OMG SysML Specification

This OMG document replaces the submission (ad/06-03-01) and the draft adopted specification (ptc/2006-05-03). It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by August 1, 2006.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on April 6, 2007. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

OMG Adopted Specification
ptc/06-05-04

Date: May 2006

OMG Systems Modeling Language (OMG SysML™) Specification

Final Adopted Specification
ptc/06-05-04

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

This document describes a language specification developed by an informal partnership of vendors and users, with input from additional reviewers and contributors. This document does not represent a commitment to implement any portion of this specification in any company's products. See the full text of this document for additional disclaimers and acknowledgments. The information contained in this document is subject to change without notice.

The specification customizes the Unified Modeling Language (UML) specification of the Object Management Group (OMG) to address the requirements of Systems Engineering as specified in the UML for Systems Engineering RFP, OMG document number ad/2003-03-41. This document includes references to and excerpts from the *UML 2.0 Superstructure Specification* (OMG document number Formal/05-07-04) and *UML 2.0 Infrastructure Specification* (OMG document number ptc/04-10-14) with copyright holders and conditions as noted in those documents.

LICENSES

Redistribution and use of this specification, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of this specification must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- The Copyright Holders listed in the above copyright notice may not be used to endorse or promote products derived from this specification without specific prior written permission.
- All modified versions of this specification must include a prominent notice stating how and when the specification was modified.

TRADEMARKS

Systems Modeling Language and SysML, which are used to identify this specification, are not usable as trademarks since SysML Partners has established their usage to identify this specification without any trademark status or restriction. Organizations that wish to establish trademarks related to this specification should distinguish them somehow from SysML and Systems Modeling Language, for example by adding a unique prefix (e.g., OMG SysML).

Unified Modeling Language and UML are trademarks of the OMG. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

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.

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, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ , OMG SysML™, and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The 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 *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

List of Figures

List of Tables

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

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

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
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*

Typographical Conventions

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

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

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

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

Helvetica/Arial - 10 pt: Exceptions

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

Part I - Introduction

This specification defines a general-purpose modeling language for systems engineering applications, called the OMG Systems Modeling Language (OMG SysML™). Throughout the rest of the specification, the language will be referred to as SysML.

SysML supports the specification, analysis, design, verification and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel, and facilities.

The origins of the SysML initiative can be traced to a strategic decision by the International Council on Systems Engineering's (INCOSE) Model Driven Systems Design workgroup in January 2001 to customize the Unified Modeling Language (UML) for systems engineering applications. This resulted in a collaborative effort between INCOSE and the Object Management Group (OMG), which maintains the UML specification, to jointly charter the OMG Systems Engineering Domain Special Interest Group (SE DSIG) in July 2001. The SE DSIG, with support from INCOSE and the ISO AP 233 workgroup, developed the requirements for the modeling language, which were subsequently issued by the OMG as part of the UML for Systems Engineering Request for Proposal (UML for SE RFP; OMG document ad/03-03-41) in March 2003.

Currently it is common practice for systems engineers to use a wide range of modeling languages, tools and techniques on large systems projects. In a manner similar to how UML unified the modeling languages used in the software industry, SysML is intended to unify the diverse modeling languages currently used by systems engineers.

SysML reuses a subset of UML 2.1 and provides additional extensions needed to address the requirements in the UML for SE RFP. SysML uses the UML 2.1 extension mechanisms as further elaborated in Chapter 17, "Profiles & Model Libraries" of this specification as the primary mechanism to specify the extensions to UML 2.1.

Since SysML uses UML 2.1 as its foundation, systems engineers modeling with SysML and software engineers modeling with UML 2.1 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, communications and information systems.

1 Scope

The purpose of this document is to specify Systems Modeling Language (SysML), a new general-purpose modeling language for systems engineering that satisfies the requirements of the UML for SE RFP. 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.

SysML reuses a subset of UML 2.1 and provides additional extensions to satisfy the requirements of the language. This specification documents the language architecture in terms of the parts of UML 2.1 that are reused and the extensions to UML 2.1. The specification includes the concrete syntax (notation) for the complete language and specifies the extensions to UML 2.1. The reusable portion of the UML 2.1 specification is not included directly in the specification but is included by reference. The specification also provides examples of how the language can be used to solve common systems engineering problems.

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, and 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. The initial version of the language supports most, but not all of the requirements of the UML for SE RFP, as shown in the Requirements Traceability Matrix in Annex E. These gaps are intended to be addressed in future versions of SysML as indicated in the matrix.

SysML is intended to be supported by two evolving interoperability standards: the OMG XMI 2.1 model interchange standard for UML 2.1 modeling tools and the ISO 10303-233 data interchange standard for systems engineering tools. While the details of this alignment are beyond the scope of this specification, overviews of the alignment approach and relevant references are furnished in Annex D.1.

The following sections provide background information about this specification. Instructions to either systems engineers and vendors who read this specification are provided in Section 3.2, “How to Read this Specification. The main body of this document (Parts II-IV) describes the normative technical content of the specification. The appendices include additional information to aid in understanding and implementation of this specification.

2 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. Refer to the OMG site for subsequent amendments to, or revisions of any of these publications.

- UML for Systems Engineering RFP (OMG document number ad/2003-03-41)
- UML 2.0 Superstructure Specification (OMG document number formal/05-07-04)
- UML 2.1 Superstructure Specification convenience document (OMG document number ptc/06-01-02)
- UML 2.0 Infrastructure Specification (OMG document number ptc/04-10-14)
- XMI 2.1 Specification (OMG document number formal/2005-09-01)

3 Additional Information

3.1 Relationships to Other Standards

SysML is defined as an extension of the OMG *UML 2.1 Superstructure Specification*.

SysML is intended to be supported by two evolving interoperability standards including the OMG XMI 2.1 model interchange standard for UML 2.1 modeling tools and the ISO 10303-233 data interchange standard for systems engineering tools. The overviews of the approach to model interchange and relevant references are included in Annex D.1.

SysML supports the OMG's Model Driven Architecture initiative by its reuse of the UML and related standards.

3.2 How to Read this Specification

This specification is intended to be read by systems engineers so that they may learn and apply SysML, and by modeling tool vendors so that they may implement and support SysML. As background, all readers are encouraged to first read Part I "- Introduction".

After reading the introduction, readers should be prepared to explore the user-level constructs defined in the next three parts: Part II - "Structural Constructs", Part III - "Behavioral Constructs", and Part IV - "Crosscutting Constructs". Systems engineers should read the Overview, Diagram Elements and Usage Examples sections in each chapter, and explore the UML Extensions as they see fit. Modeling tool vendors should read all sections. In addition, systems engineers and vendors should read Annex B - "Sample Problem" to understand how the language is applied to an example, and "Annex E: Requirements Traceability" to understand how the requirements in the UML for SE RFP are satisfied by this specification.

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

3.3 Acknowledgements

The following companies submitted or supported parts of this specification:

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 specification: 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, Veronique 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 specification: 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 2.1 that was adapted for most of the figures throughout this specification.

4 Language Architecture

SysML reuses a subset of UML 2.1 and provides additional extensions needed to address the requirements in the UML for SE RFP. This specification documents the language architecture in terms of the parts of UML 2.1 that are reused and the extensions to UML 2.1. This chapter explains design principles and how they are applied to define the SysML language architecture.

In order 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 cross-hatched region marked “UML reused by SysML,” indicates the UML modeling constructs that SysML re-uses. The compliance matrix in Table 4.1 below specifies the UML packages that a SysML tool must reuse in order to implement SysML.

The region marked “SysML extensions to UML” in Figure 4.1 indicates the new modeling constructs defined for SysML which have no counterparts in UML, or replace UML constructs. Note that there is also a part of UML 2.1 that is not required to implement SysML, which is shown by the region marked “UML not required by SysML.”

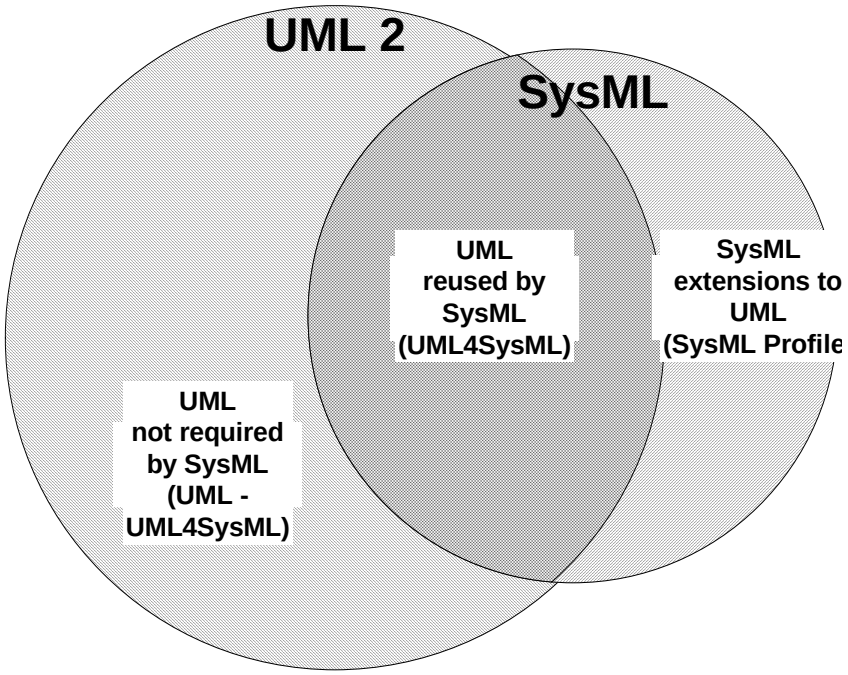


Figure 4.1 - Overview of SysML/UML Interrelationship

4.1 Design Principles

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 2.1 or later versions.

- UML extensions. SysML extends UML as needed to satisfy the requirements of the RFP. The primary extension mechanism is the UML 2.1 profile mechanism as further refined in Chapter 17, “Profiles & Model Libraries” of this specification.
- Partitioning. The package is the basic unit of partitioning in this specification. The packages partition the model elements into logical groupings which 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.

4.2 Architecture

The SysML language reuses and extends many of the packages from UML. As shown in Figure 4.2, the set of UML metaclasses to be reused are merged into a single metamodel package, UML4SysML. The detailed list of packages that are merged are shown in Table 4.1. Some UML packages are not being reused, since they are not considered essential for systems engineering applications to meet the requirements of the UML for SE RFP.

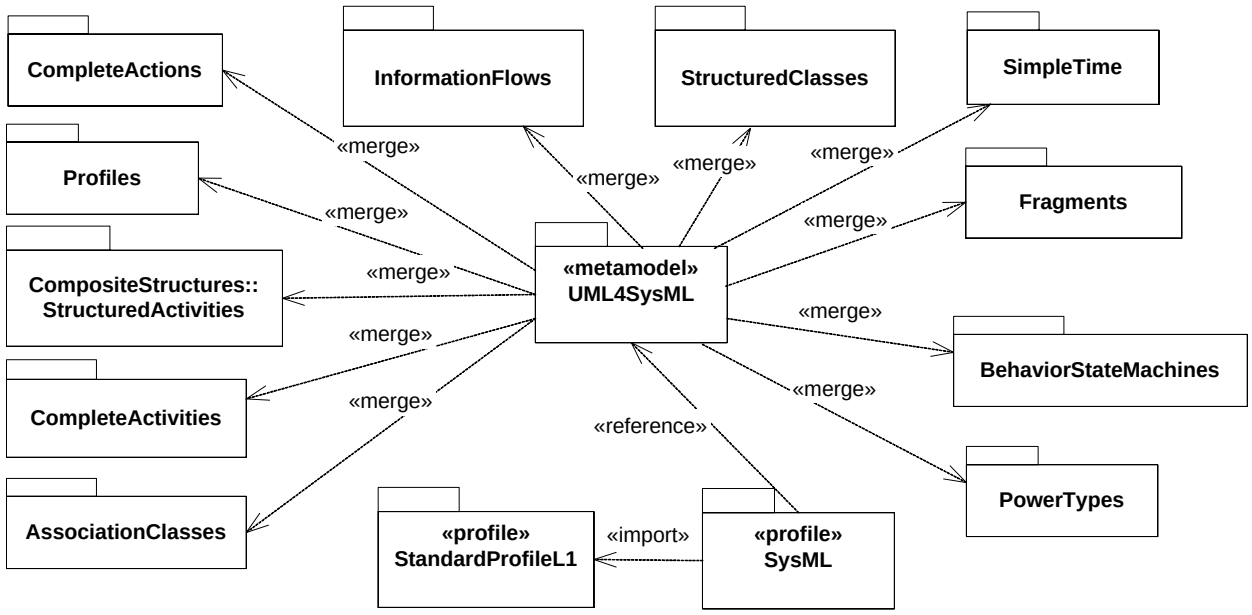


Figure 4.2 - SysML Extension of UML

The SysML profile specifies the extensions to UML. It references the UML4SysML package, thus importing all the metaclasses into SysML that are either reused as-is from UML or extended in SysML. The semantics of UML profiles ensures that when a user model “strictly” applies the SysML profile, only the UML metaclasses referenced by SysML are

available to the user of that model. If the profile is not “strictly” applied, then additional UML metaclasses which were not explicitly referenced may also be available. The SysML profile also imports the Standard Profile L1 from UML to make use of its stereotypes.

Table 4.1 - Detail of UML Reuse

UML Language Unit	UML Package	Metaclasses
Actions	Actions::BasicActions	All
	Actions::StructuredActions	All
	Actions::IntermediateActions	All
	Actions::CompleteActions	All
Activities	Activities::FundamentalActivities	All
	Activities::BasicActivities	All
	Activities::IntermediateActivities	All
	Activities::StructuredActivities	All
	Activities::CompleteActivities	All
Classes	Classes::Kernel	All
	Classes::Dependencies	All
	Classes::Interfaces	All
	Classes::PowerTypes	All
	Classes::AssociationClasses	All
General Behavior	CommonBehaviors::BasicBehaviors	All
	CommonBehaviors::SimpleTime	All
Information Flows	AuxiliaryConstructs::InformationFlows	All
Interactions	Interactions::BasicInteractions	All
	Interactions::Fragments	All
Models	AuxiliaryConstructs::Models	All
Profiles	AuxiliaryConstructs::Profiles	All
State Machines	StateMachines::BehaviourStateMachines	All
Structures	CompositeStructures::InternalStructures	All
	CompositeStructures::StructuredClasses	All
	CompositeStructures::InvocationActions	All
	CompositeStructures::Ports	All
	CompositeStructures::StructuredActivities	All
Use Cases	UseCases	All

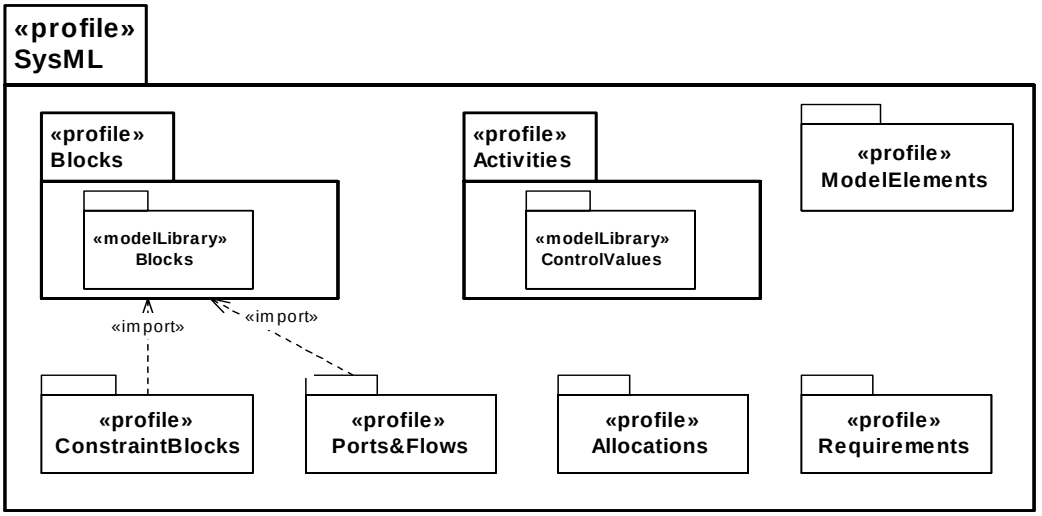


Figure 4.3 - SysML Package Structure

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 4.3 contains a set of packages that correspond to concept areas in SysML that have been extended. The reusable portion of UML that has not been extended is included by reference to the merged package (UML4SysML), and includes Interactions, State Machines, Use Cases, and Profiles.

The SysML packages extend UML as follows:

- SysML::Model Elements refactors and extends the UML kernel portion of UML classes
- SysML::Blocks reuses structured classes from composite structures
- SysML::ConstraintBlocks extends Blocks to support the parametric modeling
- SysML::Ports and Flows extends UML::Ports, UML::InformationFlows and SysML::Blocks
- SysML::Activities extends UML activities
- SysML::Allocations extends UML dependencies
- SysML::Requirements extends UML classes and dependencies

4.3 Extension Mechanisms

This specification 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 2.1 constructs with new properties and constraints. SysML diagram extensions define new diagram notations that supplement diagram notations reused from UML 2.1. SysML model libraries describe specialized model elements that are available for reuse. Additional non-normative extensions are included in Annex C: Non-normative Extensions.

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

4.4 SysML Diagrams

The SysML diagram taxonomy is shown in Figure 4.4. The concrete syntax (notation) for the diagrams along with the corresponding specification of the UML extensions is described in Parts II - IV of this specification. The Diagram Annex A describes generalized features of diagrams, such as their frames and headings.

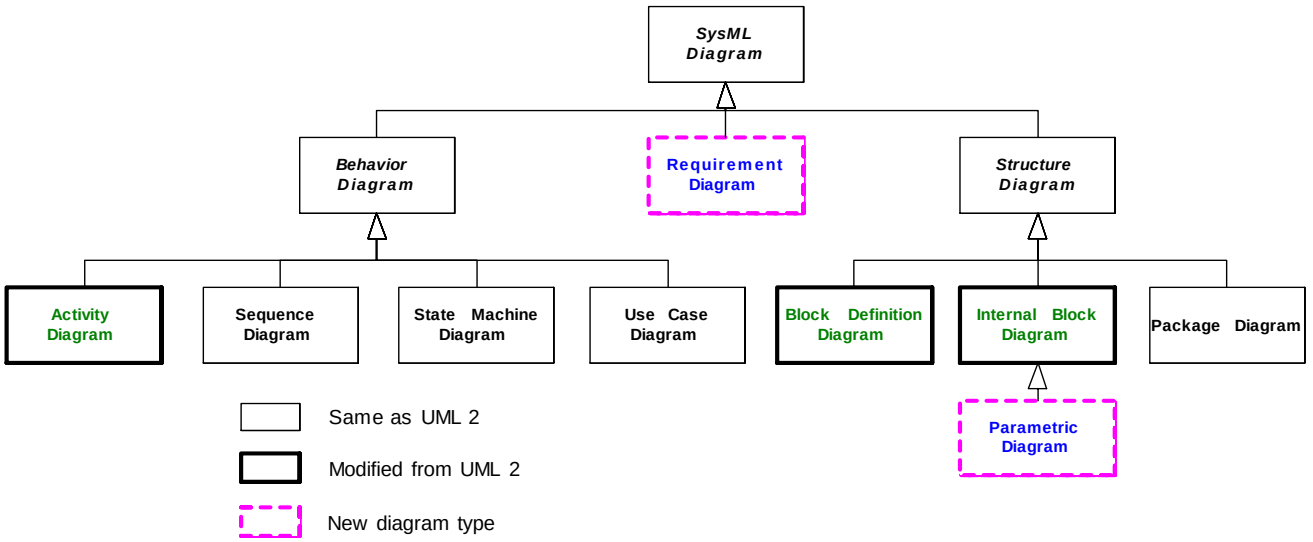


Figure 4.4 - SysML Diagram Taxonomy

5 Compliance

Compliance with SysML requires that the subset of UML required for SysML is implemented, and the extensions to the UML subset required for SysML are implemented. In order to fully comply with SysML, a tool must implement both the concrete syntax (notation) and abstract syntax (metamodel) for the required UML subset and the SysML extensions. The following sections elaborate the definition of compliance for SysML.

5.1 Compliance with UML Subset (UML4SysML)

The subset of UML required for SysML is specified by the UML4SysML package as described in Chapter 4, “Language Architecture.” UML has three compliance levels (L1, L2, L3) that SysML applies to the subset in the UML4SysML package. The levels are:

- *Level 1 (L1)*. This level provides the core UML concepts from the UML kernel and adds language units for use cases, interactions, structures, actions, and activities.
- *Level 2 (L2)*. This level extends the language units already provided in Level 1and adds language units for state machine modeling, and profiles.
- *Level 3 (L3)*. This level represents the complete UML. It extends the language units provided by Level 2 and adds new language units for modeling information flows, and model packaging.

These compliance levels are constructed in the same fashion as for UML and readers are referred to the UML 2.1 Superstructure document for further information.

5.1.1 Compliance Level Contents

The following tables identify the metamodel packages whose contents contribute to the individual compliance levels. The metaclasses in each level are included in addition to those that are defined in lower levels (Level (N) includes all the packages supported by Level (N-1)).

Table 5.1 - Metamodel packages added in Level 1

Language Unit	Metamodel Packages
Actions	Actions::BasicActions
Activities	Activities::FundamentalActivities
	Activities::BasicActivities
Classes	Classes::Kernel
	Classes::Dependencies
	Classes::Interfaces
General Behavior	CommonBehaviors::BasicBehaviors
Structures	CompositeStructure::InternalStructures
Interactions	Interactions::BasicInteractions
UseCases	UseCases

Table 5.2 - Metamodel packages added in Level 2

Language Unit	Metamodel Packages
Actions	Actions::StructuredActions
	Actions::IntermediateActions
Activities	Activities::IntermediateActivities
	Activities::StructuredActivities
General Behavior	CommonBehaviors::Communications
	CommonBehaviors::SimpleTime
Interactions	Interactions::Fragments
Profiles	AuxilliaryConstructs::Profiles
Structures	CompositeStructures::InvocationActions
	CompositeStructures::Ports
	CompositeStructures::StructuredClasses
State Machines	StateMachines::BehaviorStateMachines

Table 5.3 Metamodel packages added in Level 3

Language Unit	Metamodel Packages
Actions	Actions::CompleteActions
Activities	Activities::CompleteActivities
Classes	Classes::PowerTypes
	Classes::AssociationClasses
Information Flows	AuxilliaryConstructs::InformationFlows
Models	AuxilliaryConstructs::Models
Structures	CompositeStructures::StructuredActivities

5.2 Compliance with SysML Extensions

In addition to UML, further units of compliance for SysML are the sub packages of the SysML profile. The list of these packages is provided in Chapter 4, “Language Architecture”.

For an implementation of SysML to comply with a particular SysML package, it must also comply with any packages on which the particular package depends. For SysML, this includes not only other SysML packages, but the UML4SysML compliance level that introduces all the metaclasses extended by stereotypes in that package. The following table

identifies the level of UML4SysML on which each SysML package depends. Note that some of the SysML packages such as Model Elements, have two compliance points. This occurs when different stereotypes within the package extend metaclasses that are at more than one UML compliance level.

Table 5.4 - SysML package dependence on UML4SysML compliance levels

SysML Package	UML4SysML Compliance Level
Activities (without Probability)	Level 2
Activities (with Probablity)	Level 3
Allocations	Level 2
Blocks	Level 2
Constraint Blocks	Level 2
Model Elements (without View)	Level 1
Model Elements (with View)	Level 3
Ports and Flows (without ItemFlow)	Level 2
Ports and Flows (with ItemFlow)	Level 3
Requirements	Level 1

5.3 Meaning of Compliance

An implementation of SysML must comply with both the subset of UML4SysML and the SysML extensions as described above. The meaning of compliance in SysML is based on the UML definition of compliance, excluding diagram interchange (note that diagram interchange is different from model interchange which is included in SysML - refer to XMI below).

Compliance can be defined in terms of the following:

- *Abstract syntax compliance*. For a given compliance level, this entails:
  - compliance with the metaclasses, stereotypes and model libraries, their structural relationships, and any constraints defined as part of the abstract syntax for that compliance level and,
  - the ability to output models and to read in models based on the XMI schema corresponding to that compliance level.
- *Concrete syntax compliance*. For a given compliance level, this entails:
  - Compliance to the notation defined in the “Diagram Elements” tables and diagrams extension sections in each chapter of this specification for those metamodel elements that are defined as part of the merged metamodel or profile subset for that compliance level and, by implication, the diagram types in which those elements may appear.

Compliance for a given level can be expressed as:

- abstract syntax compliance
- concrete syntax compliance
- abstract syntax with concrete syntax compliance

The fullest compliance response is “YES,” which indicates full realization of *all language units/stereotypes* that are defined for that compliance level. This also implies full realization of all language units/stereotypes in all the levels below that level. “Full realization” for a language unit at a given level means supporting the *complete set of modeling concepts* defined for that language unit *at that level*. A compliance response of “PARTIAL” indicates partial realization and requires a feature support statement detailing which concepts are supported. These statements should reference either the language unit and metaclass, or profile package and stereotype for abstract syntax, or a diagram element for concrete syntax (the diagram elements in SysML are given unique names to allow unambiguous references). Finally, a response of “NO” indicates that none of the language units/stereotypes in this compliance point is realized.

Thus, it is not meaningful to claim compliance to, say, Level 2 without also being compliant with Level 1. A tool that is compliant at a given level must be able to import models from tools that are compliant to lower levels without loss of information.

