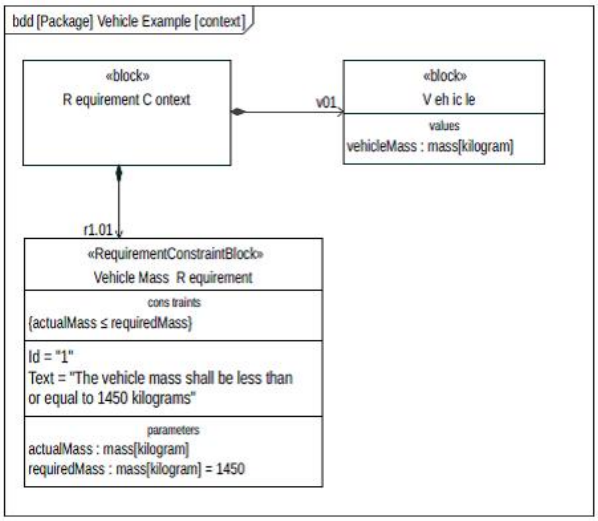


Figure E.25 Example of Requirement Evaluation Context Using PBR Based on Constraint Block

Table E-26 shows a parametric diagram of the Requirement Context block, useful for establishing the method of evaluating compliance of the vehicleMass value with the Vehicle Mass Requirement. As with any parametric model, it does not actually perform the evaluation/analysis, but it does specify the key relationships so that an evaluation tool may determine if the weight requirement has been met.

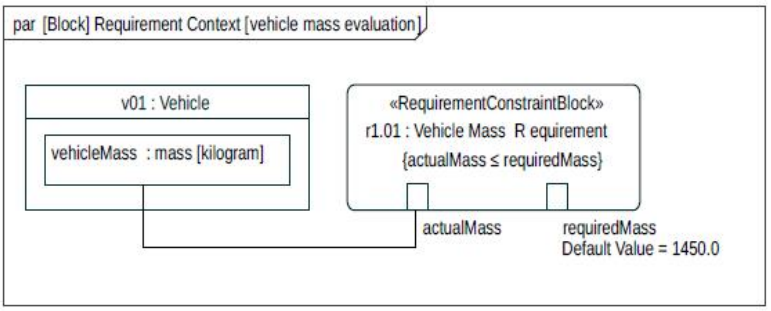


Figure E.26 Example of Parametric Diagram Using PBR based on Constraint Block

E.8.3 An Example PBR Profile Based on Constraint

Constraints are arguably the most straightforward way for representing system requirements. Their specification may be provided by opaque constraint expressions, which can be expressed in formal (and computable) languages like OCL. This allows the constraint statement to be applied directly to a specific design, without necessarily applying a formal evaluation context.

E.8.3.1 Profile/Stereotypes of PBR Based on Constraint

Table E-27shows use of both «abstractRequirement» and «constraint» to define a new PBR stereotype, named CbRequirement in this example.

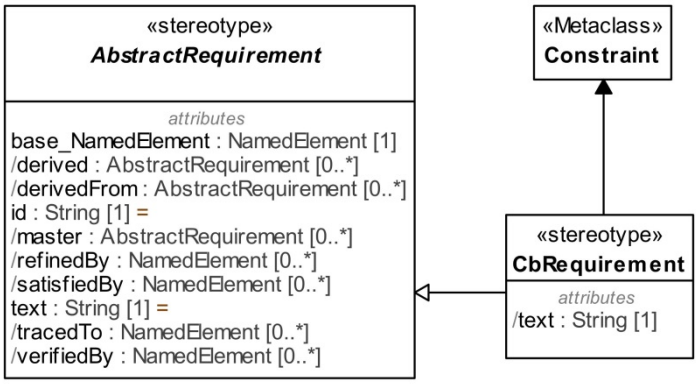


Figure E.27 Example of a PBR profile based on Constraints

E.8.3.2 Example using PBR profile Based on Constraint

Table E-28 shows how requirements are specified on the model representing a specification. Note that, as modeled here, the requirement represented by Constraint2 applies to any instance of the Vehicle block while the one represented by Constraint1 applies to instances of Vehicle block which are "used" as defined by the "vehicle" role of the Context block, such as the design weight of the vehicle on a bridge or vehicle transporter.