Table 5.5 - Example compliance statement

Compliance Summary		
Compliance level	**Abstract Syntax**	**Concrete Syntax**
UML4SysML Level 1	YES	YES
UML4SysML Level 2	PARTIAL	YES
UML4SysML Level 3	NO	NO
Activities (without Probability)	YES	YES
Activities (with Probability)	NO	NO
Allocations	PARTIAL	PARTIAL
Blocks	YES	YES
Constraint Blocks	YES	YES
Model Elements (without Views)	YES	YES
Model Elements (with Views)	NO	NO
Ports and Flows (without Item Flow)	YES	YES
Ports and Flows (with Item Flow)	NO	NO
Requirements	YES	YES

In the case of “PARTIAL” support for a compliance point, in addition to a formal statement of compliance, implementors and profile designers must also provide *feature support statements*. These statements clarify which language features are not satisfied in terms of language units and/or individual packages, as well as for less precisely defined dimensions such as semantic variation points.

An example feature support statement is shown in Table 5.6 for an implementation whose compliance statement is given in Table 5.5.

Table 5.6 - Example feature support statement

Feature Support Statement				
Compliance Level/	**Detail**	**Abstract Syntax**	**Concrete Syntax**	**Semantics**
UML4SysML::Level 2:	StateMachines::BehaviorStateMachines	Note (1)	Note(1)	Note (2)

Table 5.6 - Example feature support statement

Feature Support Statement				
Compliance Level/	**Detail**	**Abstract Syntax**	**Concrete Syntax**	**Semantics**
SysML::Blocks	Block	YES	Note (3)	

Note (1): States and state machines are limited to a single region
Shallow history pseudostates not supported

Note (2): FIFO queueing in event pool

Note (3): Don’t show Blocks::StructuredCompartment notation

6 Language Formalism

The SysML specification is defined by using UML 2.1 specification techniques. These techniques are used to achieve the following goals in the specification.

- Correctness
- Precision
- Conciseness
- Consistency
- Understandability

The specification technique used in this specification describes SysML as a UML extension that is defined using stereotypes and model libraries.

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 Chapter Specification Structure

The chapters in Parts II - IV are organized according to the SysML packages as described in the language architecture and selected reusable portions of UML 2.1 packages. This section provides information about how each chapter is organized.

6.2.1 Overview

This section 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 section 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 permutations in which they can be used. The reader should refer to the usage examples in the chapters and the sample problem annex (Annex B) for typical usages of the concrete syntax. General diagram information on the use of diagram frames and headings can be found in the Diagram Annex A.

A Backus-Naur-Form (BNF) included in Annex G is used to more rigorously define the notation in selected chapters (Model Elements, Blocks, Constraint Blocks). This formalism may be considered for application to other chapters in a future revision.

6.2.3 UML Extensions

This section specifies the SysML extensions to UML in terms of the diagram extensions and stereotype and model library extensions. The diagram extensions are included when the concrete syntax uses notation other than the standard stereotype notation as defined in the Profiles and Model Libraries chapter. The semantic extensions include both the stereotype and model library extensions. The stereotype extension includes 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. The model libraries are defined as subclasses of existing metaclasses.

6.2.4 Usage Examples

This section 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 B.

6.3 Conventions and Typography

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

- While referring to stereotypes, metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are always used.
- No visibilities are presented in the diagrams, since all elements are public.
- If a section is not applicable, it is not included.
- Stereotype, metaclass and metassociation 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’).

Part II - Structural Constructs

This Part defines the static and structural constructs used in SysML structure diagrams, including the package diagram, block definition diagram, internal block diagram, and parametric diagram. The structural constructs are defined in the model elements, blocks, ports and flows, and constraint blocks chapters. The model elements chapter refactors the kernel package from UML 2.1 and includes some extensions to provide some foundation capabilities for model management. The blocks chapter reuses and extends structured classes from UML 2.1 composite structures to provide the fundamental capability for describing system decomposition and interconnection, and different types of system properties including value properties, units and distributions. The ports and flows chapter provide the semantics for defining how blocks and parts interact through ports and how items flow across connectors. The constraint blocks chapter defines how blocks are extended to be used on parametric diagrams that model a network of constraints on system properties to support engineering analysis, such as performance, reliability, and mass properties analysis.

7 Model Elements

7.1 Overview

The ModelElements package of SysML reuses several general-purpose constructs that may be used in several diagrams. These include package, model, various types of dependencies (i.e., import, access, refined, realization), constraints, and comments. The package diagram defined in this chapter, is used to organize the model by partitioning model elements into packagable 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, requirements 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 Chapter 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. The comment is not included in the model repository. 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.

SysML has extended the concept of view and viewpoint from UML to be consistent with the IEEE 1471 standard. In particular, a viewpoint is a specification of rules for constructing a view to address a set of stakeholder concerns, and 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. Typical examples may include an operational, manufacturing, or security view/viewpoint.

7.2 Diagram Elements

7.2.1 Graphical Nodes and Paths

Table 7.1 - Graphical nodes defined by ModelElements package.

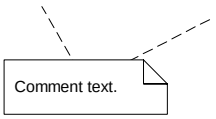
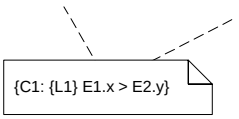
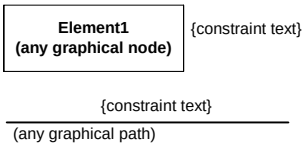
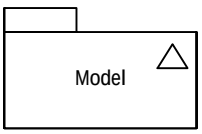
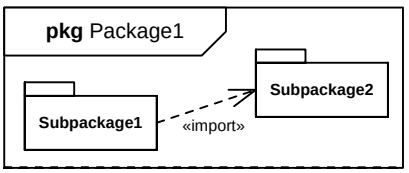
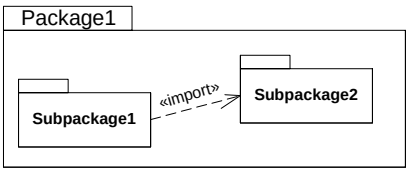
Element Name	Concrete Syntax Example	Abstract Syntax Reference
Comment		**UML4SysML::Comment**
ConstraintNote		**UML4SysML::Constraint**
ConstraintTextualNote		**UML4SysML::Constraint**
Model		**UML4SysML::Model**
PackageDiagram		**UML4SysML::Package**
PackageWith NameInTab		**UML4SysML::Package**

Table 7.1 - Graphical nodes defined by ModelElements package.

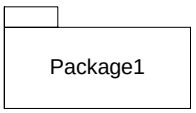
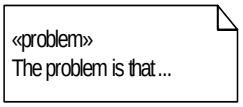
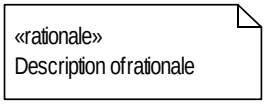
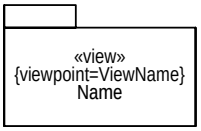
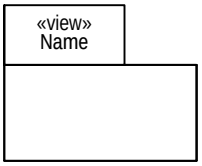
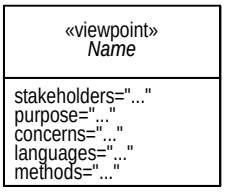
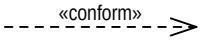
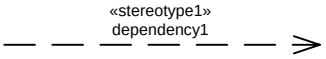
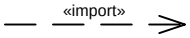
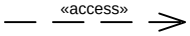
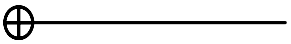
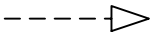
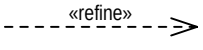
Element Name	Concrete Syntax Example	Abstract Syntax Reference
PackageWith NameInside		**UML4SysML::Package**
Problem		**SysML::ModelElements::Problem**
Rationale		**SysML::ModelElements::Rationale**
ViewWith NameInside		**SysML::ModelElements::View**
ViewWith NameInTab		**SysML::ModelElements::View**
Viewpoint		**SysML::ModelElements::Viewpoint**

Table 7.2 - Graphical paths defined by ModelElements package.

Element Name	Concrete Syntax Example	Abstract Syntax Reference
Conform		**SysML::ModelElements::Conform**
Dependency		**UML4SysML::Dependency**
PublicPackageImport		**UML4SysML::ElementImport with visibility = public**
PrivatePackageImport		**UML4SysML::ElementImport with visibility = private**
PackageContainment		**UML4SysML::Package:: ownedElement**
Realization		**UML4SysML::Realization**
Refine		**UML4SysML::Refine**

7.3 UML Extensions

7.3.1 Diagram Extensions

7.3.1.1 Stereotype Keywords or Icons Inside a Comment Note Box

Description

A comment note box may contain stereotype keywords or icons even though Comment is not a named element. UML specifies placement of a stereotype keyword relative to the name of the element. SysML makes explicit that they may appear inside a comment box as well. The stereotype keywords, if present, should appear prior to the comment text. The stereotype properties, if present, should appear after the comment text. The typical placement of stereotype icons is in the upper-right-hand corner of the containing graphical node.

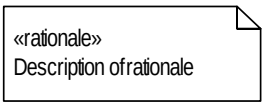


Figure 7.1 - Notation for the Rationale stereotype of Comment

7.3.1.2 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. 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.

Dependency subtypes that are imported from UML are defined in the respective chapters where they are used.

7.3.2 Stereotypes

Package ModelElements

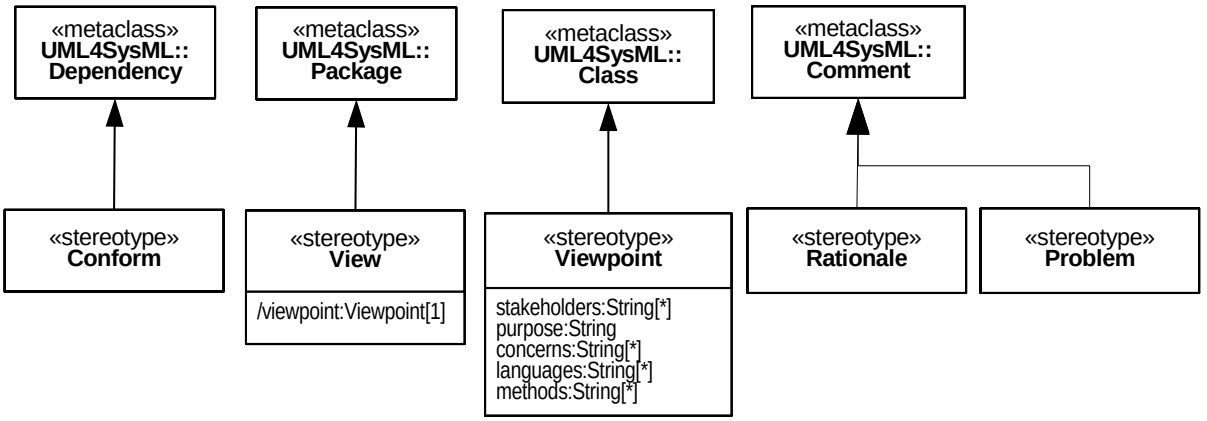


Figure 7.2 - Stereotypes defined in package ModelElements.

7.3.2.1 Conform

Description

A Conform relationship is a dependency between a view and a viewpoint. The view conforms to the specified rules and conventions detailed in the viewpoint. Conform is a specialization of the UML dependency, and as with other dependencies the arrow direction points from the (client/source) to the (supplier/target).

Constraints

[1] The supplier/target must be an element stereotyped by «viewpoint».

[2] The client/source must be an element that is stereotyped by «view».

7.3.2.2 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.

7.3.2.3 Rationale

Description

A Rational documents the justification for decisions such as 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.

7.3.2.4 View

Description

A view is a representation of a whole system from the perspective of a single viewpoint.

Attributes

- /viewpoint:Viewpoint[1]

  The viewpoint for this View, derived from the supplier of the <<conform>> dependency whose client is this View.

Constraints

[1] A view can only own element import, package import, comment, and constraint elements.

[2] The view is constructed in accordance with the methods and languages that are specified as part of the viewpoint. SysML does not define the specific methods. The precise semantic of this constraint is a semantic variation point.

7.3.2.5 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. The languages and methods for specifying a view may reference methods and languages in another viewpoint. 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

- stakeholders:String[*] Set of stakeholders.
- concerns:String[*] The interest of the stakeholders.
- purpose:String The purpose addresses the stakeholder concerns.
- languages:String[*] The languages used to construct the viewpoint
- methods:String[*] The methods used to construct the views for this viewpoint

Constraints

[1] A viewpoint cannot be the classifier of an instance specification.

[2] The property *ownedOperations* must be empty.

[3] The property *ownedAttributes* must be empty.

[4] The property *isAbstract* must be set to true.

7.4 Usage Examples

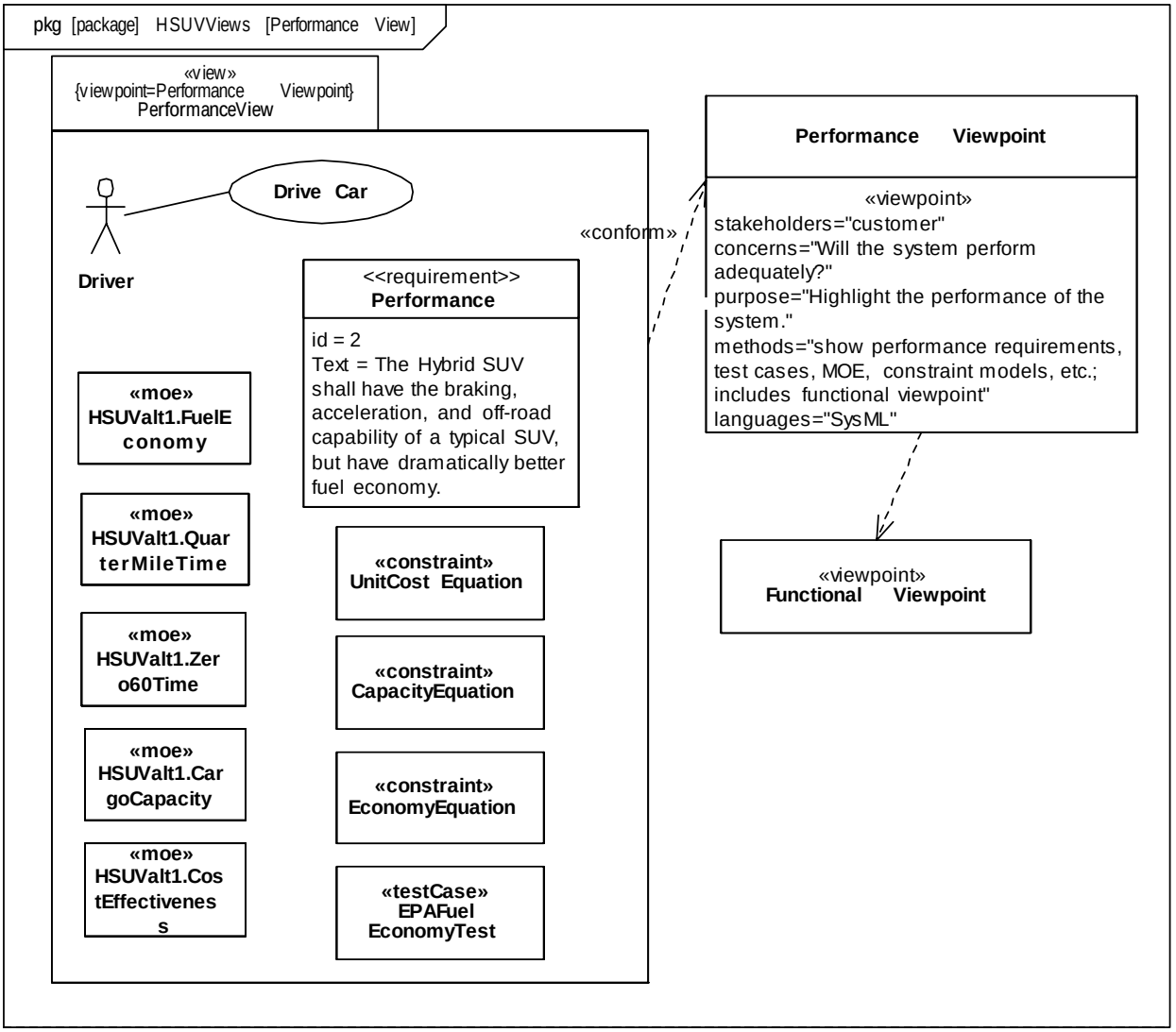


Figure 7.3 - View/Viewpoint example

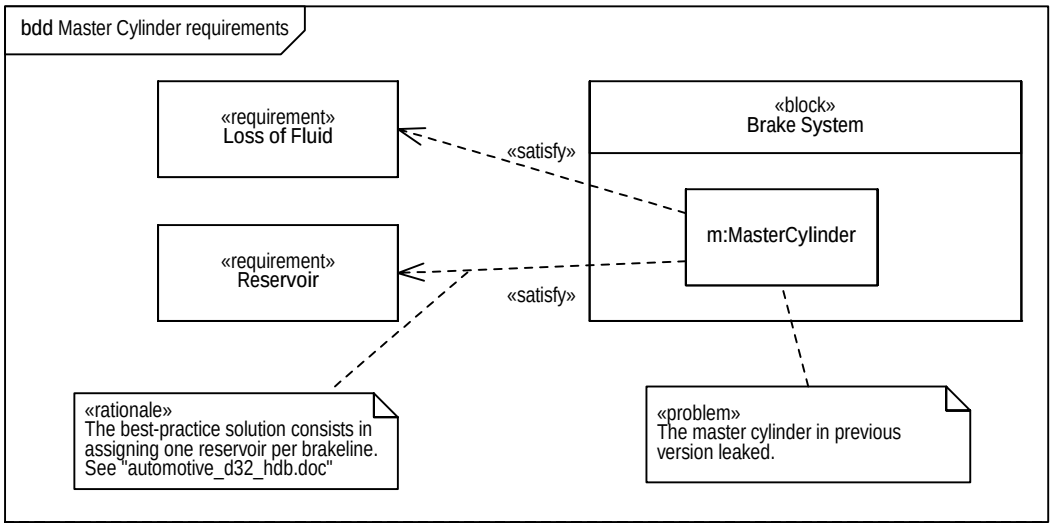


Figure 7.4 - Rationale and Problem example

8 Blocks

8.1 Overview

Blocks are modular units of a system description, which define a collection of features to describe a system or other elements 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 ways 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 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 a block 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 Chapter 9, “Ports and Flows.” Constraint Properties are a special class of property used to constrain other properties of blocks, and are described in Chapter 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 chapter 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. Chapter 9, “Ports and Flows” specifies the allowable types of interactions between blocks, and the Behavioral Constructs in Section III including activities, interactions, and state machines can be applied to blocks to specify its behavior. Chapter 15, “Allocations” in Part IV 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 and multi-level nesting of connector ends. SysML blocks include several notational extensions as specified in this chapter.

8.2 Diagram Elements

Tables in the following sections provide a high-level summary of graphical elements available in SysML diagrams. A more complete definition of SysML diagram elements, including the different forms and combinations in which they may appear, is provided in Annex G.

8.2.1 Block Definition Diagram

8.2.1.1 Graphical Nodes and Paths

Table 8.1 - Graphical nodes defined in Block Definition diagrams

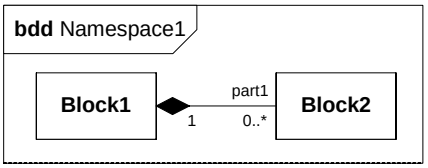
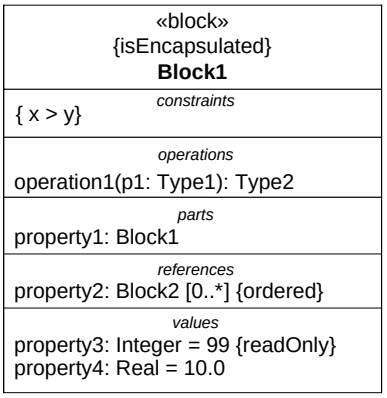
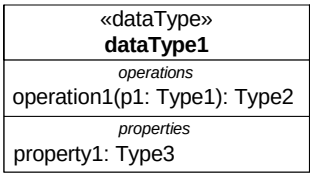
Element Name	Concrete Syntax Example	Abstract syntax Reference
BlockDefinition Diagram		**SysML::Blocks::Block** **UML4SysML::Package**
Block		**SysML::Blocks::Block**
Actor		**UML4SysML::Actor**
DataType		**UML4SysML::DataType**

Table 8.1 - Graphical nodes defined in Block Definition diagrams

Element Name	Concrete Syntax Example	Abstract syntax Reference
ValueType	«valueType**ValueType1***operations*operation1(p1: Type1): Type2*properties*property1: Type3«valueType»unit = UnitName	**SysML::Blocks::ValueType**
Enumeration	«enumeration»**Enumeration1**literalName1literalName2	**UML4SysML::Enumeration**
AbstractDefinition	***Name1*****{abstract}*****Name1*****Name1*****{abstract}**	**UML4SysML::Classifier with isAbstract equal true**
StereotypeProperty Compartment	«stereotype1»**Block1**«stereotype1»property1 = value	**UML4SysML::Stereotype**
Namespace Compartment	**Block1***namespace***Block2**part1 1 0..***Block3**	**SysML::Blocks::Block**
Structure Compartment	**Block1***structure***p1: Type1**c1: 1e1**p2: Type2**	**SysML::Blocks::Block**

Table 8.2 - Graphical paths defined by in Block Definition diagrams.

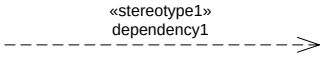
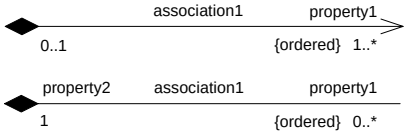
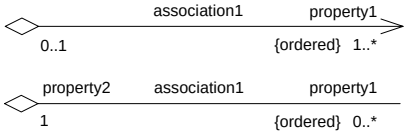
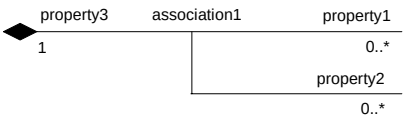
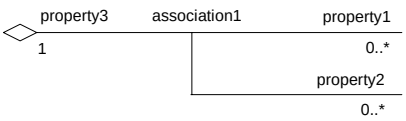
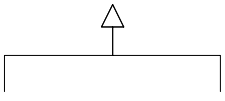
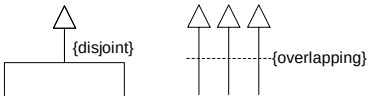
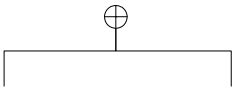
Element Name	Concrete Syntax Example	Abstract syntax Reference
Dependency		**UML4SysML::Dependency**
ReferenceAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = none**
PartAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = composite**
SharedAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = shared**
MultibranchPart Association		**UML4SysML::Association and UML::Kernel::Property with aggregationKind = composite**
MultibranchShared Association		**UML4SysML::Association and UML::Kernel::Property with aggregationKind = shared**
Generalization		**UML4SysML::Generalization**
Multibranch Generalization		**UML4SysML:Generalization**

Table 8.2 - Graphical paths defined by in Block Definition diagrams.

Element Name	Concrete Syntax Example	Abstract syntax Reference
GeneralizationSet		**UML4SysML:: GeneralizationSet**
BlockNamespace Containment		**UML4SysML::Class:: nestedClassifier**

8.2.2 Internal Block Diagram

8.2.2.1 Graphical Nodes and Paths

Table 8.3 - Graphical nodes defined in Internal Block diagrams

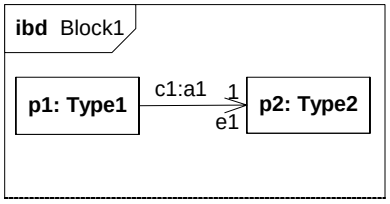
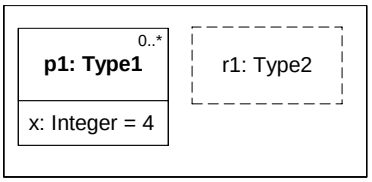
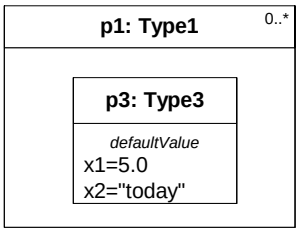
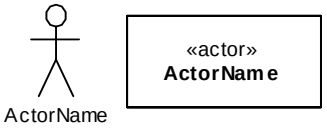
Element Name	Concrete Syntax Example	Abstract Syntax Reference
InternalBlockDiagram		**SysML::Blocks::Block**
BlockProperty	 	**SysML::Blocks::BlockProperty**
ActorPart		**SysML::Blocks::PartProperty typed by UML4SysML::Actor**

Table 8.3 - Graphical nodes defined in Internal Block diagrams

Element Name	Concrete Syntax Example	Abstract Syntax Reference
PropertySpecificType	**p1: [Type1]** *values* «uniform»{mean=2,stdDeviation=0.1} x: Integer **p2** *values* y: Integer = 5	**SysML::Blocks::BlockProperty**

Table 8.4 - Graphical paths defined in Internal Block diagrams

Element Name	*Concrete Syntax Example*	*Abstract Syntax Reference*
Dependency	«stereotype1» dependency1	**UML4SysML::Dependency**
BindingConnector	1 — 0..*	**UML4SysML::Connector**
Bidirectional Connector	p1 — c1: association1 — p2 0..1 — 0..*	**UML4SysML::Connector**
Unidirectional Connector	c1: association1 — p1 0..1 — 0..*	**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.

8.3.1.2 Block and ValueType 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 section.

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.

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 subsections of this section for a description of these compartments.

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 Chapter 10, “Constraint Blocks.**”** 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.

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 which 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.

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 which 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.

Unit and Dimension declarations

The declarations of value types have been extended to support the declaration of a unit of measure or a dimension. These declarations must refer by name to an instance of a Unit or Dimension stereotype defined separately. A sample set of predefined dimensions and units is given in Annex C, section C.4.

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.

8.3.1.3 Internal Block Diagram

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

Property types

Three general categories of properties are recognized in SysML: parts, references and value properties (see 8.3.2.2 Block Property below). 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.

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) must 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. This component is optional for many SysML diagram types, but not for an internal block diagram.) All the properties and connectors which appear inside the internal block diagram belong to the block that is named in the diagram heading name.

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 must match those of the block or value type that types the property. For a property-specific type, these compartments may be used to specify redefined or additional features of the locally defined type. An unlabeled compartment on an internal property box is by default a structure compartment.

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 must always follow an initial compartment which 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.

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 which 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.

This notation is purely a notational shorthand for a property which 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 below is equivalent to the innermost nested box shown at the right:

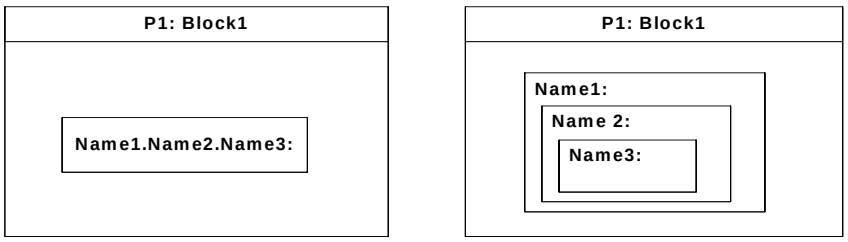


Figure 8.1 - Nested property reference

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 which 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.

Property-specific type

Enclosing the type name of an internal property in square brackets specifies that the type is a local specialization of the referenced type, which may be overridden to specify additional values or other customizations that are unique to the property. Redefined or added features of the newly defined type, such as a value or distribution specifications, may be shown in the compartments for the property. If the property name appears on its own, with no colon or type name, then the property-specific type is entirely provided by its local declarations.

Default value compartment

A compartment with a label of "defaultValue" may be used to show the default value for a property as an alternative to an "="suffix string on its declaration within its containing block. It may be used for a property whose type has substructure and a default value with many subvalues. A default value compartment on a property may be used instead of a property-

specific type when all that is required are property-specific values. If a default value is specified for a property nested any level deeper than the top level of an internal block diagram frame, then its containing property must still have a property-specific type, so that the default value specification can be included within that type.

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.

8.3.1.4 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. N-ary associations, shown in UML by a large open diamond with multiple branches, can be modeled by an intermediate block with no loss in expressive power. Qualified associations, shown in SysML by an open box at the end of an association path with a property name inside, are a specialized feature of UML that specifies how a property value can represent an identifier of an associated target. This capability, while useful for data modeling, does not seem essential to accomplish any of the SysML requirements for support of systems engineering. 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 the 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 an owned end of an association. SysML still supports use of an arrowhead on one end of a unidirectional association. Generalization relationships between associations are not supported. Their semantics in UML is not explicitly stated. SysML restricts the semantics of associations to providing supplementary specifications on properties at either or both ends of the association, which must be owned by the associated classifier.

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

8.3.1.5 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 chapter. Other SysML chapters add some of these notations into the supported contents of an internal block diagram.

8.3.2 Stereotypes

Package Blocks

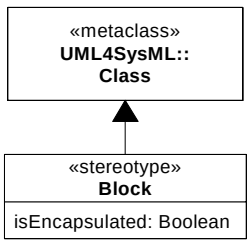


Figure 8.2 - Stereotypes defined in SysML Blocks package

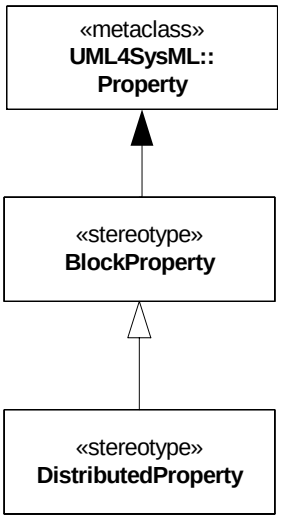


Figure 8.3 - Abstract syntax extensions for SysML properties

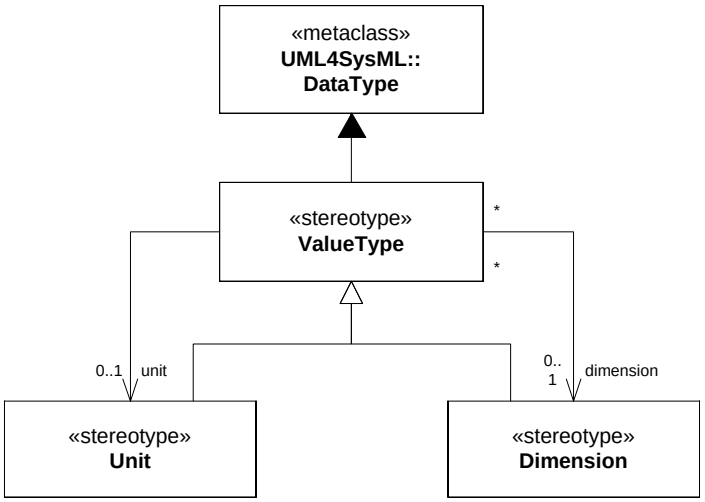


Figure 8.4 - Abstract syntax extensions for SysML value types

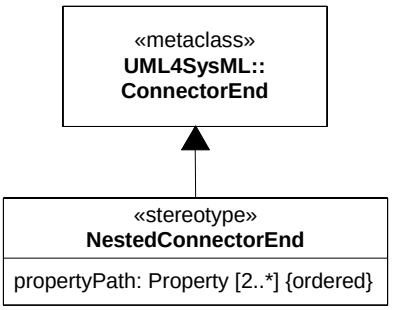


Figure 8.5 - Abstract syntax extensions for SysML connector ends

8.3.2.1 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, that 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. 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. The type of a connector or its connected ends may specify the semantic interpretation of a specific connector. A Binding Connector is a connector that is not typed by an association. If the two ends of a binding connector have the same type, the connector specifies that the properties at the end of the connector must have the same values, recursively through any nested properties within the connected properties.

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.

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, then connections can be established to elements of its internal structure via deep-nested connector ends.

Constraints

[1] For an association in which both ends are typed by blocks, the number of ends must be exactly two.

[2] The number of ends of a connector must 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.)

[3] In the UML metamodel on which SysML is built, any instance of the Property metaclass that is typed by a block (a Class with the «block» stereotype applied) and which is owned by an Association may not have a name and may not be defined as a navigable owned end of the association. (While the Property has a “name” property as defined by its NamedElement superclass, the value of the “name” property, which is optional, must be missing.)

[4] In the UML metamodel on which SysML is built, a Property that is typed by a block must be defined as an end of an association. (An inverse end of this association, whether owned by another block or the association itself, must always be present so there is always a metamodel element to record the inverse multiplicity of the reference.)

[5] The following constraint under Section 9.3.6, “Connector” in the UML 2.0 Superstructure Specification (OMG document formal/05-07-04) is removed by SysML: “[3] 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.”

8.3.2.2 BlockProperty

Description

The BlockProperty stereotype enforces additional constraints specific to SysML.

A property of a block may refer to another element of a system that is required to exist for the system to exist. Such properties are called part properties. Part properties must always be defined as an end of an association whose other (inverse) end can used to indicate whether the part can exist independently of the referencing system. A multiplicity of 0..1 on the inverse end indicates that instances of the block that types part property may exist without being referenced by the part property of a system that may own them. While referenced by the part property, however, these instances cannot cease to exist unless the owning system also ceases to exist. A part property also imposes the restriction that an instance on the part end can be owned only by a single block at any given time.

Parts may be used to show all the components from which a larger system is built. Consistent with UML, however, SysML currently does not provide any means to indicate whether all the parts which make up a larger system are either shown on a particular diagram or contained within a model. Various forms of diagram or model annotation, such as a Diagram Description note as shown in Annex A, may be used to communicate completeness of a diagram or model to a user.

SysML also supports the definition of properties which are typed by a block and that hold their instances under a form of shared ownership, as indicated by a “white diamond” form of shared association. SysML defines no semantics or constraints for properties held by a shared association. Specific interpretations of shared ownership properties may be established by a particular model and communicated to a user under local conventions.

A property of a block may refer to another element of a system or system description, that is not owned by the block; these are called reference properties. Any given element may be referenced by multiple reference properties within a system description. Reference properties can be used to represent elements of a system that require the identification of other elements to define a compound situation or to express causal dependencies. For example, the mating of two system elements may require the identification of two mated elements to describe the resulting connection between them, or a sensor may require the identification of a system whose properties it measures. Reference properties can also be used to share common information across multiple elements of a model, such as descriptions of material properties for physical parts.

Alternatively a property of a block may hold a value; this is called a value property and will be typed either by a UML Data Type, or a SysML Value Type. A value may be used to express information about a system, but cannot be identified apart from the value itself. Since it lacks any identity separate from the value itself, each value held by a value property is independent of any other, unless other forms of constraints are imposed. A value property may be used to express inherent characteristics about a system such as its mass or length. A value property may also be used to hold elements of a system description that are not inherent to the system itself but are nevertheless of interest, such as a design status or measured test results collected during development of a system. The unit and dimension of a value property is obtained from its type (if a Value Type is used); to specify the unit and dimension of the property directly, a property specific value type can be created for it and unit and dimension specified for that.

Constraints

[1] If the aggregation attribute of the property is equal to “composite” or “shared” then the type of the property must be a block.

8.3.2.3 DistributedProperty

DistributedProperty is a stereotype of BlockProperty 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.

8.3.2.4 Dimension

A kind of quantity that may be stated by means of defined units. For example, the dimension of length may be measured by units of meters, kilometers, or feet.

Dimension is defined as a stereotype of ValueType, but it may not be used directly to declare the type of any value. The only valid use of a Dimension instance is to be referenced by the “dimension” property of a ValueType stereotype.

Constraints

[1] The “dimension” and “unit” attributes inherited from the ValueType stereotype must not contain any value.

8.3.2.5 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.

Attributes

- propertyPath: Property [2..*] (ordered)

  The propertyPath list of the NestedConnectorEnd stereotype must identify a path of containing properties that identify the connected property in the context of the block that owns the connector. The ordering of properties is from the outermost property of the block that owns the connector, through the properties of each intermediate block that types the preceding property, but not including the property which is directly connected.

Constraints

[1] The property at the first position in the propertyPath attribute of the NestedConnectorEnd must be owned by the block that owns the connector.

[2] The property at each successive position of the propertyPath attribute, following the first position, must be contained in the block that types the property at the immediately preceding position.

8.3.2.6 Unit

A quantity in terms of which the magnitudes of other quantities that have the same dimension 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 stereotype of ValueType, but it may not be used directly to declare the type of any value. The only valid use of a Unit instance is to be referenced by the “unit” property of a ValueType stereotype.

Constraints

[1] The “unit” attribute inherited from the ValueType stereotype must not contain any value. (The “dimension” attribute may be used to declare the dimension that the unit is declared to measure.)

8.3.2.7 ValueType

Description

A type that defines values which may be used to express information about a system, but which 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.

Values 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 or dimension associated with the value. A dimension 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 dimension may be expressed.

If these additional characteristics are not required then UML DataType may be used.

Attributes

- dimension: ValueType [0..1]

  A kind of quantity that may be stated by means of defined units, as identified by an instance of the Dimension stereotype. A value type may optionally specify a dimension without any unit. Such a value has no concrete representation, but may be used to express a value in an abstract form independent of any specific units.

- unit: ValueType [0..1]

  A quantity in terms of which the magnitudes of other quantities that have the same dimension can be stated, as identified by an instance of the Unit stereotype.

Constraints

[1] The dimension attribute must reference a ValueType to which the «dimension» stereotype has been applied.

[2] The unit attribute must reference a ValueType to which the «unit» stereotype has been applied.

[3] If a value is present for the unit attribute, the dimension attribute must be equal to the dimension property of the referenced unit.

8.3.3 Model Libraries

Package Blocks

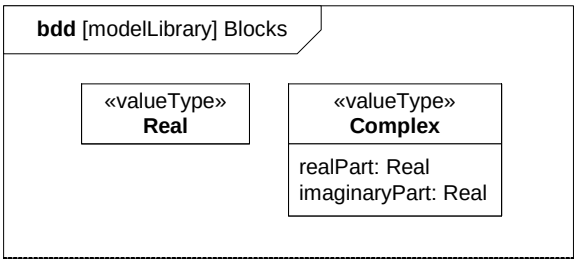


Figure 8.6 - Model Library for Blocks

8.3.3.1 Complex

Description

A value type to represent 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.

Attributes

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

8.3.3.2 Real

A value type to represent 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.

8.4 Usage Examples

8.4.1 Wheel Hub Assembly

In Figure 8.7 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 Section C.5.

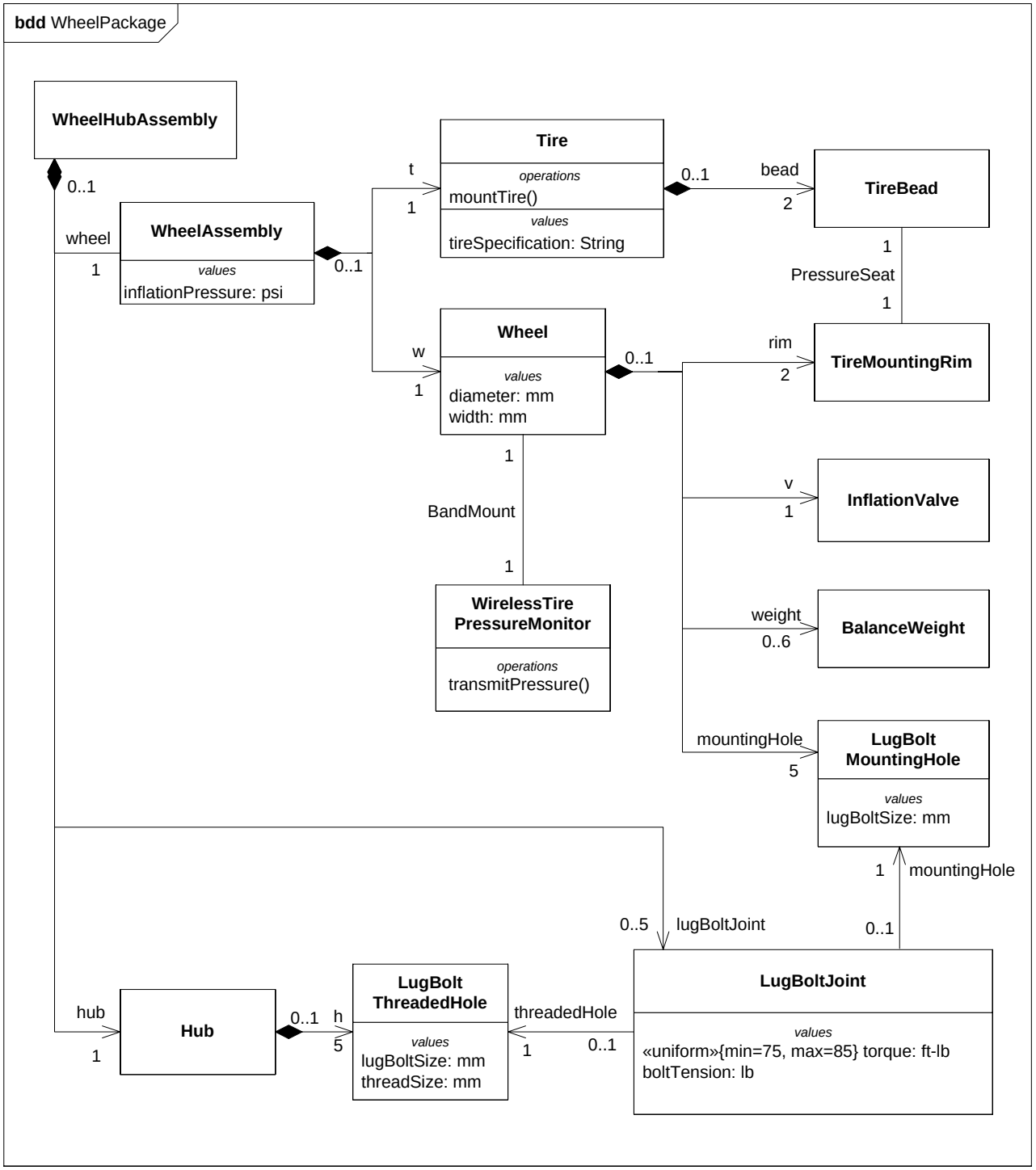


Figure 8.7 - Block diagram for the Wheel Package

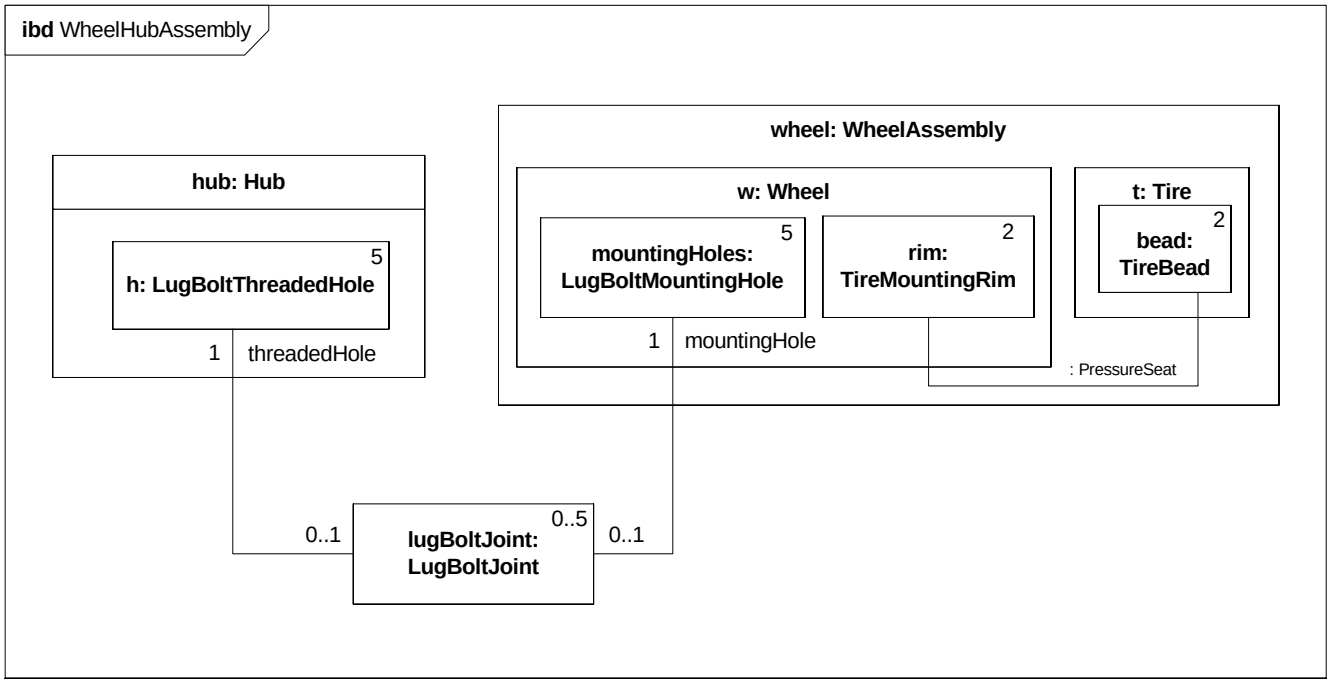


Figure 8.8 - Internal Block Diagram for WheelHubAssembly

In Figure 8.8 an internal block diagram 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.

8.4.1.1 SI Value Types

In Figure 8.9, several value types using SI units and dimensions are defined to be generally available in the SI Value Types package for typing value properties. Because a unit already identifies the type of quantity, or dimension, that the unit measures, a value type only needs to identify the unit to identify the dimension as well. The value types in this example refer to units which are assumed to be defined in an imported package, such as the SI Definitions model library defined in Section C.4.

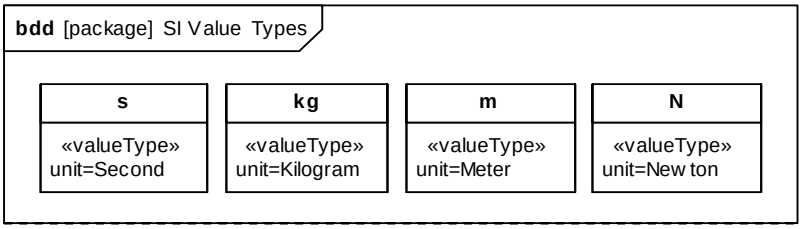


Figure 8.9 - Defining Value Types with units and dimensions

8.4.1.2 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 8-10 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 part-specific types 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 property-specific types. The following example illustrates the approach.

.

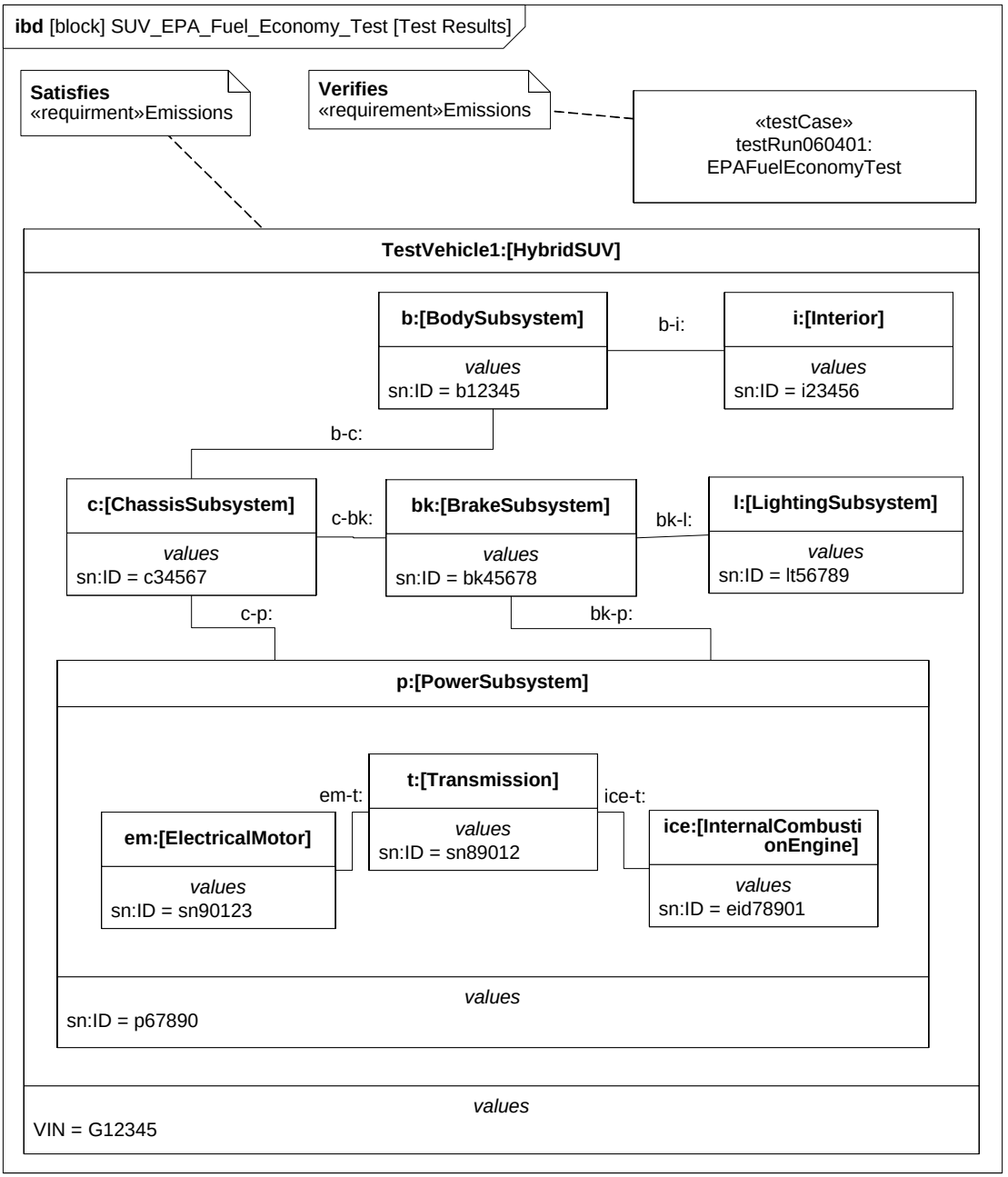


Figure 8.10 - SUV EPA Fuel Economy Test

9 Ports and Flows

9.1 Overview

This chapter specifies flow ports that enable flow of items between blocks and parts, while standard ports enable invocation of services on blocks and parts. A port is an interaction point between a block or part and its environment that is connected with other ports via connectors. The main motivation for specifying such ports on system elements is to allow the design of modular reusable blocks, with clearly defined interfaces. (Note: the block owns its ports and therefore the port is part of the blocks definition). This chapter also specifies item flows across connectors and associations.