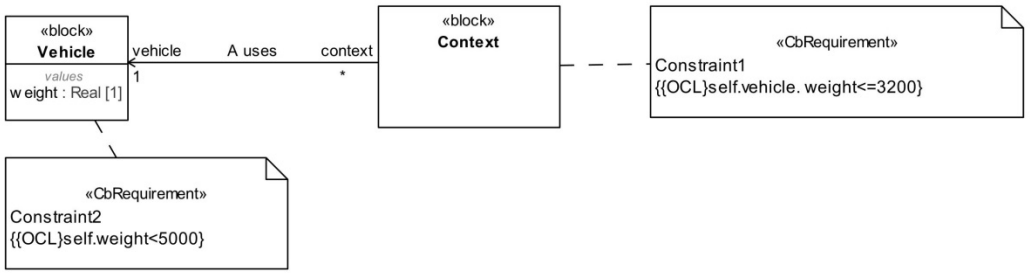


Figure E.28 Example of PBR based on Constraint used in different contexts

Table E-29 shows a particular case where testedVehicle is an instance of the Vehicle block and AnalysisContext an instance of the Context block, as specified above. A simple evaluation of model constraints using a classical OCL evaluator would produce a report showing that Requirement/Constraint2 is met, while Requirement/Constraint1 is violated.

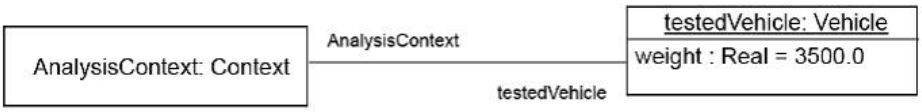


Figure E.29 Establishing an Analysis Context for evaluating requirement compliance using PBR based on Constraint

E.8.4 An Example Property Based Requirement based on Block

Property based requirements can be based on a Block which allows to define additional properties like value properties.

Table E-30 shows use of both “abstractRequirement” and “Block” to define a new PBR stereotype, named «PBR» in this example.

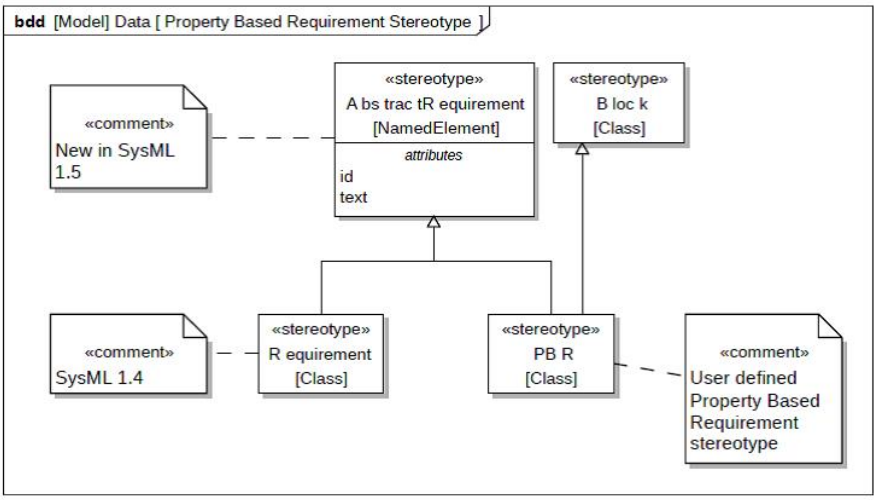


Figure E.30 PBR Example

Table E-31gives an example where a requirement element “Max Peak Power Requirement is created. It defines “id,” “text,” and “maxPeakPwr.”

It also has additionally a constraint property “maxPower” which permits to define constraints for the value properties. The requirement is contextualized in the block “System Specification.” The block “Verification Context” contextualizes the block “System Design” which holds the as-designed “totalPower” value property. In this context the as-designed value is bound to the requirement constraint for the purpose of analysis to verify that the designed value satisfies the required value.