9.1.1 Standard Ports

A Standard Port specifies the services the owning Block provides (offers) to its environment as well as the services that the owning Block expects (requires) of its environment. The specification of the services is achieved by typing the Standard Port by the provided and/or required interfaces. In general Standard Ports are used in the context of service-oriented architectures, which is typical for software component architectures. Since standard ports contain operations which specify bi-directional flow of data, standard ports are typically used in the context of peer-to-peer synchronous request/reply communications. A special case of a service is signal reception, which signifies a one way communication of signal instances, where the handling of the request is asynchronous.

For example, a Block representing an automatic transmission in a car could have a Standard Port that specifies that the Transmission Block can accept commands to switch gears. Standard Ports are another name for UML2.1 ports, in other words they are defined by the same meta-class.

9.1.2 Flow Ports

A FlowPort specifies the input and output items that may flow between a Block and its environment. FlowPorts 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 FlowPort with a specification of things that flow. This can include typing an atomic flow port with a single item that flows in our out, or typing a non-atomic 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. FlowPorts extend UML2.1 ports.

9.1.3 Item Flows

Item flows represent the things that flow between blocks and/or parts and across associations or connectors. Whereas the FlowPort specifies what “can” flow in or out of a block, the 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, a tank may include a FlowPort that can accept fluid as an input. In a particular use of the tank, “gasoline” flows across a connector into its FlowPort, and in another use of the tank, “water” flows across a connector into the its FlowPort. The item flow would specify what “does” flow on the connector in the particular usage (e.g., gas, water), and the FlowPort specifies what can flow (e.g., fluid). This enables type matching between the item flows and between flow ports 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. FlowAllocation is described in Chapter 15, “Allocations”and can be used to help ensure consistency across the different parts of the model.

9.2 Diagram Elements

9.2.1 Extensions to Block Definition Diagram.

Table 9.1 - Extensions to Block Definition Diagram

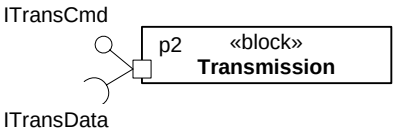
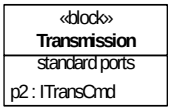
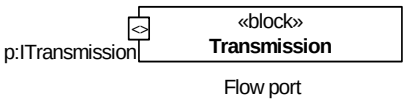
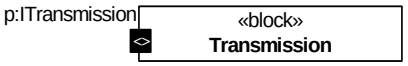
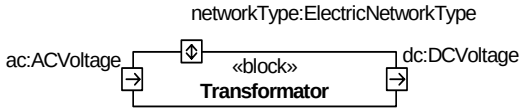
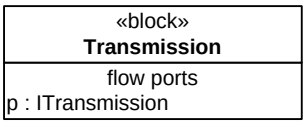
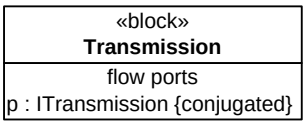
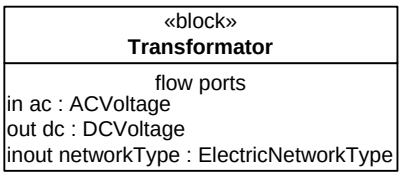
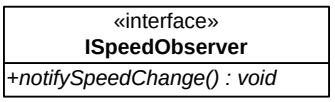
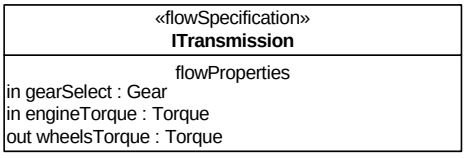
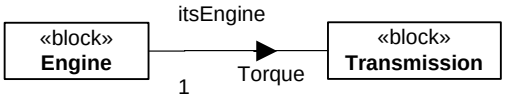
Node Name	Concrete Syntax	Abstract Syntax Reference
StandardPort		**UML4SysML::Port**
StandardPort (Compartment Notation)		**SysML::PortsAndFlows:Standard-Port**
FlowPort	 Flow port  Conjugated Flow port  Atomic Flow Ports	**SysML::PortsAndFlows::FlowPort**

Table 9.1 - Extensions to Block Definition Diagram

Node Name	Concrete Syntax	Abstract Syntax Reference
FlowPort (Compartment Notation)	 Flow port  Conjugated Flow port  Atomic Flow Ports	**SysML::PortsAndFlows::FlowPort**
Interface		**UML4SysML::Interfaces::Interface**
FlowSpecification		**SysML::PortsAndFlows::FlowSpecification**
ItemFlow		**SysML::PortsAndFlows::ItemFlow**

9.2.1.1 Extensions to Internal Block Diagram

Table 9.2 - Extension to Internal Block Diagram

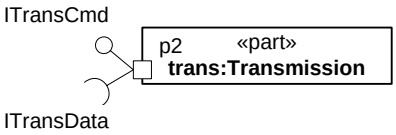
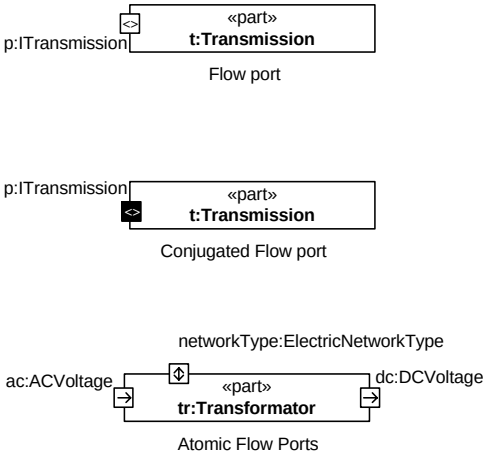
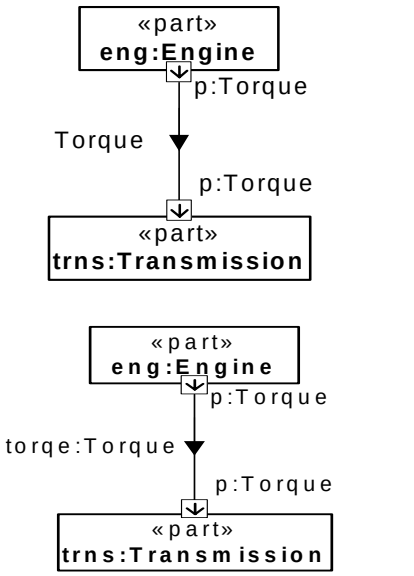
Node Name	Concrete Syntax	Abstract Syntax Reference
StandardPort		**SysML::PortsAndFlows::StandardPort**
FlowPort		**SysML::PortsAndFlows::FlowPort**

Table 9.2 - Extension to Internal Block Diagram

Node Name	Concrete Syntax	Abstract Syntax Reference
ItemFlow	 *ItemFlow with an itemProperty*	**SysML::PortsAndFlows::ItemFlow**

9.3 UML Extensions

9.3.1 Diagram Extensions

9.3.1.1 FlowPort

Flow Ports are interaction points through which input and/or output of items such as data, material or energy may flow. The notation of FlowPort 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 FlowPorts have an arrow inside them indicating the direction of the port with respect to the owning Block. A non-atomic FlowPort have 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, unless the flow port is conjugated, in which case the fill color of the square is black and the text is in white.

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}]

9.3.1.2 FlowProperty

A FlowProperty signifies a single flow element to/from a Block. A FlowProperty has the same notation of a Property only with a direction prefix (in | out | inout). Flow Properties are listed in a compartment labeled “flowProperties.”

9.3.1.3 FlowSpecification

A FlowSpecification specifies inputs and outputs as a set of flow properties. It has a “flowProperties” compartment that lists the flow properties.

9.3.1.4 ItemFlow

An Item Flow describes the flow of items across a connector or an association. The notation of ItemFlow is a black arrow-head on the connector or association. The arrow head is towards the target element. For an Item Flow with an itemProperty, the label shows the name and type of the itemProperty (in name:type format). Otherwise the Item Flow is labeled with the name of the conveyed Classifier.

9.3.2 Stereotypes

9.3.2.1 Package Ports&Flows

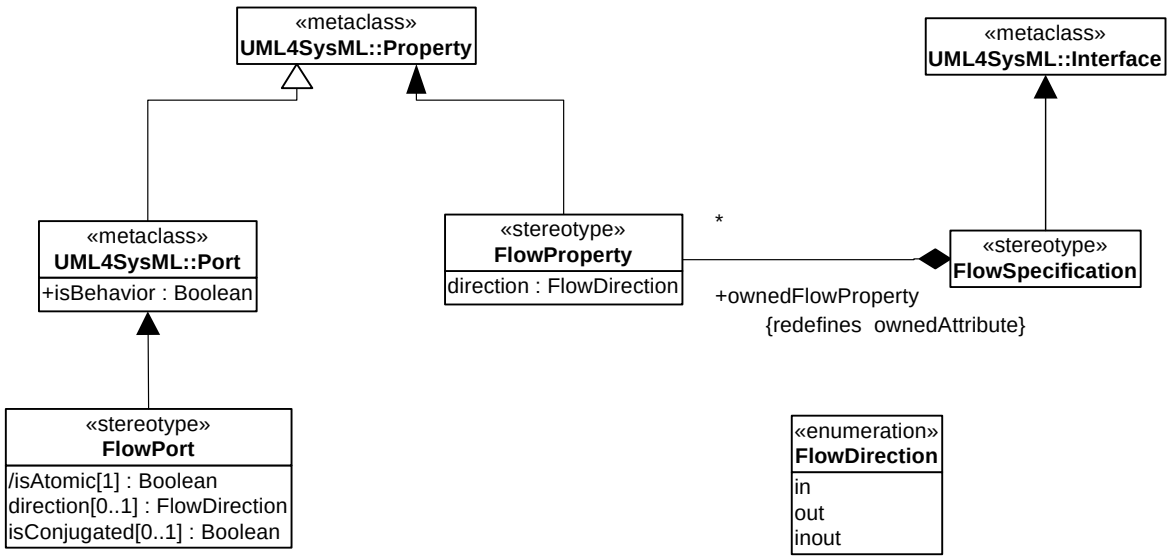


Figure 9.1 - Port Stereotypes

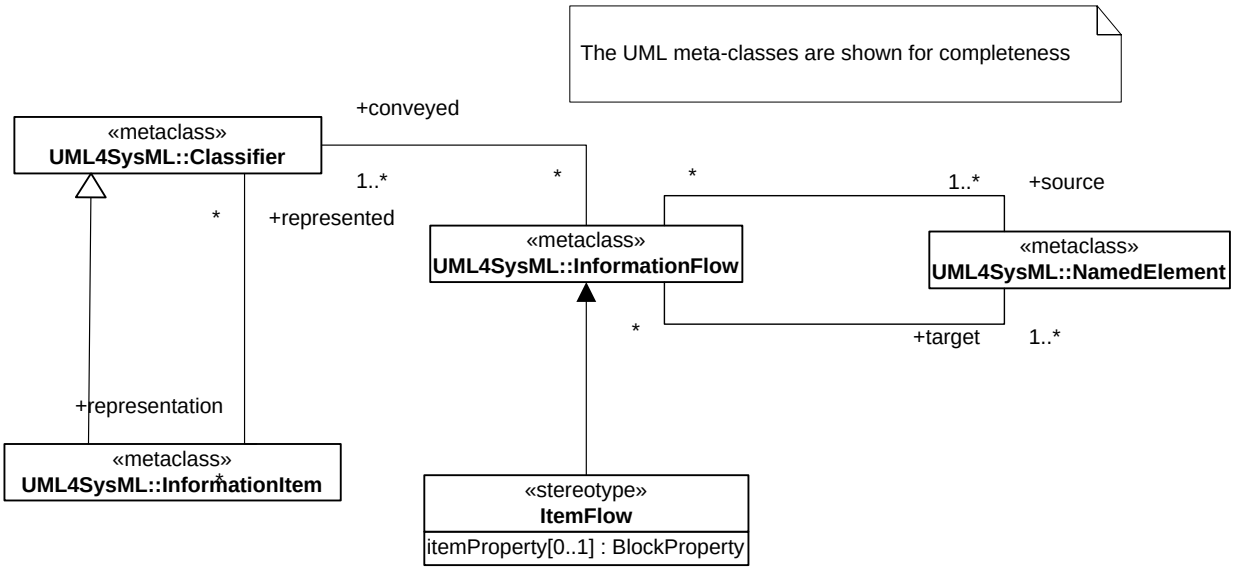


Figure 9.2 - ItemFlow Stereotype

9.3.2.2 Block

Description

Blocks may own StandardPorts and/or FlowPorts. See Chapter 8, “Blocks” for details of Block.

9.3.2.3 Standard Port

Description

StandardPorts are interaction points through which a Block provides and requires a set of services to and from its environment.

The services that the Block provides to its environment via the StandardPort are specified by a set of provided interfaces. The services that the Block requires from the environment via the StandardPort are specified by a set of required interfaces.

An interface may specify operations or signals. If the interface is provided, then external parts may call operations or send signals via the port to its owning block. If the interface is required, then the block may call operations or send signals via the port to its environment.

StandardPorts are UML 2.1 ports. As a guideline, it is recommended StandardPorts are used in the context of service based components and/or architectures, either when specifying software components or applying a service based approach to system specification.

9.3.2.4 FlowDirection

Description

FlowDirection is an enumeration type that defines literals used for specifying input and output directions. FlowDirection is used by FlowProperties to indicate if the property is an input or an output with respect to its owner.

Literal Values are

in: Indicates that the flow property is input to the owning Block.

out: Indicates that the flow property is an output of the owning Block.

nout: Indicates that the flow property is both an input and an output of the owning Block.

9.3.2.5 FlowPort

Description

Flow Ports are interaction points 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 what are the interaction points through which the exchange is made.

We distinguish between Atomic Flow Port and a Non-Atomic Flow Port: Atomic Flow Ports relay a single usage of a Block, Value-Type, Data-Type or Signal. A Non-Atomic Flow Port relays items of several types as specified by a FlowSpecification.

The distinction between Atomic and Non-Atomic Flow Ports is made according to the FlowPort's type: If a FlowPort is typed by a FlowSpecification then it is Non-Atomic, if the FlowPort is typed by a Block, ValueType, DataType or Signal, then it is Atomic.

In addition, if the Flow Port is behavioral (in case the isBehavioral:Boolean meta-attribute from UML2.0 Port is set to True) then the port relays the item that flow through it to/from its owner. The relay of items of a behavioral FlowPort is bound only to its owner's classifier behavior.[1] If the Flow Port is not behavioral then the items are relayed via internal Connectors to internal Parts (internal with respect to the port's owner).

FlowPorts and associated Flow Specifications define "what can flow" between the block and its environment. Whereas ItemFlows specify "what does flow" in a specific usage context.

Behavioral FlowPorts relay items to/from the associated connector to/from properties of the owning block or parameters of the block behavior. This means that every FlowProperty contained within a FlowPort is bound to a property owned by the block or a parameter of the block behavior. The binding of the flow properties on the ports to behavior parameters and/or block properties is a semantic variation point of this specification. One approach to specify binding is based on name and type matching, and another approach is to explicitly use binding relationships between the ports properties and behavior parameters or block properties.

In case of flow properties or Atomic FlowPort of type Signals, inbound properties/atomic FlowPort are mapped to a Reception of the signal type (or a sub type) of the flow property's type. Outbound flow properties only declare the ability of the FlowPort to relay the Signal over external connectors attached to it and are not mapped to a property of the behavioral flow-port's owning Block.

1. Other owned behaviors of the owner's classifier (a Classifier may have additional owned behaviors) are invoked internally and therefore the port cannot relay items to them.

The Item Flows specified as flowing on a connector between FlowPorts must match to the Flow Properties of the ports at each end of the connector: the source of the Item Flow should be the port which 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 FlowPort is connected to multiple external and/or internal connectors then the items are propagated (broadcasts) over all connectors that have matching properties at the other end.

Attributes

- direction : FlowDirection [0..1]
  Indicates the direction in which an Atomic FlowPort relays its items. It applies only to Atomic FlowPort (otherwise the multiplicity of this attribute is zero). If the direction is set to *in* then the items are relayed from an external connector via the FlowPort into the FlowPort's owner (or one of its Parts). If the direction is set to *out*, then the items are relayed from the FlowPort's owner, via the FlowPort, through an external connector attached to the FlowPort, and if the direction is set to *inout* then items can flow both ways. By default, the value is *inout*.

- isConjugated : Boolean [0..1]
  If set to True then all the directions of the FlowProperties specified by the FlowSpecification that types a Non-Atomic FlowPort are relayed in the opposite direction (i.e., *in* flow property is treated as an *out* flow property by the FlowPort and vice-versa). By default, the value is False. This attribute applies only to Non-Atomic FlowPorts.

- /isAtomic : Boolean (derived)
  This is a derived attribute (derived from the FlowPort's type). For Atomic FlowPort the value of this attribute is True, for Non-Atomic FlowPort the value is False.

Constraints

[1] A FlowPort must be typed by a FlowSpecification, Block, Signal, DataType, or ValueType.

[2] If the FlowPort is Atomic (i.e., typed by a Block, Signal, DataType, or ValueType), then isAtomic=True, the multiplicity of Direction is one, and the multiplicity of isConjugated is zero.

[3] If the FlowPort is Non-Atomic (i.e., typed by a FlowSpecification), then isAtomic=False, the multiplicity of Direction is zero, and the multiplicity of isConjugated is one.

[4] A Flow Port can be connected (via connectors) to one or more flow ports that have matching flow properties. The matching of flow properties is done in the following steps:

1. Type Matching: The type being sent is the same type or a sub-type 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 must by opposite, or at least one of the ends should be inout. If the connector is internal to the owner of one of the flow ports, then the direction should be the same or at least one of the ends should be inout

3. Name Matching: In case there is type and direction match to several flow properties at the other end, the property that have the same name at the other end is selected. If there is no such property then the connection is ambiguous (ill-formed)

9.3.2.6 FlowProperty

Description

A FlowProperty signifies a single flow element that can flow to/from a Block. A Flow Property's values are either received from or transmitted to an external Block. Flow Properties are defined directly on Blocks or Flow Specifications which are those specifications which type the Flow Ports.

FlowProperties enable item flows across connectors connecting parts of the corresponding block types, either directly (in case of the property is defined on the block) or via flowPorts. For Block, Data Type and Value Type properties, setting an *out* FlowProperty value of a Block usage on one end of a connector will result in assigning the same value of an *in* FlowProperty of a Block usage at the other end of the connector, provided the FlowProperties are matched. FlowProperties of type Signal imply sending and/or receiving of a Signal usages. An *out* FlowProperty of type Signal means that the owning Block may broadcast the signal via connectors and an *in* FlowProperty means that the owning Block is able to receive the Signal.

Attributes

- direction : FlowDirection
  Specifies if the property value is received from an external Block (direction=*in*), transmitted to an external Block (direction=*out*) or both (direction=*inout*).

Constraints

[1] FlowProperties are typed by a ValueType, DataType, Block or Signal.

[2] An *in* FlowProperty value cannot be modified by its owning Block.

[3] An *out* FlowProperty cannot be read by its owning Block.

9.3.2.7 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 flow items can flow via the port.

Constraints

[1] FlowSpecifications cannot own operations or receptions (they can only own FlowProperties).

9.3.2.8 ItemFlow

Description

An Item Flow describes the flow of items across a connector or an association. It may constrain the item exchange between Blocks, Block usages or FlowPorts as specified by their FlowProperties. For example, a Pump connected to a Tank: the Pump has an out FlowProperty of type Liquid and the Tank has an in FlowProperty of type Liquid. To signify that only Water flow 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 that may be conveyed. For example: a label Water would imply that usages of Water might be transmitted over this ItemFlow. In addition, if there is an itemProperty (corresponds to the conveyed Classifier), then one can label the itemFlow with the itemProperty. For example, a label liquid:Water would imply that the itemFlow relays Water and this relay is associated with an itemProperty liquid of the ItemFlow, i.e., the liquid itemProperty is set once Water are relayed.

Attributes

- itemProperty : BlockProperty [0..1]
  An optional property that relates the flowing item to the instances of the connector's enclosing Block. This property is applicable only for ItemFlows assigned to connectors, the multiplicity is zero if the ItemFlow is assigned to an Association

Constraints

[1] An ItemFlow can be assigned to a Connector or an Association.

[2] An ItemFlow itemProperty is typed by a Block or by a ValueType.

[3] ItemProperty is specified in the context of the Block owning the Connector or Association.

[4] The type of itemProperty should be the same or a sub-type of the conveyedClassifier.

[5] The itemProperty multiplicity of itemProperty is zero in case the ItemFlow is assigned to an Association. The multiplicity is [0..1] if the ItemFlow is assigned to a Connector.

9.4 Usage Examples

9.4.1 Standard Ports

Figure 9.3 is a fragment of the ibd:PwrSys diagram used in the HybridSUV sample (Annex B). The ecu:PowerControlUnit part has three StandardPorts, each connected to a standard port of another part. Each of the standard ports in this example has one provided and one required interface that specify the messages that can be sent via the ports. For example, the I_ICECmds interface specifies the operations setMixture and setThrottle (Figure 9.4). This interface is provided by the ctrl port of InternalCombustionEngine and is required 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 send the messages setThrottle and setMixture to the ice:InternalCombustionEngine part from its ice port, across the connector to the ctrl port of ice:InternalCombustionEngine. By sending these messages, the PowerControlUnit can set the throttle and mixture of the InternalCombustionEngine. Inversely, the InternalCombustionEngine can report (notify) changes in its temperature, RPM and knockSensor by having the I_ICEData (Figure 9.4) as required interface on its ctrl port and connecting this port to the ice port of the PowerControlUnit where this interface is provided.

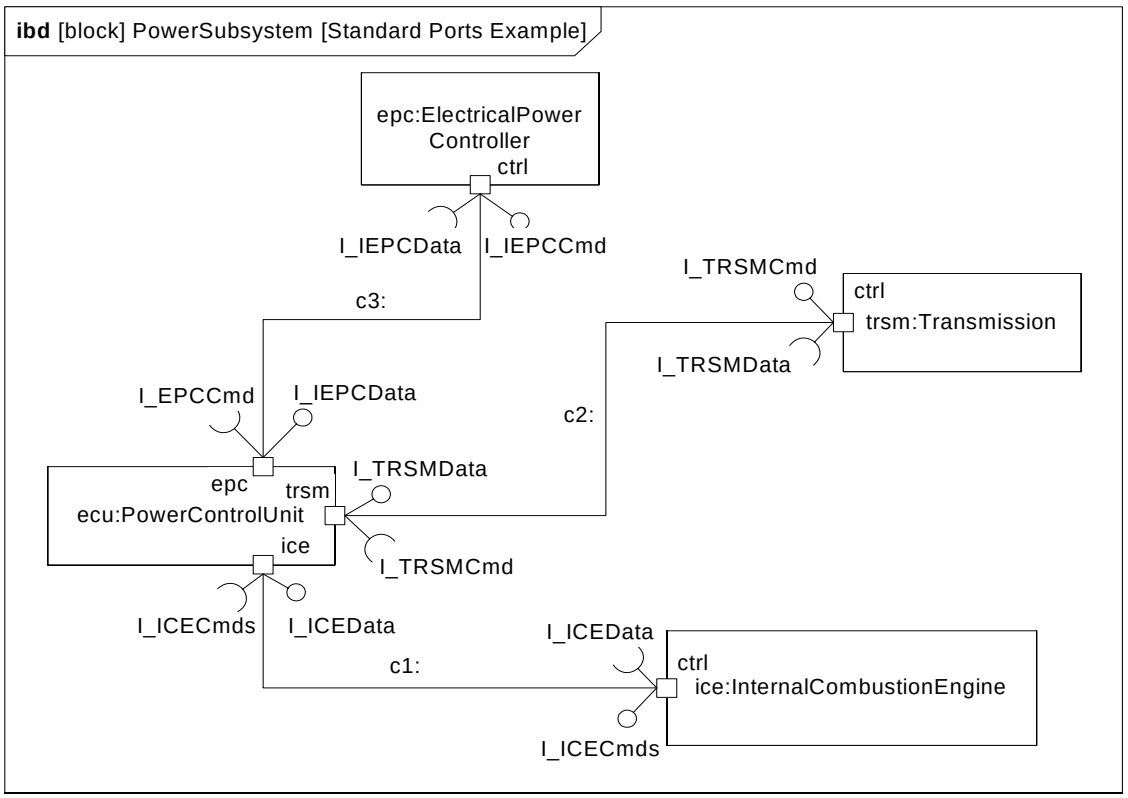


Figure 9.3 - Usage Example of StandardPorts

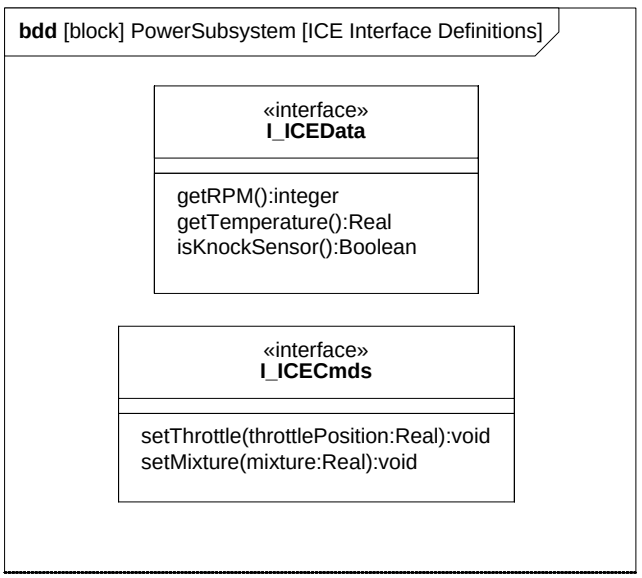


Figure 9.4 - Interfaces of the Internal Combustion Engine ctrl Standard Port

9.4.1.1 Atomic Flow Ports and Item Flows

Figure 9.5 is taken from the HybridSUV example in Appendix . Here 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 atomic flow ports of type Fuel 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 flow 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 flow 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 atomic flow ports of the engine's internal parts.