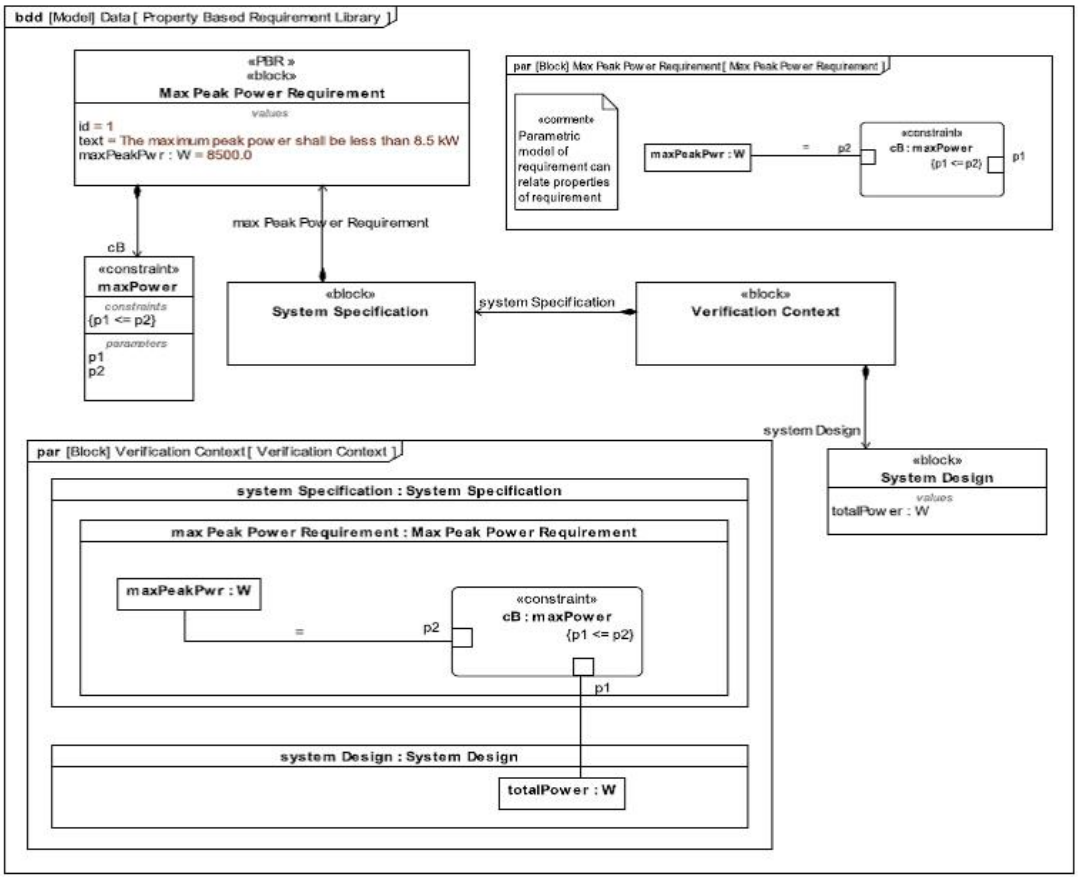


Figure E.31 PBR

Annex F: Requirements Traceability

(Informative)

The OMG SysML requirements traceability matrix traces this International Standard to the original source requirements in the UML for Systems Engineering RFP (ad/2003-03-41). The traceability matrix is included by reference in a separate document (ptc/2007-03-09).

This page intentionally left blank.

Annex G: Model Interchange

(Informative)

G.1 Overview

This annex describes two methods for exchanging SysML models between tools. The first method discussed is XML Metadata Interchange (XMI), which is the preferred method for exchanging models between UML-based tools. The second approach describes the use of ISO 10303-233 Application Protocol: Systems engineering (AP233), which is one of the series of STEP (Standard for the Exchange of Product Model Data) engineering data exchange standards. Other model interchange approaches are possible, but the ones described in this annex are expected to be the primary ones supported by SysML.

G.2 Context for Model Interchange

Developing today's complex systems typically requires engineering teams that are distributed in time and space and that are often composed of many companies, each with their own culture, methods, and tools. Effective collaboration requires agreement on, and a thorough understanding of, the various work assignments and resulting artifacts.

Many of these artifacts pertain to shared engineering data (e.g., requirements, system structural and behavioral models, verification & validation) that transcend the entire life cycle of the system of interest and are the basis for important systems engineering considerations and decisions. So it is critical that the system information contained in these artifacts and information models be accurately captured and readable by all appropriate team members in a timely manner.

Today, this information resides in an array of tools where each is only concerned with a portion of systems engineering data and can't share its data with other tools because they only understand their own native schema. To mitigate this situation, collaborating organizations are usually forced to either adopt a common set of tools or develop a unique, bidirectional interface between many of the tools that each organization uses. This can be an expensive and untimely approach to data exchange between team members. So, there is a need to define standardized approaches for model interchange between the different data schemas in use.

G.3 XMI Serialization of SysML

UML 2.5.1 is formally defined using the OMG Meta Object Facility (MOF). MOF can be considered a language for specifying modeling languages. The OMG XML Metadata Interchange (XMI) 2.5.1 standard specifies an XML-based interchange format for any language modeled using MOF. This results in a standard, convenient format for serializing UML user models as XMI files for interchange between UML tools. The XMI specification also includes rules for generating an XML Schema that can be used for basic validation of the structure of those UML user model XMI files.

The UML language includes an extension mechanism called UML Profiles. UML Profiles are themselves defined as UML models (MOF is not used). However, their intent is to specify extensions to the UML language semantics in much the same way one could extend the UML language by adding to the MOF definition of UML. As UML Profiles are valid UML models, XMI does provide a mechanism for exchanging the UML Profiles between UML tools. However, as they are extensions to concepts defined in the UML language itself, the definition of a UML Profile refers to the UML language definitions. An XMI 2.5.1 representation of the SysML profile (i.e., the UML Profile for SysML), as well as XMI 2.5.1 representations of Model Libraries defined by SysML, are provided as support documents to this International Standard. As with UML, XMI provides a convenient serialized format for model interchange between SysML tools and basic validation of those files using an XML Schema as well.

The namespace for the standard profile is: https://www.omg.org/spec/SysML/20181001/SysML.xmi.

G.4 SysML Model Interchange Using AP233

AP233 is a data exchange standard designed to support the exchange of systems engineering data between the many and varied software tools that systems engineers use in the course of their work. Data from systems modeling tools is included in the scope of AP233, in fact, requirements for AP233 and SysML have been largely aligned by the OMG and the ISO teams working together and in close cooperation with the INCOSE Model Driven System Design working group.

G.4.1 Scope of AP233

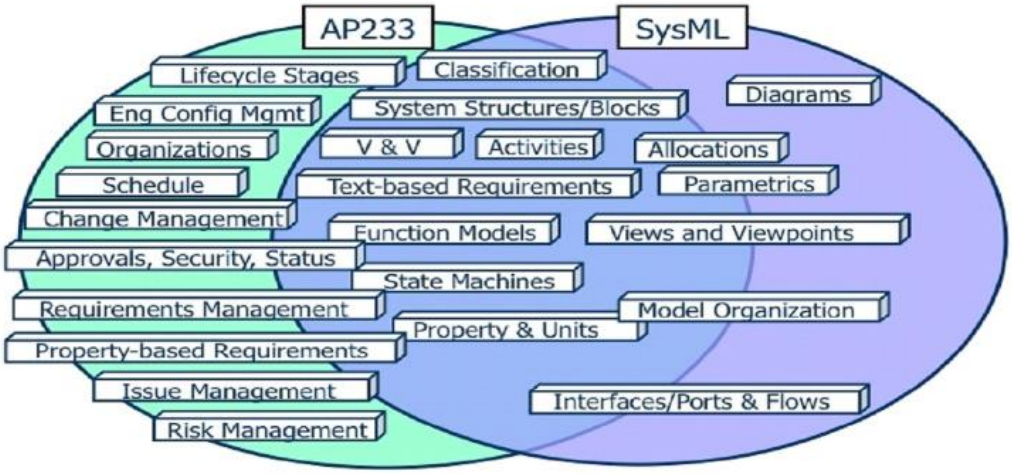


Figure G.1: SysML/AP233 Data Overlaps

AP233 includes support for assigning program management information as well as system modeling information to systems engineering data.

Program management capabilities include issue management, risk management and aspects of project management such as project breakdown, project resource information, organization structure, schedule, and work structure.

System modeling capabilities include requirements and requirements allocation, trade studies with measures of effectiveness, interface to analysis, function-based behavior, state-based behavior, system hierarchies for the design system, the realized system and all interfaces.

Additional information about AP233 can be found at http://www.ap233.org/.