Figure 9.5 also 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.

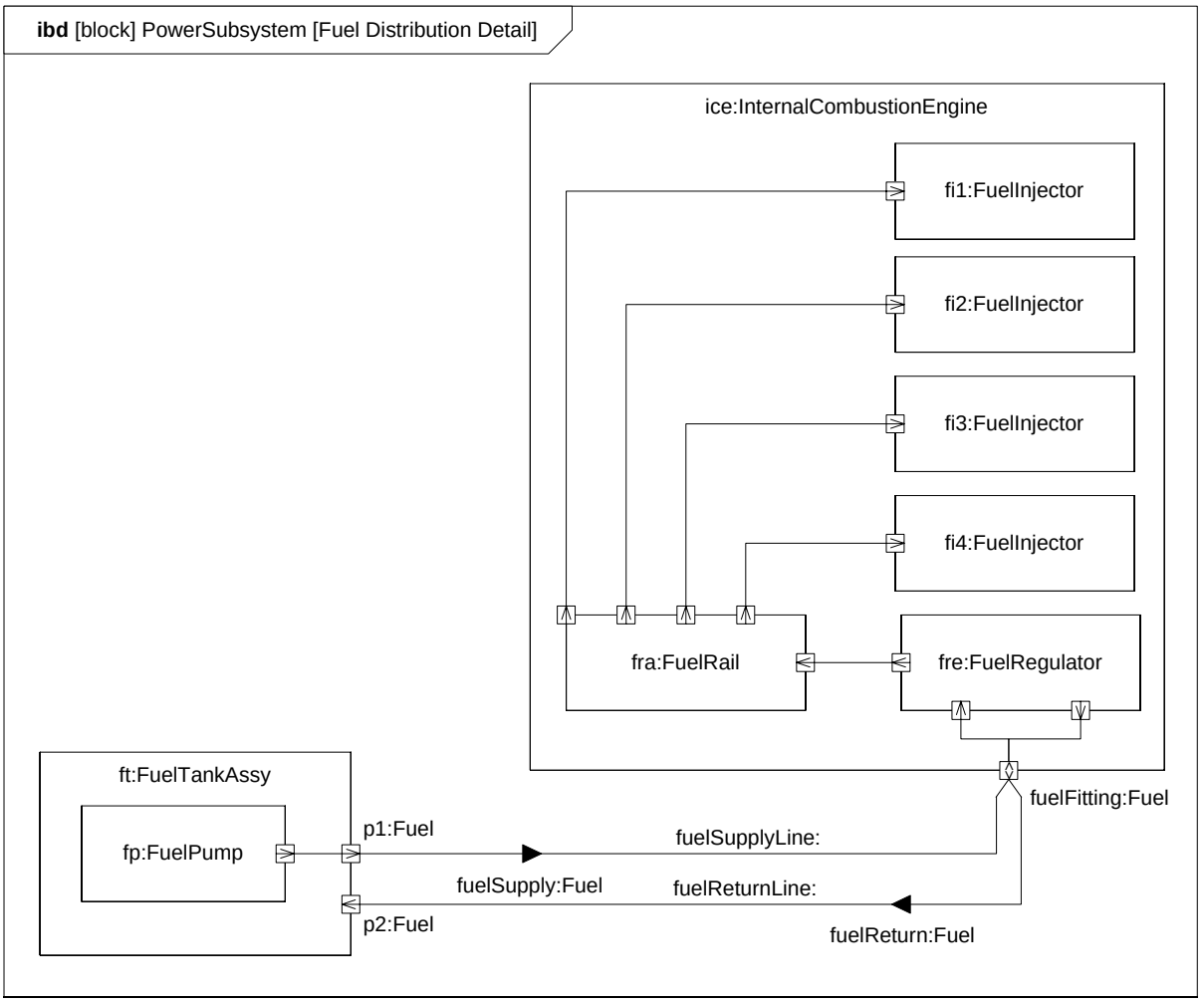


Figure 9.5 - Usage of Atomic Flow Ports in the HybridSUV Sample - ibd:FuelDist diagram

9.4.1.2 Non-Atomic Flow Ports and Flow Specification

Figure 9.6 taken from “Annex B: Sample Problem” shows a way to connect the PowerControlUnit to other parts over a CAN bus. Since connections over buses are characterized by broadcast asynchronous communications, flow ports are used to connect the parts to the CAN bus. To specify the flow between the flow ports, we need to specify Flow Specifications as done in Figure 9.7. Here the flow specification has three flow properties: an out flow property of type signal (ICEData) and two in flow properties of type float. This allows the InternalCombustionEngine to transmit an ICEData signal via its fp flow port which will be transmitted over the CAN bus to the ice port of PowerControlUnit (a conjugated flow port typed by the FS_ICE flow specification). 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 the FS_ICE flow specification.

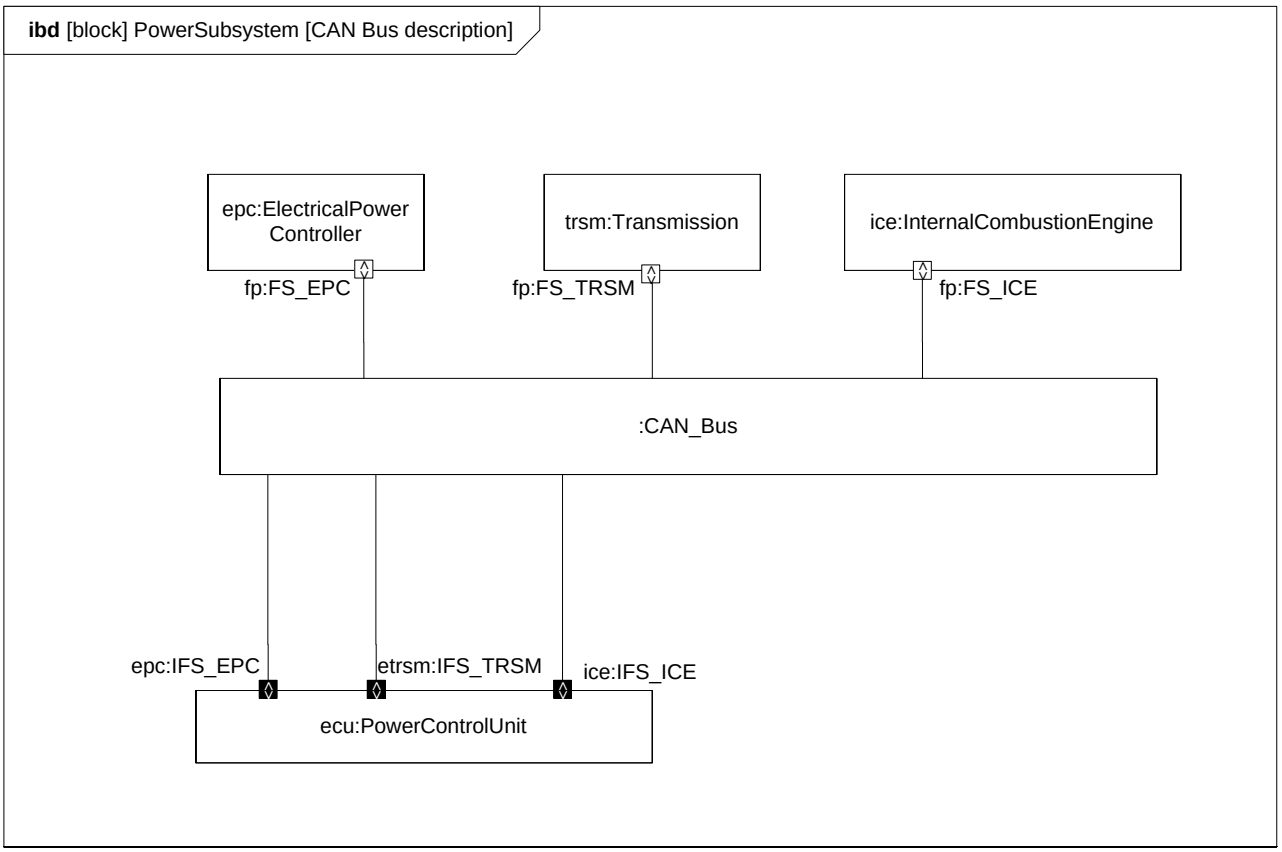


Figure 9.6 - Using Flow Ports to Connect the PowerControlUnit to the ElectricalPowerController, Transmission and InternalCombustionEngine over a CAN bus

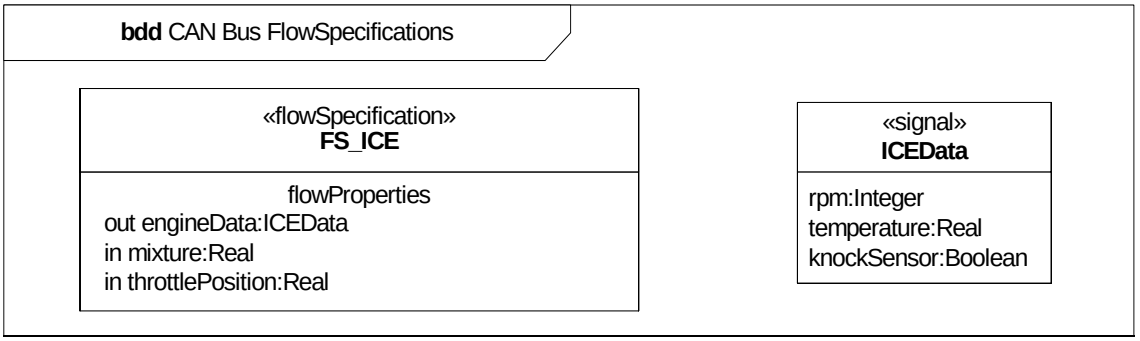


Figure 9.7 - Flow Specification for the InternalCombustionEngine flow port to allow its connection over the CAN bus

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, dimensions, and 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 must 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 which 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 which 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 which 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 C: 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) must 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. The properties of this block define the parameters of the constraint. 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

Tables in the following sections provide a high-level summary of graphical elements available in SysML diagrams. A more complete definition of SysML diagram elements, including the different forms and combinations in which they may appear, is provided in Annex G: BNF Diagram Syntax Definitions.

10.2.1 Block Definition Diagram

The diagram elements described in this section are additions to the Block Definition diagram described in Chapter 8, Blocks.

10.2.1.1 Graphical Nodes

Table 10.1 - Graphical nodes defined in Block Definition diagrams

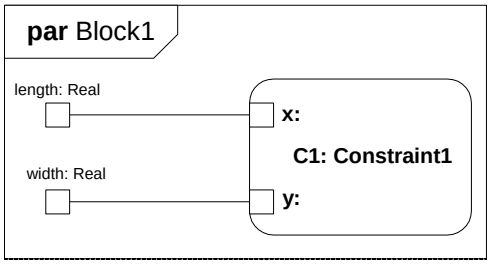
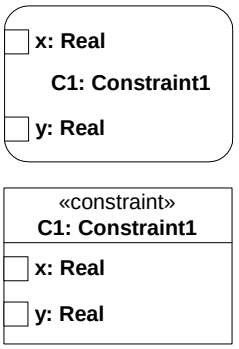
Element Name	Concrete Syntax Example	Metamodel Reference
ConstraintBlock	«constraint» **ConstraintBlock1** *constraints* {{L1} x > y} nested: ConstraintBlock2 *parameters* x: Real y: Real	**SysML::ConstraintBlocks::ConstraintBlock**

10.2.2 Parametric Diagram

The diagram elements described in this section are additions to the Internal Block Diagram described in Chapter 8: Blocks. The Parametric Diagram includes all of the notations of an Internal Block Diagram, subject only to the restrictions described in “Parametric Diagram” on page 74.

10.2.2.1 Graphical Nodes

Table 10.2 - Graphical nodes defined in Parametric diagrams.

Element Name	Concrete Syntax Example	Metamodel Reference
ParametricDiagram		**SysML::Constraint-Blocks::ConstraintBlock** **SysML::Blocks::Block**
ConstraintProperty		**SysML::ConstraintBlocks::ConstraintProperty**

10.3 UML Extensions

10.3.1 Diagram Extensions

10.3.1.1 Block Definition Diagram

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.”

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, must either be bound directly to a constraint parameter, or contain a property that is bound to one (through any number of levels of containment).

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.

«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. The stereotype ConstraintProperty is applied to a constraint property, but only the shorthand keyword «constraint» is used when shown on an internal property**.**

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 ConstraintBlocks

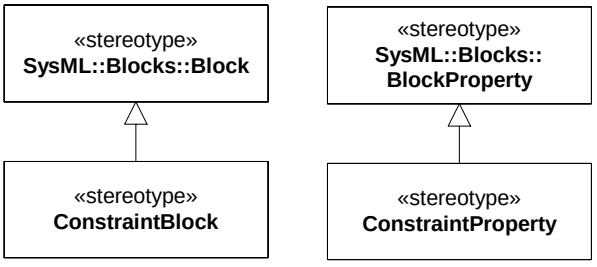


Figure 10.1 - Stereotypes defined in SysML ConstraintBlocks package

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 includes the definition of one or more parameters which can be bound to properties in a specific context where the constraint is used. In a containing block where the constraint is used, the constraint block is used to type a property that holds the usage. A constraint parameter is a property of a constraint block which may be bound to a property in a surrounding usage context. All properties of a constraint block define parameters of the constraint block, with the exception of constraint properties that hold internally nested usages of other constraint blocks.

[1] A constraint block may not own any structural or behavioural 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.

10.3.2.2 ConstraintProperty

Description

A Constraint Property is a Block Property that is typed by a Constraint Block, and owned by a containing block. Parameters of the constraint property may be bound to properties of the surrounding context using binding connectors as defined in Chapter 8: Blocks.

Constraints

[1] The type of a constraint property must be a constraint block.

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 must 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 10.2. 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.

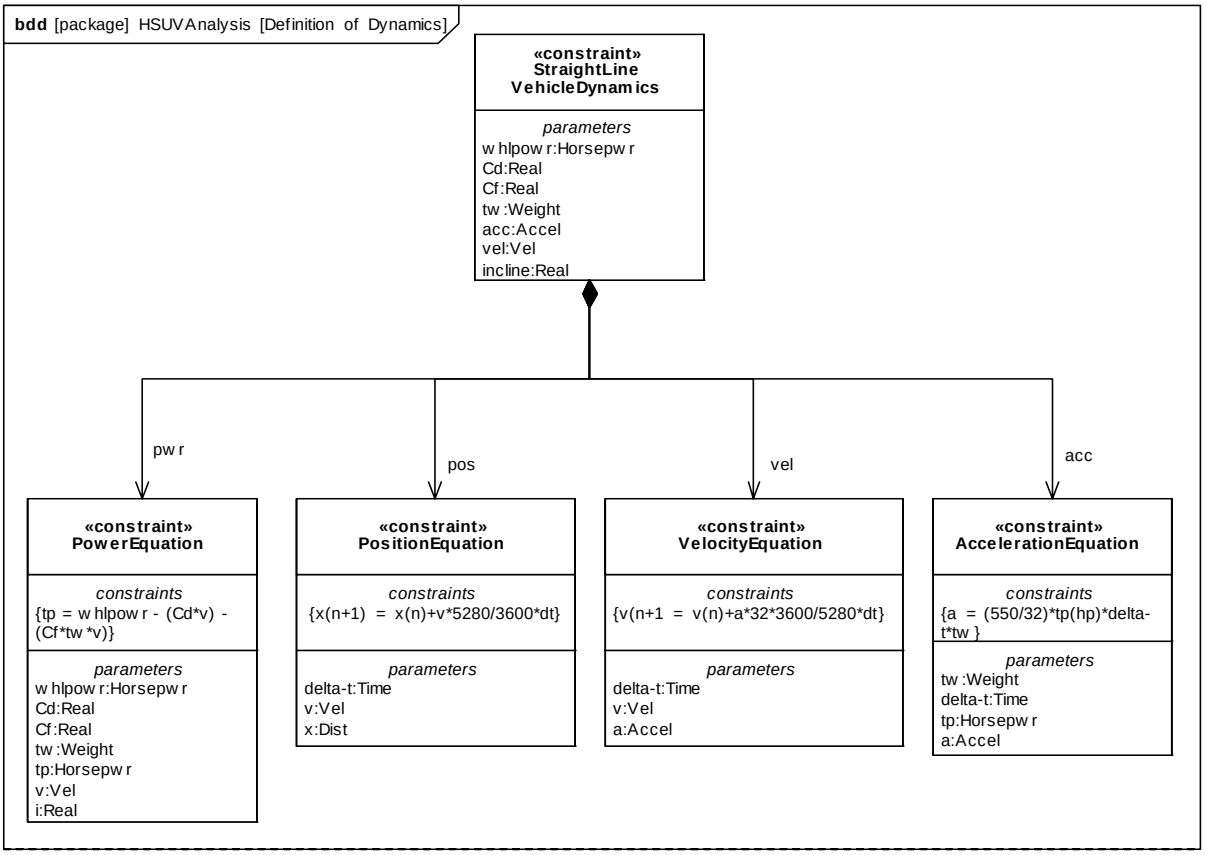


Figure 10.2 - Constraint block definitions in a Block Definition diagram

10.4.2 Usage of Constraint Blocks on a Parametric Diagram

Figure 10.3 shows the use of constraint properties on a parametric diagram (note that this is a subset of the corresponding diagram in the sample problem). 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 connected to constraint parameters on at least one end. The Sample Problem in Annex B: Sample Problem provides definitions of the containing EconomyContext block for which this parametric diagram is shown.

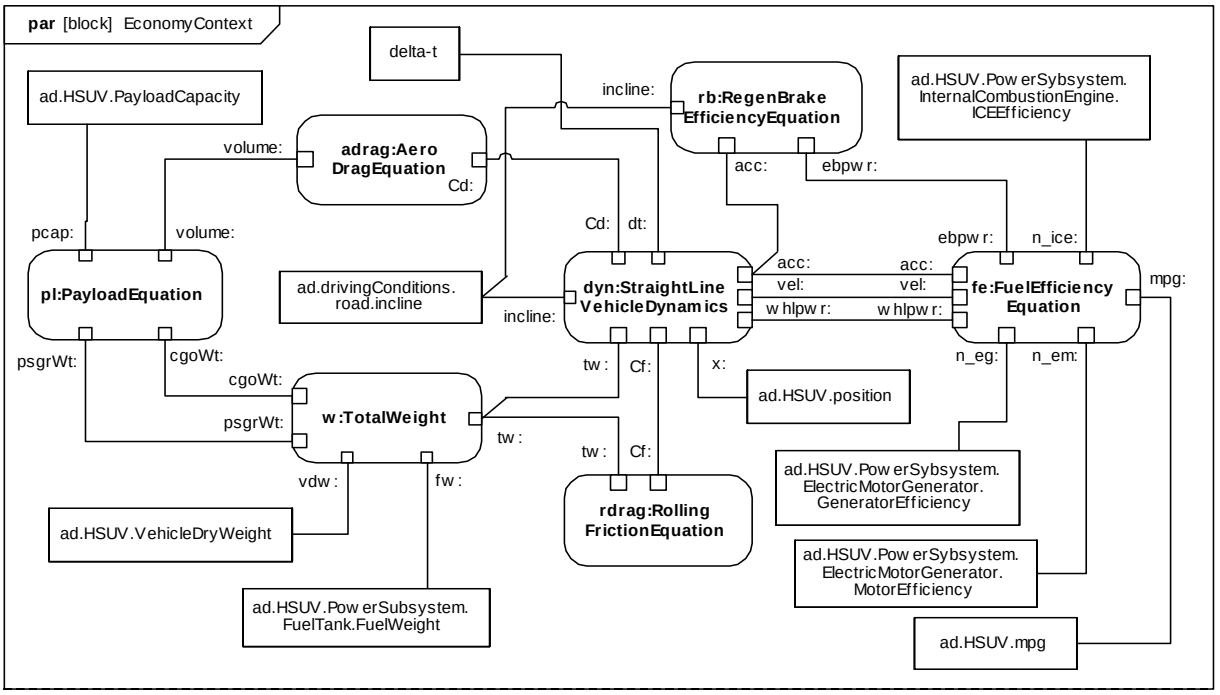


Table 10.3 - Constraints on a parametric diagram

Part III - Behavioral Constructs

This Part specifies the dynamic, behavioral constructs used in SysML behavioral diagrams, including the activity diagram, sequence diagram, state machine diagram, and use case diagram. The behavioral constructs are defined in Chapter 11, “Activities,” Chapter 12, “Interactions,” Chapter 13, “State Machines,” and Chapter 14, “Use Cases.” The activities chapter defines the extensions to UML 2.0 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 flow of control and flow of inputs and outputs among actions. The state machines chapter describes the constructs used to specify state based behavior in terms of system states and their transitions. The interactions chapter defines the constructs for describing message based behavior used in sequence diagrams. The use case chapter describes behavior in terms of the high level functionality and uses of a system, that are further specified in the other behavioral diagrams referred to above.

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 2.1 Activity diagrams. For additional information see extensions for Enhanced Functional Flow Block Diagrams in Annex C: Non-normative Extensions.

11.1.1 Control as Data

SysML extends control in activity diagrams as follows.

- In UML 2.1 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 ControlValue 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).

11.1.1.1 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.

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.

Activities as classes

In UML 2.1, all behaviors including activities are classes, and their instances are executions. Behaviors can appear on block definition and class 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 section 11.3.1.

Timelines

Timelines can be developed to correspond to SysML activity diagrams. The simple time model can be used to represent time and constraints can be used to annotate the activity diagram to represent timing constraints. Although UML 2 timing diagram was not included in this version of SysML, it does provide a capability to represent a timeline and can be used as a complement to SysML. More sophisticated SysML modeling techniques can incorporate constraint blocks from Chapter 10, “Constraint Blocks” to specify resource and related constraints on the properties of the inputs, outputs, and other system properties. (Note: refer to “ObjectNode” on page 91 for constraining properties of object nodes). This in turn could be used to drive a simulation engine.