G.4.2 STEP Architecture

AP233 is standardized under ISO Technical Committee 184 (Industrial Automation Systems and Integration), Subcommittee 4 (Industrial Data). AP233 is part of the family of ISO 10303 standards, referred to as STEP, that include standardized models and infrastructure for the exchange of product model data.

The STEP architecture is modular. This enables the component information models to be reused across disciplines and life-cycle stages in different application protocols, which are the models used for implementation. STEP models are written using the ISO 10303-11 EXPRESS language.

STEP also standardizes a series of implementation methods: a text file structure (ISO 10303-21), a data access interface (ISO 10303-22) and an XML file format (ISO 10303-28). The data access interface has bindings that provide standardized APIs for accessing EXPRESS-based data in various programming languages. A conforming STEP implementation is the combination of a STEP application protocol and one or more of the implementation methods.

The scope of STEP is very large and a number of data exchange standards (e.g., geometry, product life-cycle support, structural, electrical, and engineering analysis) have been in wide use in industry for more than 15 years. Support for several systems engineering viewpoints within the scope of AP233 are shared with other application protocols. Since AP233 is part of STEP, it is easy to relate systems engineering data to that of other engineering disciplines over the life cycle of a system and to related product models.

For more information on the STEP architecture see the ISO TC184/SC4 Industrial Data subcommittee web page at http://www.tc184-sc4.org.

G.4.3 EXPRESS

AP233, like all STEP application protocols, is defined using the EXPRESS modeling language. EXPRESS is a precise text-based information modeling language with a related graphical representation called EXPRESS-G.

An example of the text-based format follows:

```
SCHEMA Ap233_systems_engineering_arm_excerpt;

ENTITY Product;

id : STRING;

name : STRING;

description : OPTIONAL STRING;

END_ENTITY;


ENTITY Product_version;

id : STRING;

description : OPTIONAL STRING;

of_product : Product;

END_ENTITY;


ENTITY Product_view_definition;
```

```
id : OPTIONAL STRING;
name : OPTIONAL STRING;
additional_characterization : OPTIONAL STRING;
initial_context : View_definition_context;
additional_contexts : SET [0:?] OF View_definition_context;
defined_version : Product_version;
WHERE
WR1: NOT (initial_context IN additional_contexts);
WR2: EXISTS(id) OR (TYPEOF(SELF\Product_view_definition) <> TYPEOF(SELF));
END_ENTITY;

ENTITY View_definition_context;
application_domain : STRING;
life_cycle_stage : STRING;
description : OPTIONAL STRING;
WHERE
WR1: (SIZEOF (USEDIN(SELF, AP233_SYSTEMS_ENGINEERING_ARM_EXCERPT. +
PRODUCT_VIEW_DEFINITION.INITIAL_CONTEXT )) > 0) OR
(SIZEOF (USEDIN(SELF, AP233_SYSTEMS_ENGINEERING_ARM_EXCERPT. +
PRODUCT_VIEW_DEFINITION.ADDITIONAL_CO NTEXTS)) > 0);
END_ENTITY;

ENTITY System
SUBTYPE OF (Product);
END_ENTITY;

ENTITY System_version
SUBTYPE OF (Product_version);
SELF\Product_version.of_product : System;
```

```
END_ENTITY;


ENTITY System_view_definition

SUBTYPE OF (Product_view_definition);

SELF\Product_view_definition.defined_version : System_version;

END_ENTITY;


END_SCHEMA;
```

EXPRESS expressions are similar in nature to OCL expressions and the two languages have similar expressiveness. EXPRESS has also been approved as an OMG standard with a September 2009 publication of Version 1.0 of the Reference Metamodel for the EXPRESS Information Modeling Language Specification. This will enable the use of OMG Model Driven Architecture technologies against AP233 and other STEP models written in EXPRESS.

G.4.4 SysML-AP233 Mapping

A formal and standardized mapping between SysML and AP233 is being developed within the OMG. The mapping is a specification for SysML and other tool vendors to implement so that their tools can import from and export to AP233 data exchange files. AP233 usage is aimed primarily at scenarios where SysML data is fed to downstream applications such as those used in manufacturing, life cycle management or systems maintenance. Additional information can be found at the OMG SysML Portal at http://www.omgwiki.org/OMGSysML/.

This page intentionally left blank.