11.2 Diagram Elements

Table 11.1 - Graphical nodes included in activity diagrams

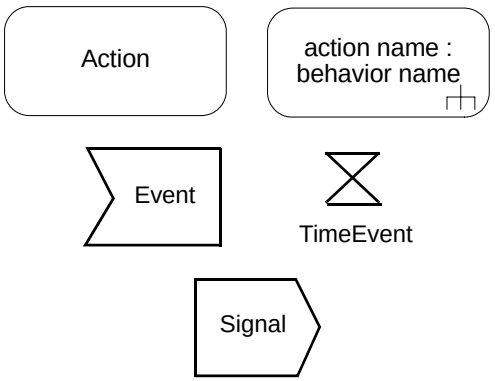
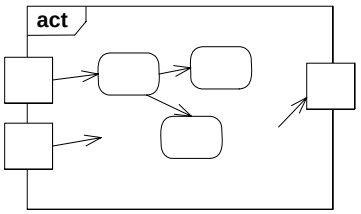
Node Name	Concrete Syntax	Abstract Syntax Reference
Action, CallBehaviorAction, AcceptEventAction, Send-SignalAction		**UML4SysML::Action, UML4SysML::CallBehaviorAction UML4SysML::AcceptEventAction UML4SysML::SendSignalAction**
Activity		**UML4SysML::Activity**

Table 11.1 - Graphical nodes included in activity diagrams

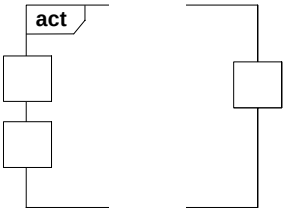
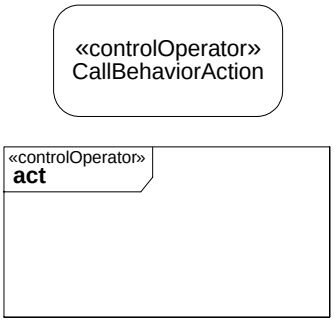
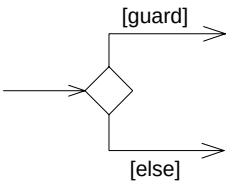
Node Name	Concrete Syntax	Abstract Syntax Reference
ActivityFinal		**UML4SysML::ActivityFinalNode**
ActivityNode	**See ControlNode and ObjectNode.**	**UML4SysML::ActivityNode**
ActivityParameterNode		**UML4SysML::ActivityParameter-Node**
ControlNode	**See DecisionNode, FinalNode, ForkNode, Initial-Node,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

Table 11.1 - Graphical nodes included in activity diagrams

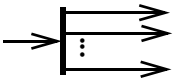
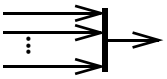
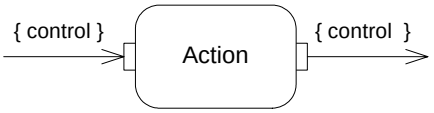
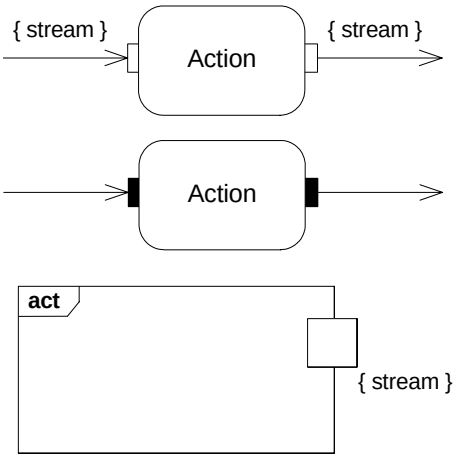
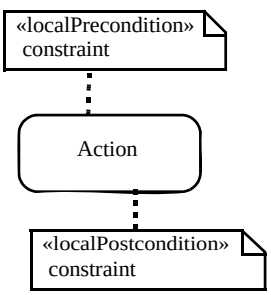
Node Name	Concrete Syntax	Abstract Syntax Reference
ForkNode		**UML4SysML::ForkNode**
InitialNode		**UML4SysML::InitialNode**
JoinNode		**UML4SysML::JoinNode**
isControl		**UML4SysML::Pin.isControl**
isStream		**UML4SysML::Parameter.isStream**
Local pre- and postconditions		**UML4SysML::Action.localPrecondition,** **UML4SysML::Action.localPostcondition**

Table 11.1 - Graphical nodes included in activity diagrams

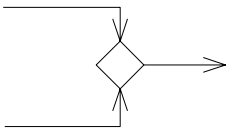
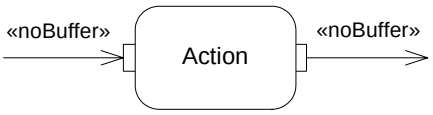
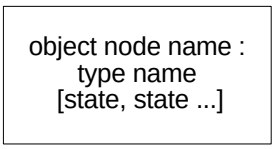
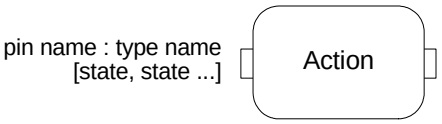
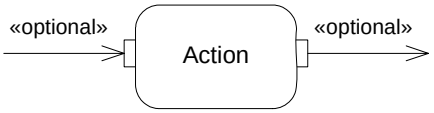
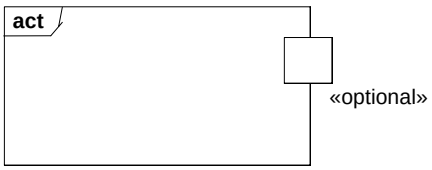
Node Name	Concrete Syntax	Abstract Syntax Reference
MergeNode		**UML4SysML::MergeNode**
NoBuffer		**SysML::Activities::NoBuffer**
ObjectNode	 	**UML4SysML::OjectNode and its children, SysML:: Activities::ObjectNode**
Optional	 	**SysML::Activities::Optional**
OverWrite		**SysML::Activities::Overwrite**

Table 11.1 - Graphical nodes included in activity diagrams

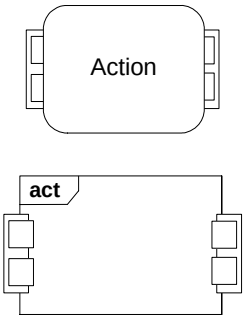
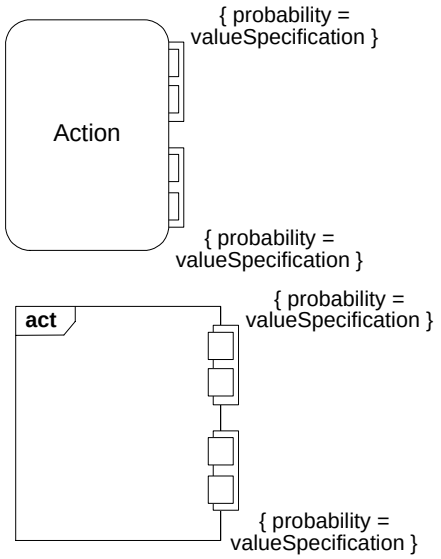
Node Name	Concrete Syntax	Abstract Syntax Reference
ParameterSet		**UML4SysML::ParameterSet**
Probability		**SysML::Activities::Probability**

Table 11.1 - Graphical nodes included in activity diagrams

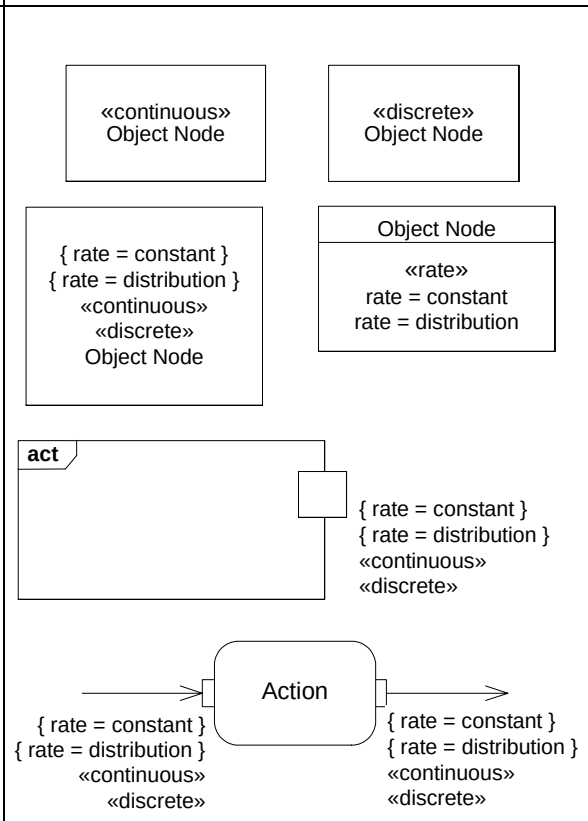
Node Name	Concrete Syntax	Abstract Syntax Reference
Rate		**SysML::Activities::Rate, SysML::Activities::Continuous, SysML::Activities::Discrete**

Table 11.2 - Graphical paths included in activity diagrams

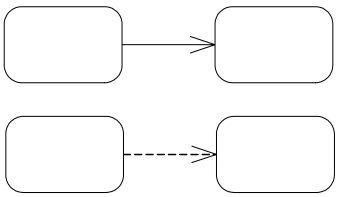
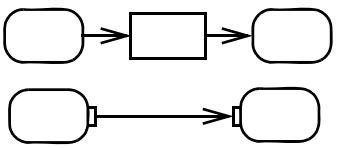
Path Name	Concrete Syntax	Abstract Syntax Reference
ActivityEdge	**See ControlFlow and ObjectFlow**	**UML4SysML::ActivityEdge**
ControlFlow		**UML4SysML::ControlFlow SysML::Activities::ControlFlow**
ObjectFlow		**UML4SysML::ObjectFlow**

Table 11.2 - Graphical paths included in activity diagrams

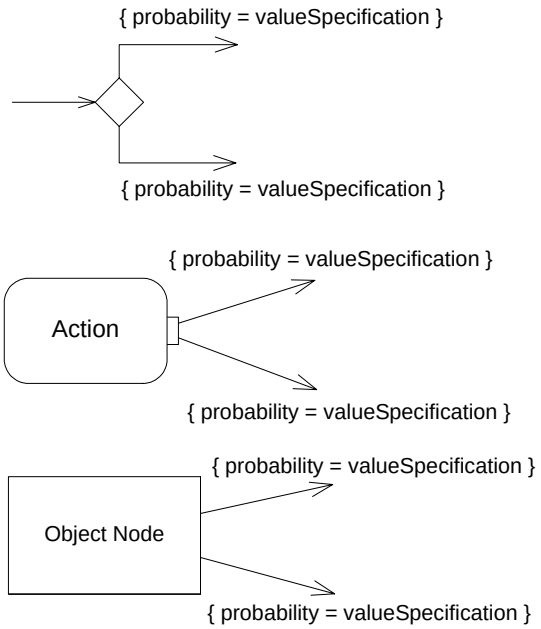
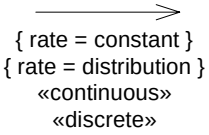
Path Name	Concrete Syntax	Abstract Syntax Reference
Probability		**SysML::Activities::Probability**
Rate		**SysML::Activities::Rate, SysML::Activities::Continuous, SysML::Activities::Discrete**

Table 11.3 - Other graphical elements included in activity diagrams

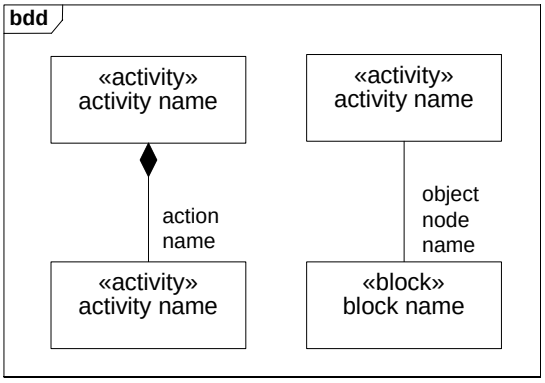
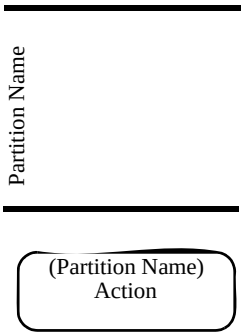
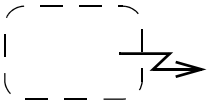
Element Name	Concrete Syntax	Abstract Syntax Reference
In Block Definition Diagrams, Activity, Association		**SysML::Activities, Diagram Usage for Block Definition Diagrams**

Table 11.3 - Other graphical elements included in activity diagrams

Element Name	Concrete Syntax	Abstract Syntax Reference
ActivityPartition		**UML4SysML::ActivityPartition**
InterruptibleActivity Region		**UML4SysML::InterruptibleActivity-Region**

11.3 UML Extensions

11.3.1 Diagram Extensions

The following specify diagram extensions to the notations defined in Chapter 17: Profiles & Model Libraries.

11.3.1.1 Activity

Notation

In UML 2.1, 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 classes 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 classes, 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. The lower multiplicity on the part end is always zero, because there will be some time during the execution of the containing activity that the lower level activity is not executing. See Constraints sections below.

Activities in block definition diagrams or class diagrams appear as regular blocks or classes, using the «activity» keyword for clarity, as shown in Figure 11.1. See example in “Usage Examples” on page 97. This provides a means for representing activity decomposition in a way that is similar to classical functional decomposition hierarchies. The names of the CallBehaviorActions that correspond to the association can be used as end names of the association on the part end. Activities in block definition diagrams or class 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.

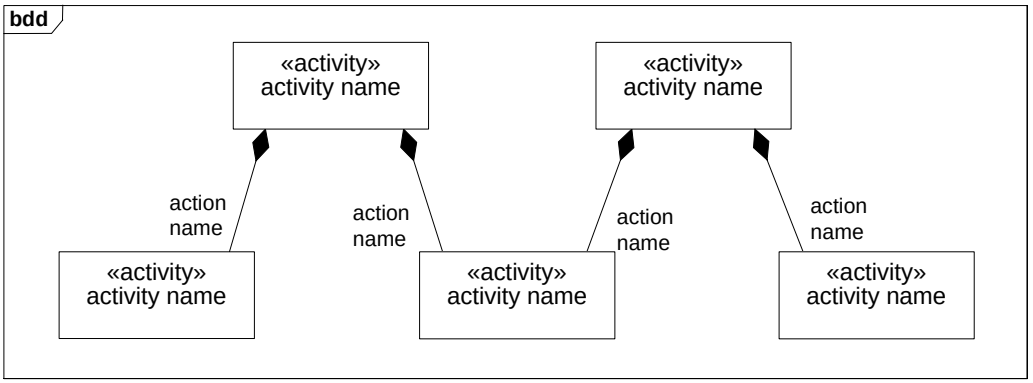


Figure 11.1 - Block definition diagram with activities as blocks.

Constraints

The following constraints apply when composition associations in block definition diagrams or class diagrams are defined between activities:

[1] The part end name must be the same as the name of a synchronous CallBehaviorAction in the composing activity. If the action has no name, and the invoked activity is only used once in the calling activity, then the end name is the same as name of the invoked activity.

[2] The part end activity must be the same as the activity invoked by the corresponding CallBehaviorAction.

[3] The lower multiplicity at the part end must be zero.

[4] The upper multiplicity at the part end must be 1 if the corresponding action invokes a nonreentrant behavior.

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.

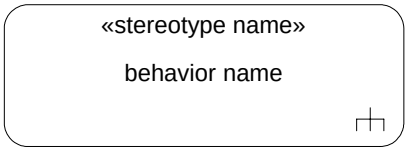


Figure 11.2 - CallBehaviorAction notation.with behavior stereotype

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.

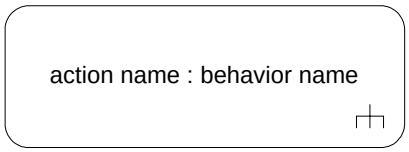


Figure 11.3 - CallBehaviorAction notation.with action name

11.3.1.3 ControlFlow

Presentation Option

Control flow may be notated with a dashed line and stick arrowhead, as shown in Figure 11.3.

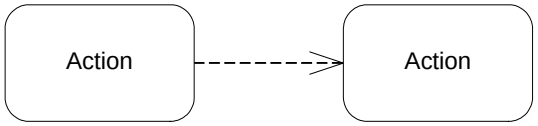


Figure 11.4 - Control flow notation

11.3.1.4 ObjectNode

Notation

See Section 11.3.1.1concerning activities appearing in block definition diagrams or class diagrams. Associations can be used between activities and classifiers (classes, blocks, or datatypes) that are the type of object nodes 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 and happen to be contained by the object node at the time the link exists. The names of the object node that correspond to the association can be used as end names of the association on the end towards the object node type. Like any association end or property, these can be the subject of parametric constraints, design values, units and dimensions. The upper multiplicity on the object node end restricts the number of instances of the item type that can reside in the object node at one time, which must be lower than the maximum amount allowed by the object node itself. The lower multiplicity on the object node end is always zero, because there will be some time during the execution of the containing activity that there is no item in the object node. 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 “Usage Examples” on page 97.

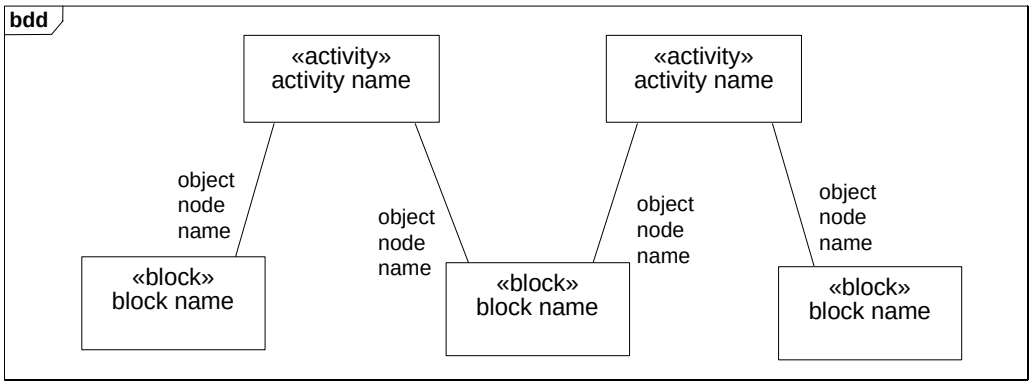


Figure 11.5 - Class or block definition diagram with activities as classes associated with types of object nodes

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.

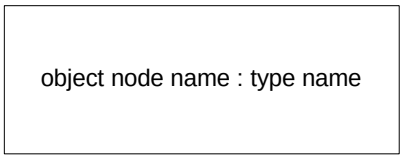


Figure 11.6 - ObjectNode notation in activity diagrams

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.

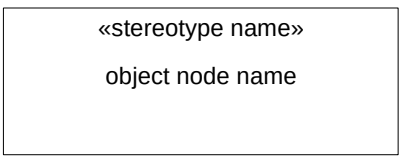


Figure 11.7 - ObjectNode notation in activity diagrams

Constraints

The following constraints apply when associations in block definition diagrams and class diagrams are defined between activities and classifiers typing object nodes:

[1] The end name towards the object node type is the same as the name of an object node in the activity at the other end.

[2] The classifier must be the same as the type of the corresponding object node.

[3] The lower multiplicity at the object node type end must be zero.

[4] The upper multiplicity at the object node type end must be equal to the upper bound of the corresponding object node.

11.3.2 Stereotypes

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

Package Activities

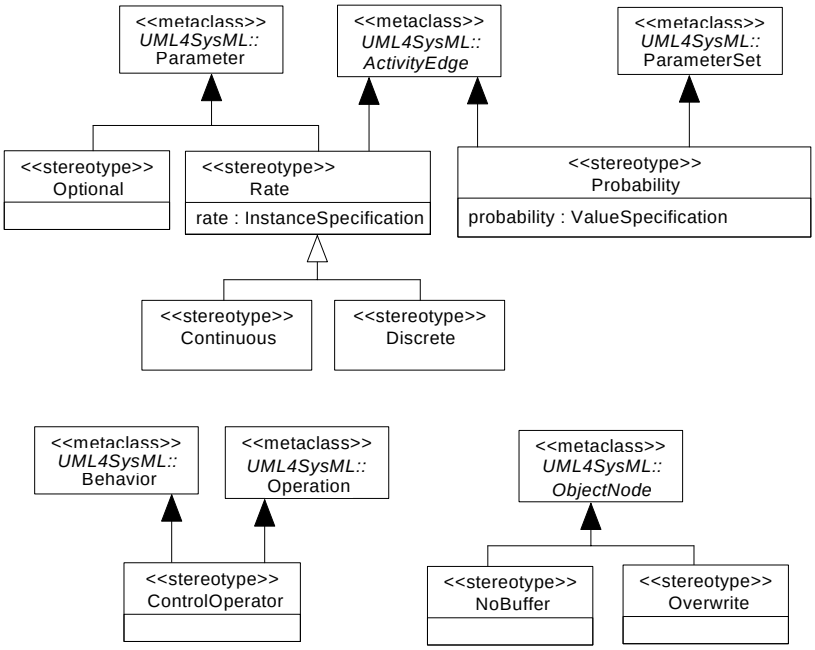


Figure 11.8 - Abstract Syntax for SysML Activity Extensions

11.3.2.1 Continuous

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 “Rate” on page 96. 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.

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 “ControlValue” on page 96). 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.

Constraints

[1] When the «controlOperator» stereotype is applied, the behavior or operation must 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.

[2] A behavior must have the «controlOperator» stereotype applied if it is a method of an operation that has the «controlOperator» stereotype applied.

11.3.2.3 Discrete

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

Constraints

[1] The «discrete» and «continuous» stereotypes cannot be applied to the same element at the same time.

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, 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 an object node that are refused by outgoing edges, or action for input pins, are held until they can leave the object node.

Constraints

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

11.3.2.5 Overwrite

Description

When the «overwrite» stereotype is applied to object nodes, a token arriving at a full object node replaces the ones already there (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 replaced is the one that would be the last to be selected according to the ordering kind for the node. For FIFO ordering, this is the most recently added token, for LIFO it is the least recently added token. 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.

Constraints

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

11.3.2.6 Optional

Description

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

Constraints

[1] A parameter with the «optional» stereotypes applied must have multiplicity.lower equal to zero, otherwise multiplicity.lower must be greater than zero.

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 must 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 must 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.

Constraints

[1] The «probability» stereotype can only be applied to activity edges that have decision nodes or object nodes as sources, or to output parameter sets.

[2] When the «probability» stereotype is applied to an activity edge, then it must be applied to all edges coming out of the same source.

[3] When the «probability» stereotype is applied to an output parameter set, it must also be applied to all the parameter sets of the behavior or operation owning the original parameter set.

[4] When the «probability» stereotype is applied to an output parameter set, all the output parameters must be in some parameter set.

11.3.2.8 Rate

Description

When the «rate» stereotype is applied to an activity edge, it specifies the number of objects and values that traverse the edge per time interval, that is, the rate 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 must 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 Section 11.3.2.9, “Model Library,” on page 96, and “Discrete” on page 94. The «rate» stereotype has a rate property of type InstanceSpecification. The values of this property must be instances of classifiers stereotyped by «valueType» or «distributionDefinition», see Chapter 8: Blocks on page 33, and they must use units and dimensions appropriate to rates of flow. In particular, the denominator for units used in the rate property must be time units.

Constraints

[1] The value of the rate attribute must be an instance specification that is typed by a classifier that is stereotyped by SysML::«valueType» or SysML::«distributionDefinition».

[2] When the «rate» stereotype is applied to a parameter, the parameter must be streaming.

[3] The rate of a parameter must be less than or equal to rates on edges that come into or go out from pins and parameters nodes corresponding to the parameter.

11.3.2.9 Model Library

The SysML model library for activities is shown in Figure 11.9.

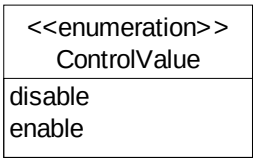


Figure 11.9 - Control values.

11.3.2.10 ControlValue

Description

The ControlValue enumeration is a type for treating control values as data (see Section 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).

Constraints

[1] UML4SysML::ObjectNode::isControlType is true for object nodes with type ControlValue.

11.4 Usage Examples

The following examples illustrate modeling continuous systems (see Continuous Systems in Section 11.1). Figure 11.10 shows a simplified model of driving and braking in a car that has an automatic braking system. 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 control 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 Figures 11.11 and 11.12. An alternative notation for this activity decomposition is shown in Figure 11.13.

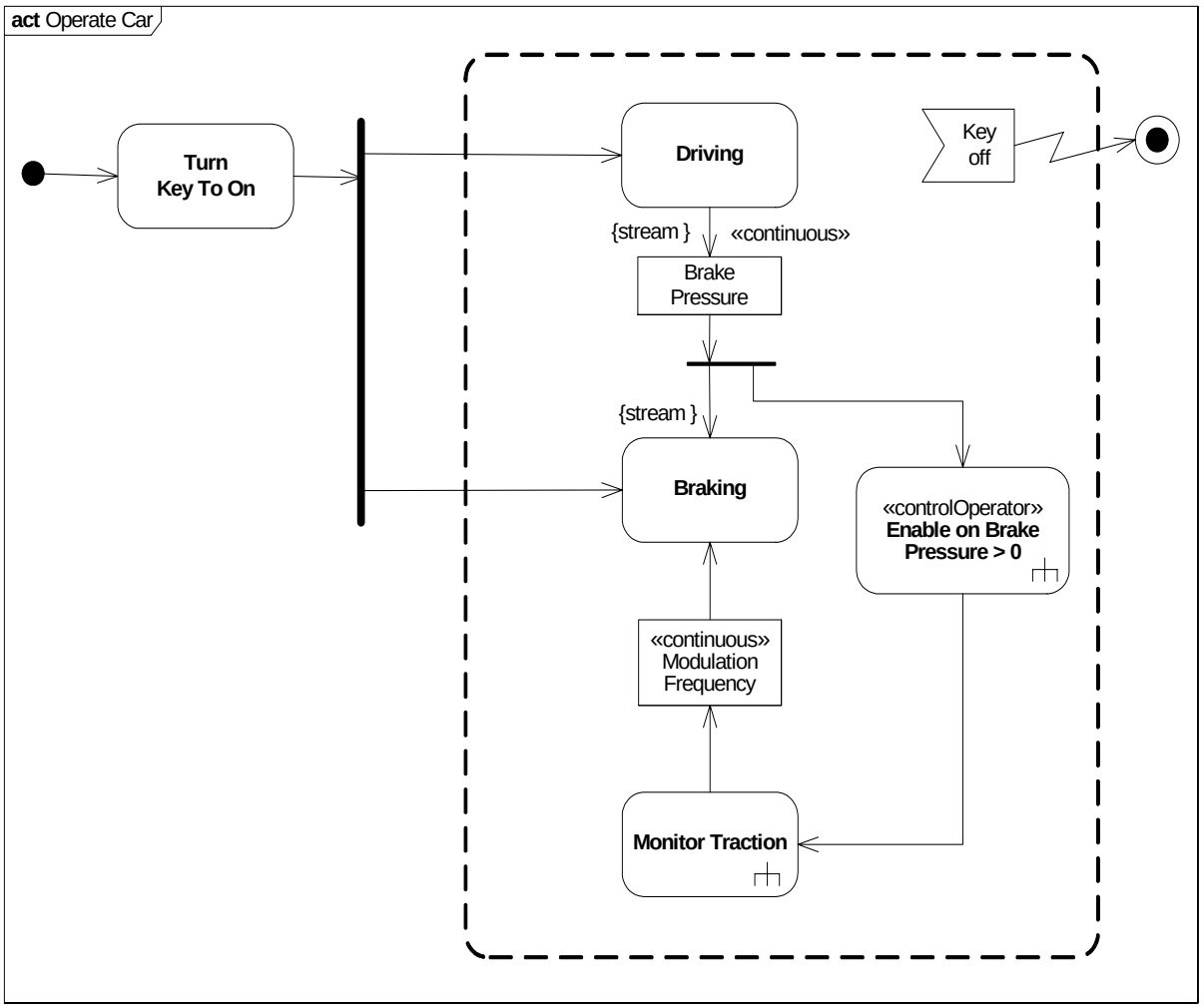


Figure 11.10 - Continuous system example 1.

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.

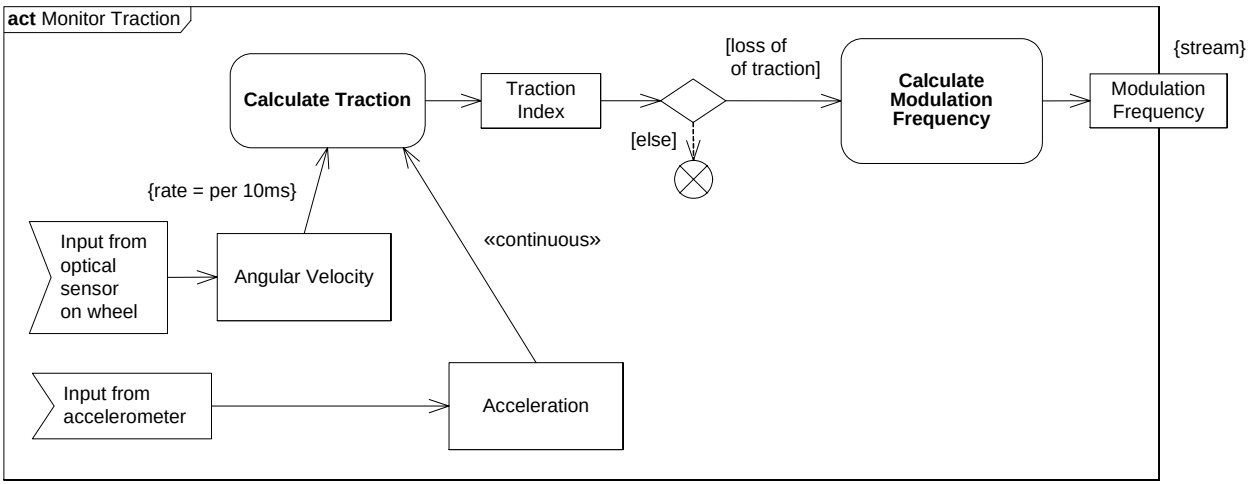


Figure 11.11 - Continuous system example 2.

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.

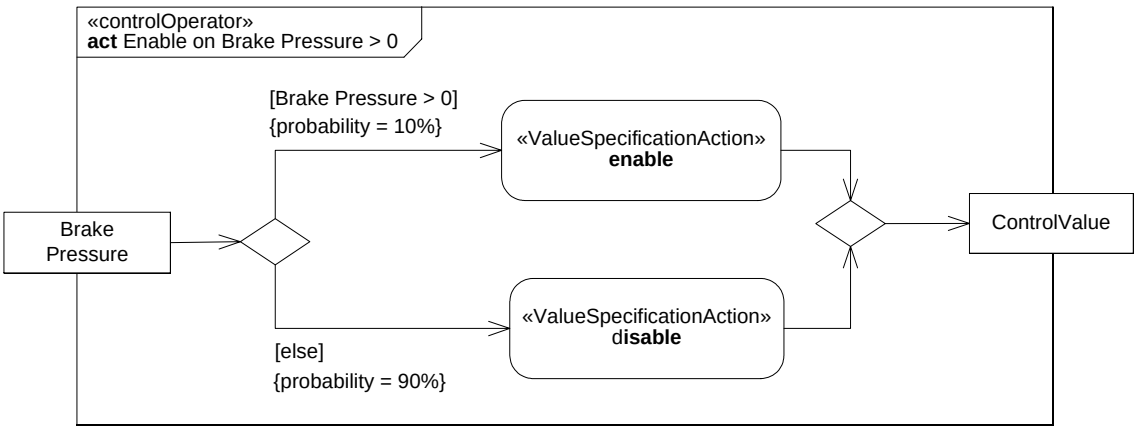


Figure 11.12 - Continuous system example 3

Figure 11.13 shows a block definition diagram with composition associations between the activities 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.

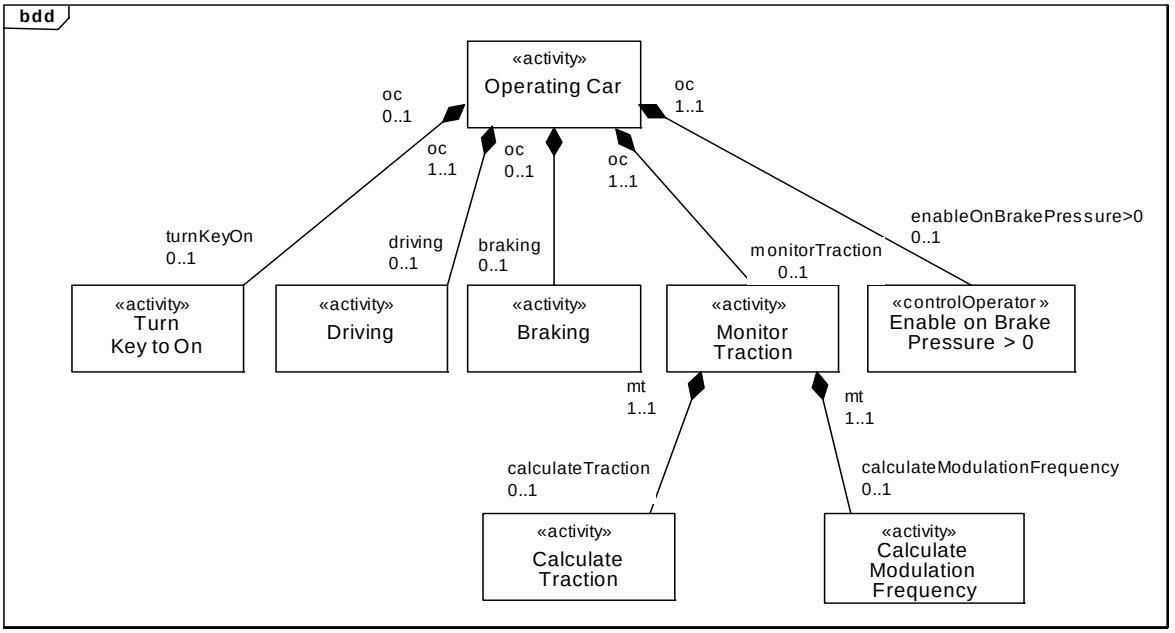


Figure 11.13 - Example block definition diagram for activity decomposition

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. In an instance of Operating Car, which is one execution of it, instances of Break Pressure and Modulation Frequency are linked to the execution instance when they are in the object nodes of the activity.

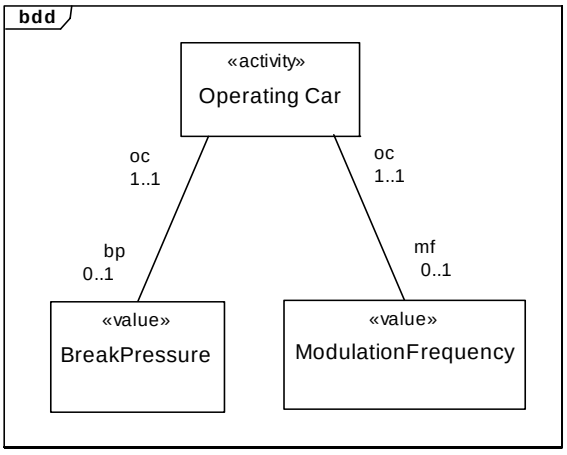


Figure 11.14 - Example block definition diagram for object node types

12 Interactions

12.1 Overview

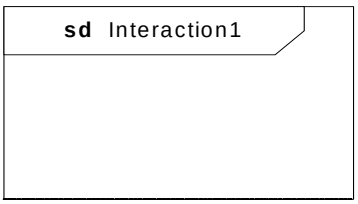
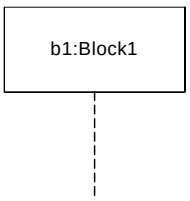
Interactions are used to describe interactions between entities. UML 2.1 Interactions are supported by four diagram types including the Sequence Diagram, Communications Diagram, Interaction Overview Diagram, and Timing Diagram. The Sequence Diagram is the most common of the Interaction Diagrams. SysML 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 system engineering needs.

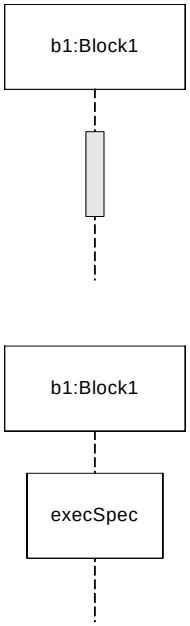
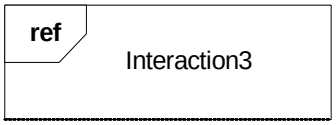
The sequence diagram describes the flow of control between actors and systems (blocks) or between parts of a system. This diagram represents the sending and receiving of messages between the interacting entities called lifelines, where time is represented along the vertical axis. The sequence diagrams 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

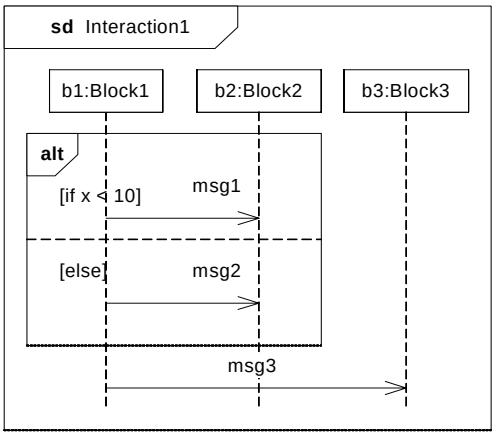
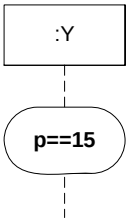
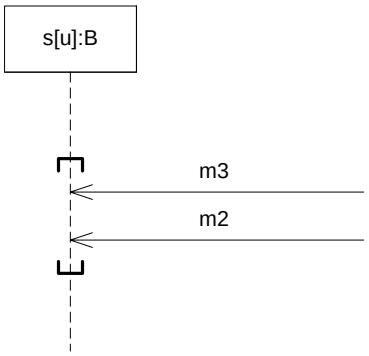
12.2.1 Sequence Diagram

Table 12.1 - Graphical nodes included in sequence diagrams[1].

Node Name	Concrete Syntax	Abstract Syntax Reference
SequenceDiagram		**UML4SysML::Interaction**
Lifeline		**UML4SysML::Lifeline**

Node Name	Concrete Syntax	Abstract Syntax Reference
Execution Specification		**UML4SysML::ExecutionSpecification**
InteractionUse		**UML4SysML::InteractionUse**

1. Table is compliant with UML 2.0 Superstructure source document dated 050704.

Node Name	Concrete Syntax	Abstract Syntax Reference
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
StateInvariant / Continuations		**UML4SysML::Continuation** **UML4SysML::StateInvariant**
Coregion		**UML4SysML::CombinedFragment** (under *parallel*)

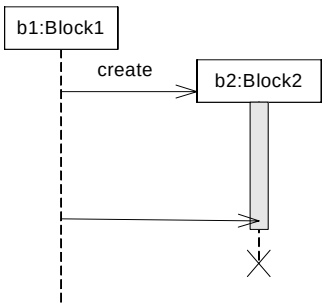
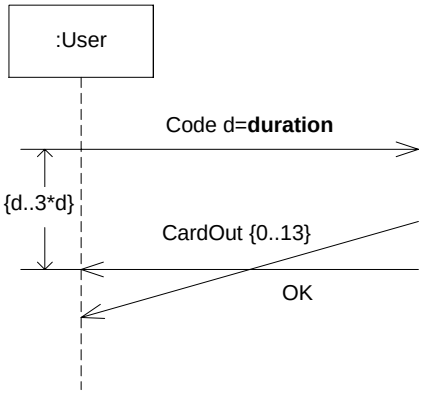
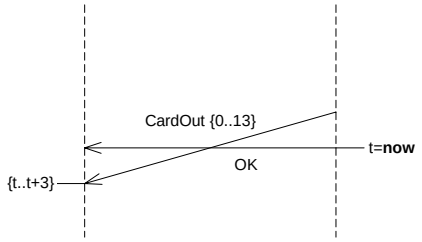
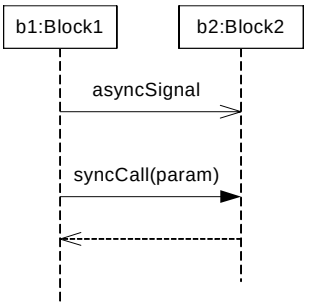
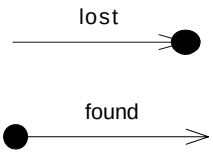
Node Name	Concrete Syntax	Abstract Syntax Reference
CreationEvent **DestructionEvent**		**UML4SysML::CreationEvent** **UML4SysML::DestructionEvent**
DurationConstraint **Duration** **Observation**		**UML4SysML::Interactions**
TimeConstraint **TimeObservation**		**UML4SysML::Interactions**

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**

12.3 UML Extensions

12.3.1 Diagram Extensions

The following specify diagram extensions to the notations defined in Chapter 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 system engineering needs.

12.4 Usage Examples

12.4.1 Sequence Diagrams

The diagram in Figure 12.1 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.

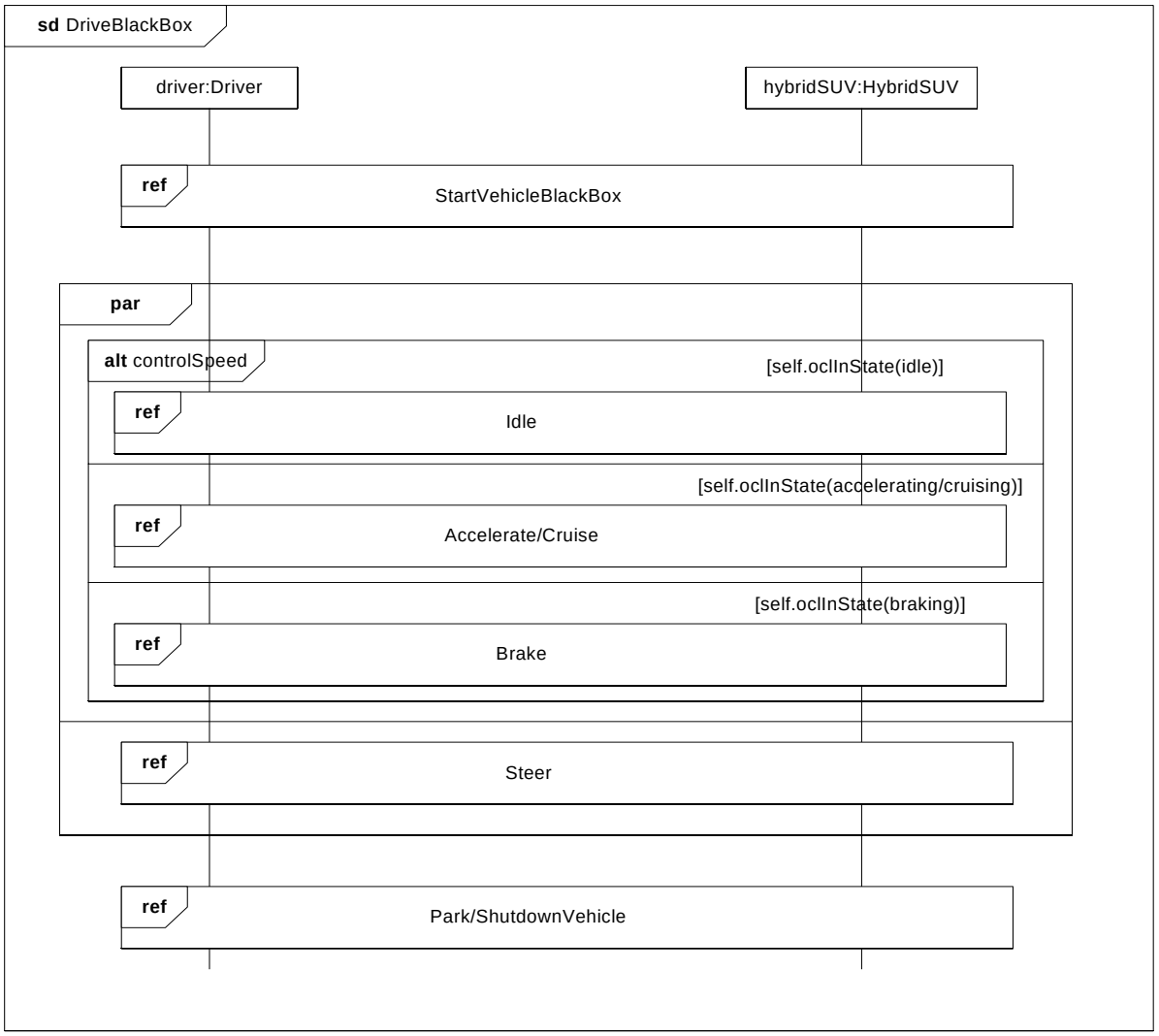


Figure 12.1 - Hierarchical Sequence Diagram illustrating system behavior for “Operate the vehicle” use case

The diagram in Figure 12.2 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.

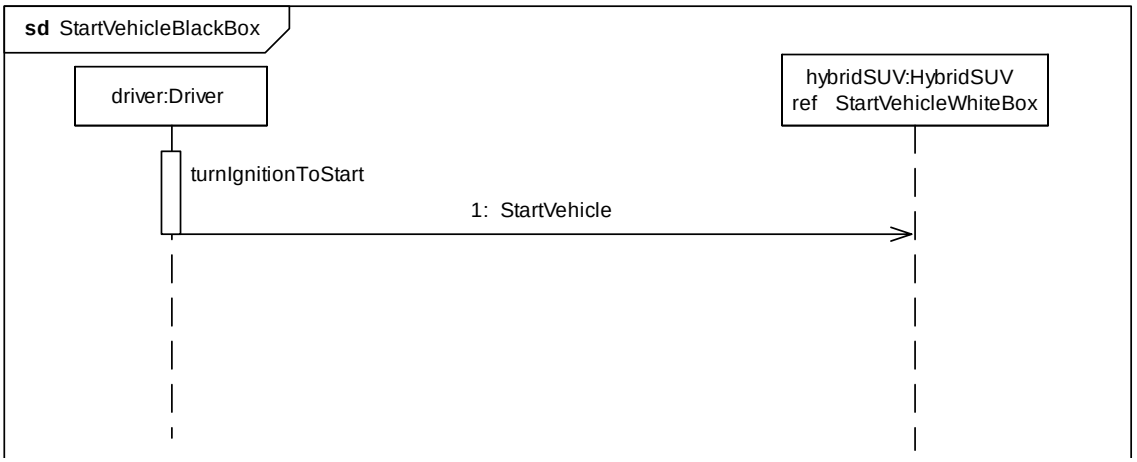


Figure 12.2 - Black box interaction during "starting the Hybrid SUV"

The diagram in Figure 12.3 shows the sequence of communication that occurs inside the HybridSUV when the vehicle is started successfully.

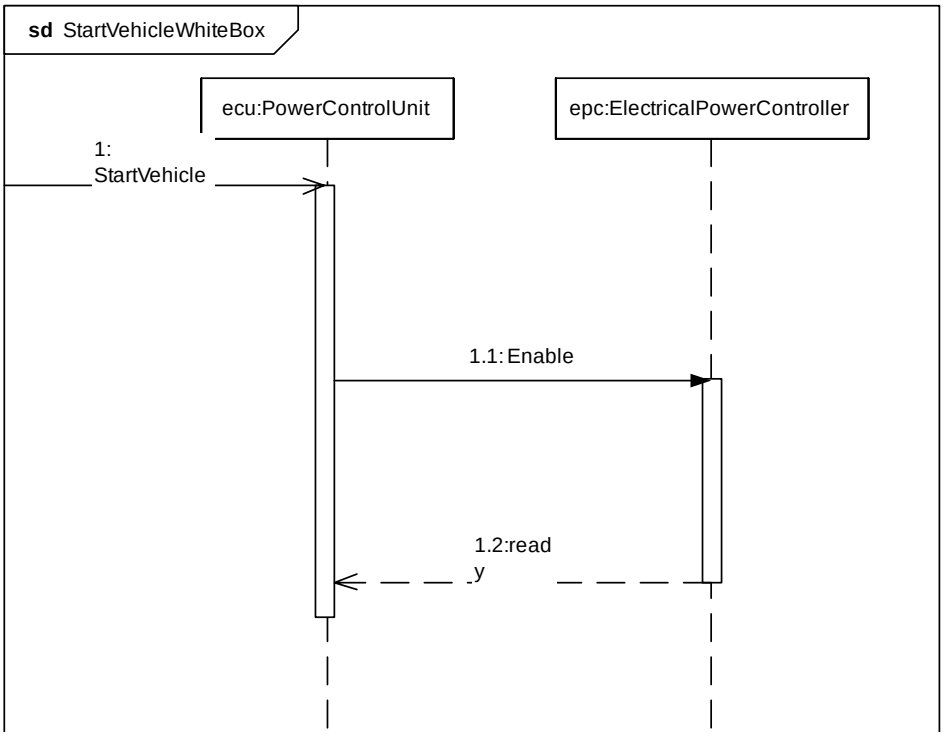


Figure 12.3 - White box interaction for "starting the Hybrid SUV"

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.

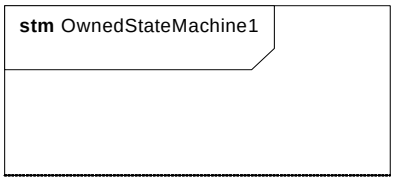
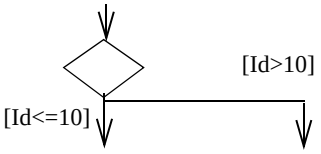
The UML concept of protocol state machines is excluded from SysML to reduce the complexity of the language. The standard UML state machine concept (called behavior state machines in UML) are thought to be sufficient for expressing protocols.

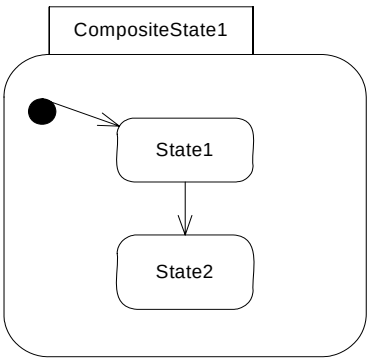
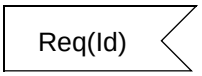
13.2 Diagram Elements

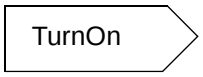
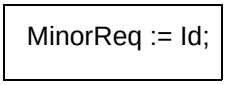
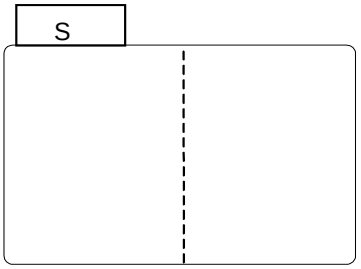
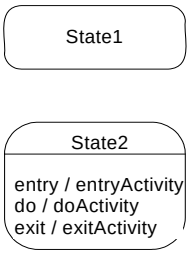
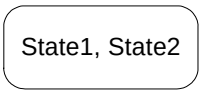
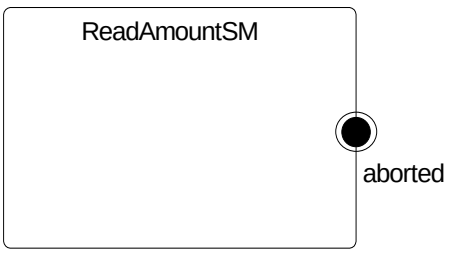
13.2.1 State Machine Diagram

No differences between SysML State Machine Diagrams and UML 2.1 State Machine Diagrams.

Table 13.1 - Graphical nodes included in state machine diagrams.

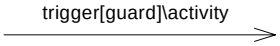
Node Name	Concrete Syntax	Abstract Syntax Reference
StateMachineDiagram		**UML4SysML::StateMachines**
Choice pseudo state		**UML4SysML::PseudoState**

Node 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**

Node Name	Concrete Syntax	Abstract Syntax Reference
Send signal action		**UML4SysML::Transition**
Action		**UML4SysML::Transition**
Region		**UML4SysML::Region**
Simple state		**UML4SysML::State**
State list		**UML4SysML::State**
State Machine		**UML4SysML::StateMachine**

Node Name	Concrete Syntax	Abstract Syntax Reference
Terminate node		**UML4SysML::PseudoState**
Submachine state		**UML4SysML::State**

Table 13.2 - Graphical paths included in state machine diagrams

Path Name	Concrete Syntax	Abstract Syntax Reference
Transition		**UML4SysML::Transition**

13.3 UML Extensions

None.

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 13.1.

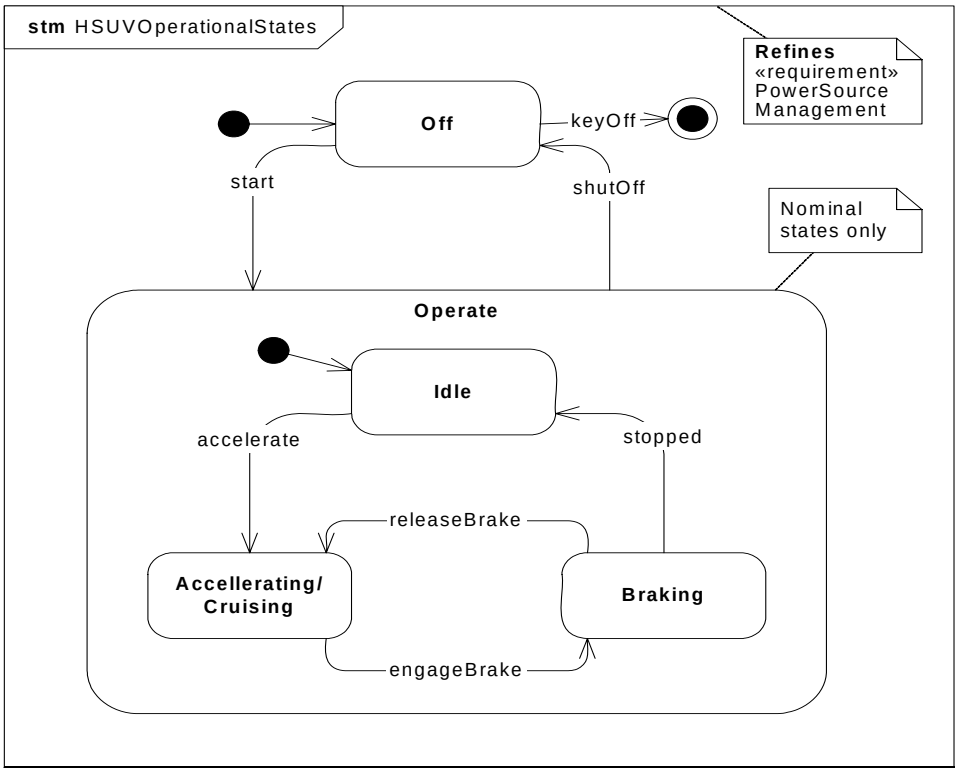


Figure 13.1 - High level view of the states of the HybridSUV

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 occurs 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.

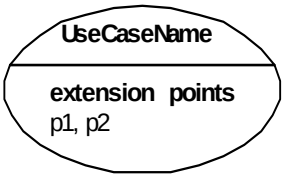
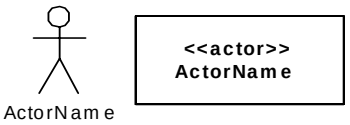
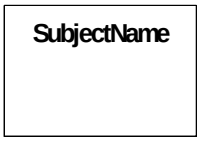
The use case relationships are “communication,” "include," "extend," and "generalization." Actors are connected to use cases via communication paths, that are represented by an association relationship. The "include" relationship provides a mechanism for factoring out common functionality which is shared among multiple use cases, and is always performed as part of the base use case. The "extend" relationship provides optional functionality, 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

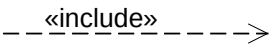
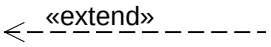
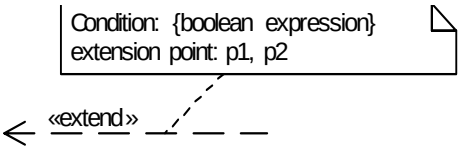
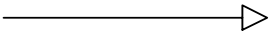
Node Name	Concrete Syntax	Abstract Syntax Reference
Use Case		**UML4SysML::UseCase**
Use Case with ExtensionPoints		UML4SysML::UseCase
Actor		**UML4SysML::Actor**
Subject		**Role name on Classifier**

.

Table 14.2 - Graphical paths included in Use Case diagrams

Path Type	concrete Syntax	Abstract Syntax Reference
Communication path		**UML4SysML::Association**

Table 14.2 - Graphical paths included in Use Case diagrams

Path Type	concrete Syntax	Abstract Syntax Reference
Include		**Subclass of UML4SysML::Directed Relationship**
Extend		**Subclass of UML4SysML::Directed Relationship**
Extend with Condition		**Subclass of UML4SysML::Directed Relationship**
Generalization		**UML4SysML::Kernel**

14.3 UML Extensions

There are no SysML extensions to UML 2.1 use cases.

14.4 Usage Examples

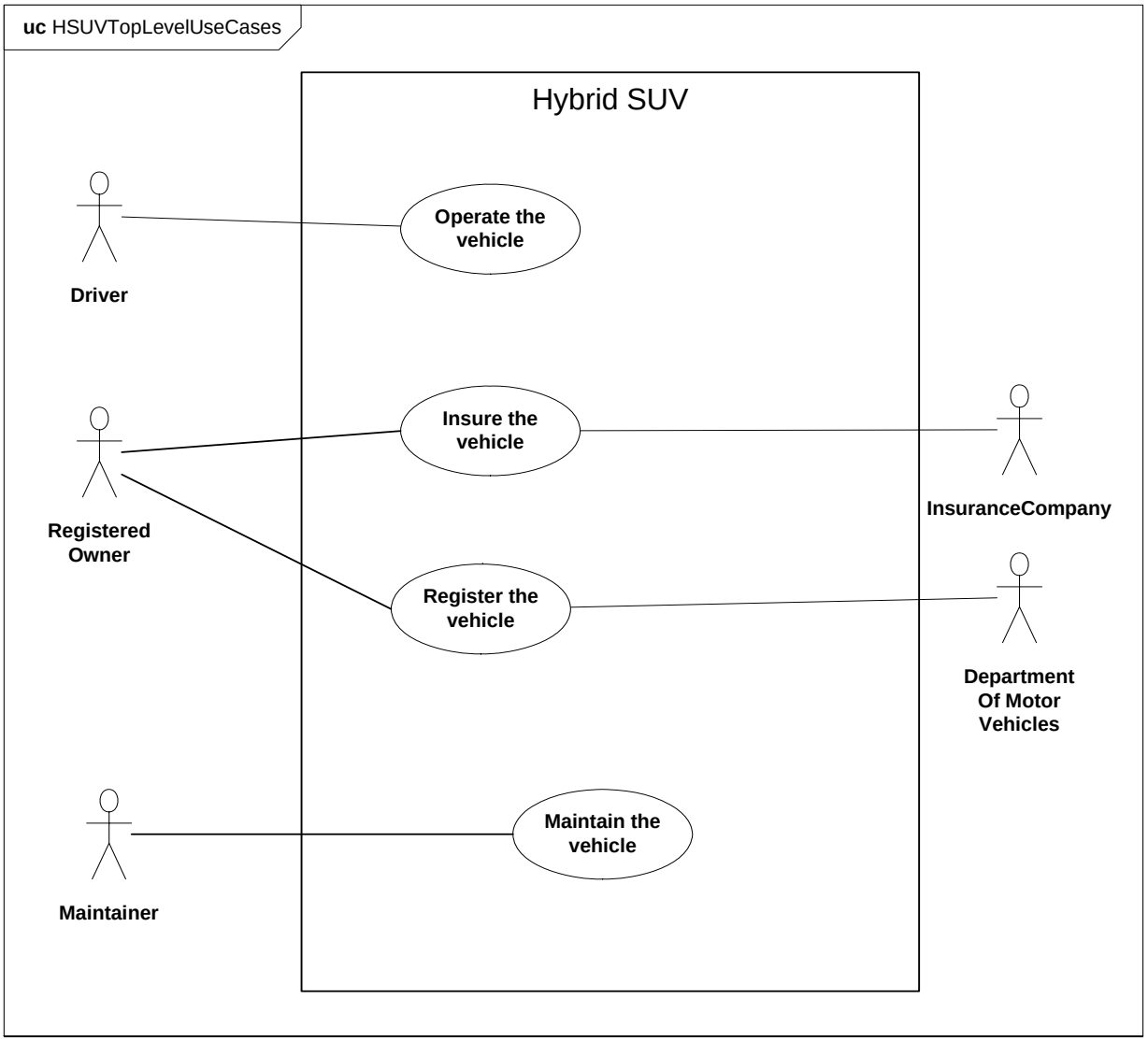


Figure 14.1 - Top level use case diagram for the Hybrid SUV subject

Figure 14.1 is a top-level set of use cases for the Hybrid SUV System. Figure 14.2 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.

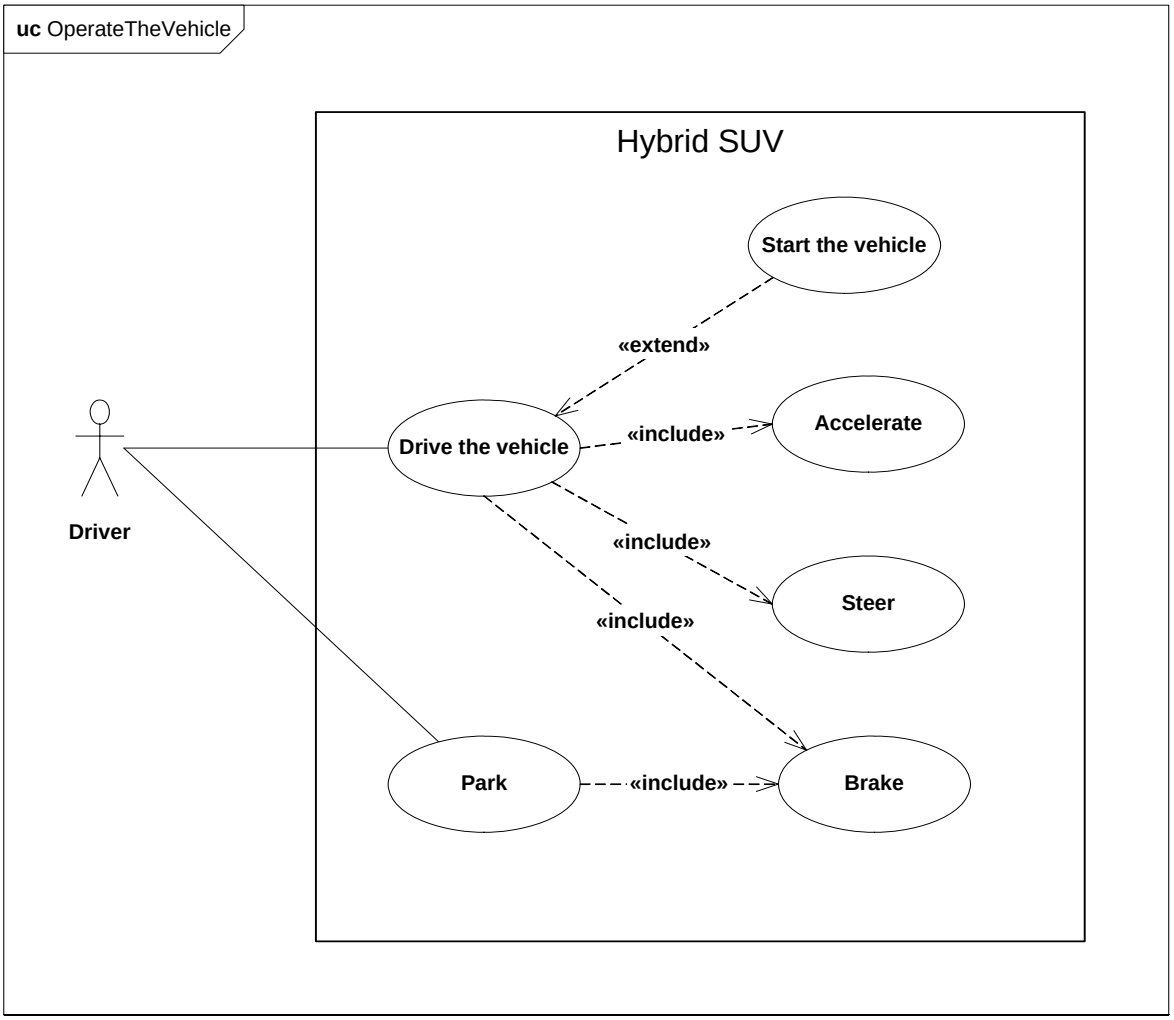


Figure 14.2 - Operate the Vehicle use case at a lower level of abstraction

Part IV - Crosscutting Constructs

This Part specifies cross-cutting constructs that apply to both structure and behavior. These constructs are defined in Chapter 15, “Allocations,” Chapter 16, “Requirements,” and Chapter 17, “Profiles & Model Libraries.” The Allocations chapter defines a basic allocation relationship that can be used to allocate a set of model elements to another, such as allocating behavior to structure or allocating logical to physical components. The Requirements chapter specifies constructs for system requirements and their relationships. The Profiles and Model Libraries chapter specifies the approach to further customize and extend SysML for specific applications.

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 chapter 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 chapter 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

15.2.1 Representing Allocation on Diagrams

Table 15.1 - Extension to graphical nodes included in diagrams

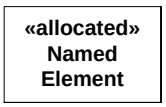
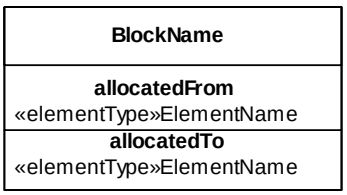
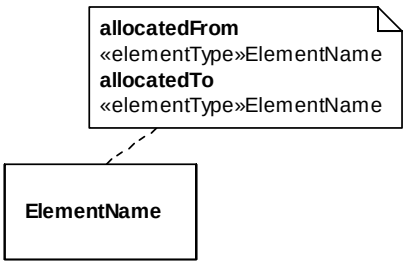
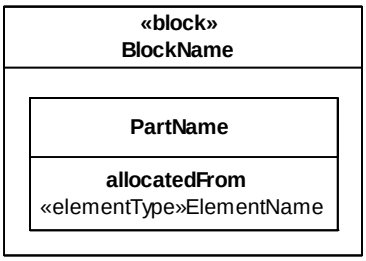
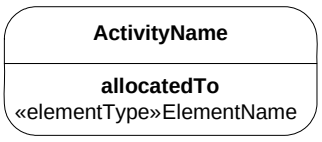
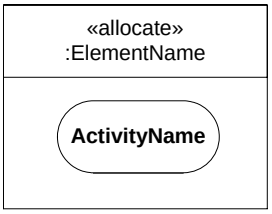
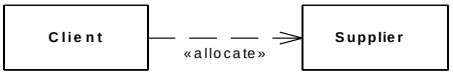
Node Name	Concrete Syntax	Abstract Syntax Reference
Allocated stereotype		**SysML::Allocation:Allocated**
Allocation derived properties displayed in compartment of a Block.		**SysML::Allocation:Allocated**
Allocation derived properties displayed in Comment.		**SysML::Allocation:Allocated**
Allocation derived properties displayed in compartment of Part on Internal Block Diagram.		**SysML::Allocation:Allocated**
Allocation derived properties displayed in compartment of Action on Activity Diagram.		**SysML::Allocation:Allocated**

Table 15.1 - Extension to graphical nodes included in diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Allocation Activity Partition		**SysML::Allocation:Allocate ActivityPartition**
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 Allocated Property Compartment Format

When properties of an «allocated» model element are displayed in a property compartment, a shorthand notation is used as shown in Table 15.1. This shorthand groups and displays the AllocatedFrom properties together, then the AllocatedTo properties. These properties are shown without the use of brackets {}.

15.3.1.4 Allocated Property Callout Format

When an «allocate» property component is not used, a property callout may be used. An «allocate» property callout uses the same shorthand notation as the «allocate» property compartment. This notation is also shown in Table 15.1. For brevity, the «elementType» portion of the AllocatedFrom or AllocatedTo property 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

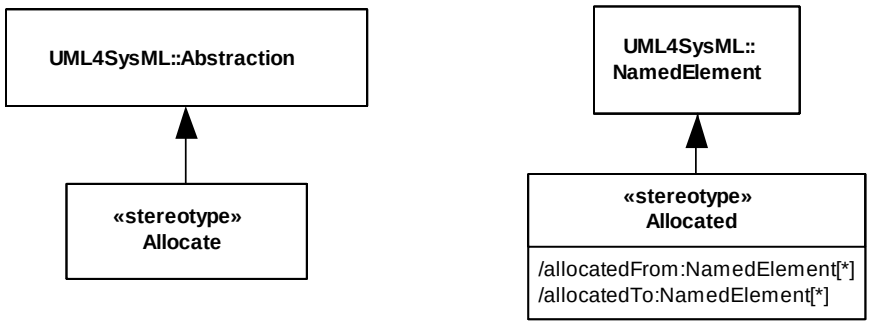


Figure 15.1 - Abstract syntax extensions for SysML Allocation

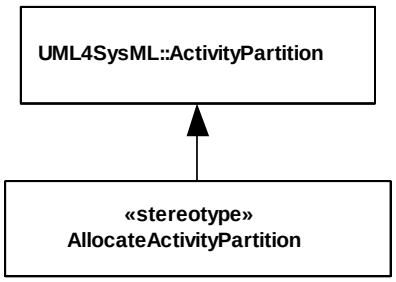


Figure 15.2 - Abstract syntax expression for AllocatedActivityPartition

15.3.2.1 Allocate(from Allocations)

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 which 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 at least one NamedElement is the “to” end (the end with the arrow).

The following paragraphs describe types of allocation that are typical in system 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, nor 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 InformationFlow.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 Chapter 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 must 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 specification will not define “logical” or “physical” in this context, except to acknowledge the stated need to capture allocation relationships between separate system representations.

Constraints

A single «allocate» dependency shall have only one supplier (from), but may have one or many clients (to).

If subtypes of the «allocate» dependency are introduced to represent more specialized forms of allocation then they should have constraints applied to supplier and client as appropriate.

15.3.2.2 Allocated(from Allocations)

Description

«allocated» is a stereotype that applies to any NamedElement that has at least one allocation relationship with another NamedElement. «allocated» elements may be designated by either the /from or /to end of an «allocate» dependency.

The «allocated» stereotype provides a mechanism for a particular model element to conveniently retain and display the element at the opposite end of any «allocate» dependency. This stereotype provides for the properties “allocatedFrom” and “allocatedTo,” which are derived from the «allocate» dependency.

Attributes

The following properties are derived from any «allocate» dependency:

- /allocatedTo:NamedElement[*]

  The element types and names of the set of elements that are clients (“to” end of the concrete syntax) of an «allocate» whose client is extended by this stereotype (instance). This property is the union of all clients to which this instance is the supplier, i.e., there may be more than one /allocatedTo property per allocated model element. Each allocatedTo property will be expressed as «elementType» ElementName.

- /allocatedFrom:NamedElement[*]

  Reverse of allocatedTo: the element types and names of the set of elements that are suppliers (from) of an «allocate» whose supplier is extended by this stereotype (instance). The same characteristics apply as to /allocatedTo. Each allocatedFrom property will be expressed as «elementType» ElementName.

For uniformity, the «elementType» displayed for the /allocatedTo or /allocatedFrom properties should be from the following list, as applicable. Other «elementType» designations may be used, if none of the below apply.

«activity», «objectFlow», «controlFlow», «objectNode»

«block», «itemFlow», «connector», «port», «flowPort», «atomicFlowPort», «interface», «value»

Note that the supplier or client may be an Element (e.g., Activity, Block), Property (e.g., Action, Part), Connector, or BehavioralFeature (e.g., Operation). For this reason, it is important to use fully qualified names when displaying / allocatedFrom and /allocatedTo properties. An example of a fully qualified name is the form (PackageName::ElementName.PropertyName). Use of such fully qualified makes it clear that the «allocate» is referring to the definition of the element, or to it’s specific usage as a property of another element.

15.3.2.3 AllocateActivityPartition(from Allocations)

Description

AllocateActivityPartition is used to depict an <allocate> relationship on an Activity diagram. The AllocateActivityPartition is a standard UML2::ActivityPartition, with modified constraints such that any Actions within the partition must result in an <allocate> dependency between the Activity used by the Action, and the element that the partition represents.

Constraints

An Action appearing in an <AllocateActivityPartition> will be the /supplier (from) end of an <allocate> dependency. The element represented by the <AllocateActivityPartition> will be the /client (to) end of the same <allocate> dependency.

The «AllocateActivityPartition» maintains the constraints, but not the semantics, of the UML2::ActivityPartition. Classifiers, Instances, or Parts represented by an «AllocateActivityPartition» do not have any direct responsibility for invoking behavior depicted within the partition boundaries.

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.

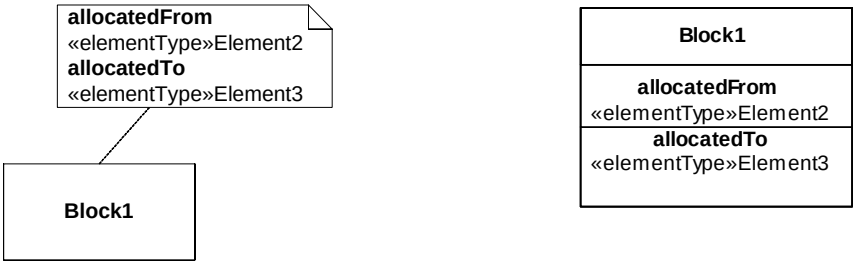


Figure 15.3 - Generic Allocation, including /from and /to association ends

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.

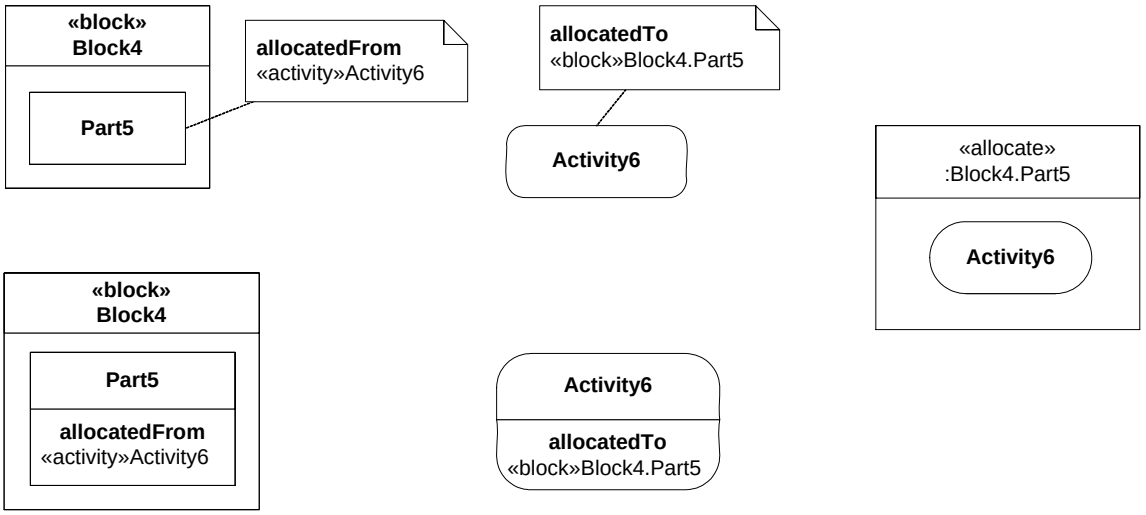


Figure 15.4 - Behavior allocation

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. Independent of the ObjectFlow allocation, it may be valuable to allocate the corresponding ObjectNode to an ItemProperty associated with the ItemFlow.

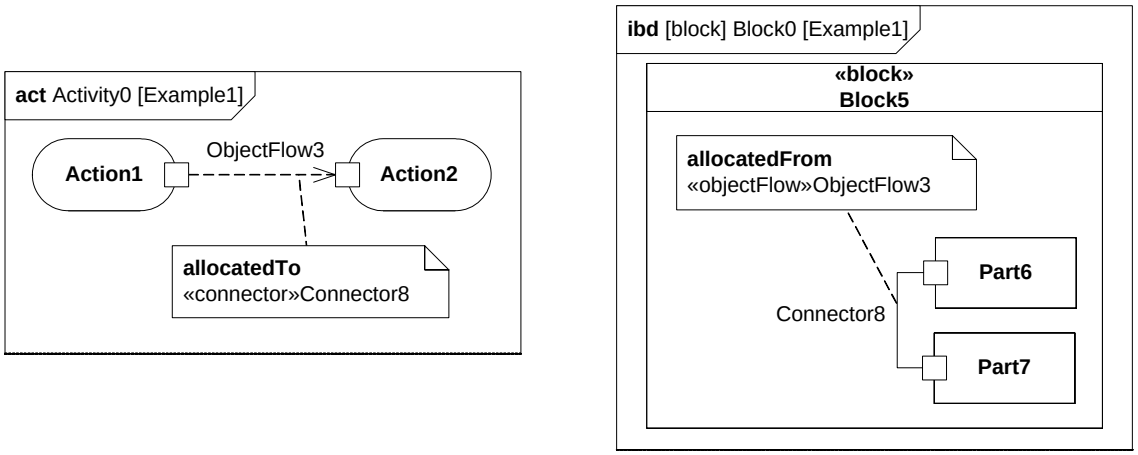


Figure 15.5 - Example of flow allocation from ObjectFlow to Connector

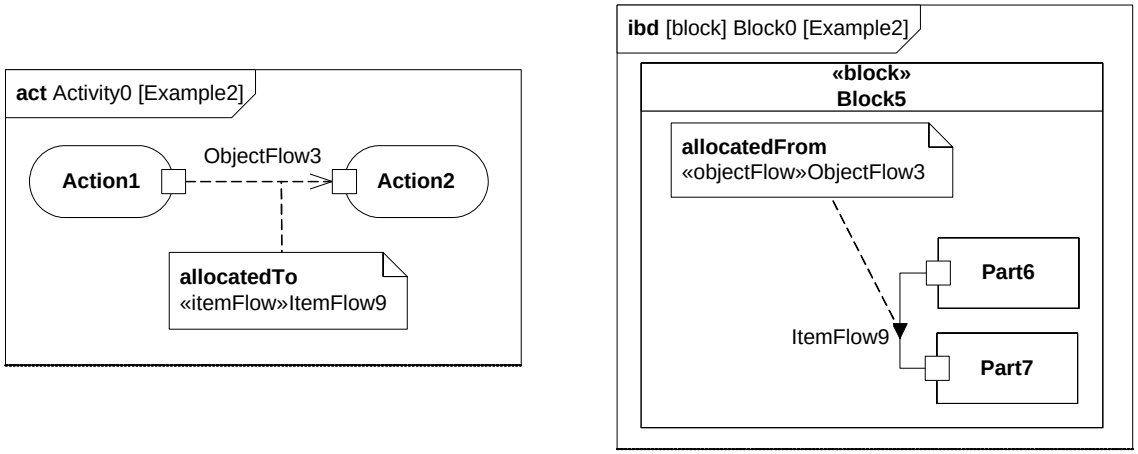


Figure 15.6 - Example of flow allocation from ObjectFlow to ItemFlow

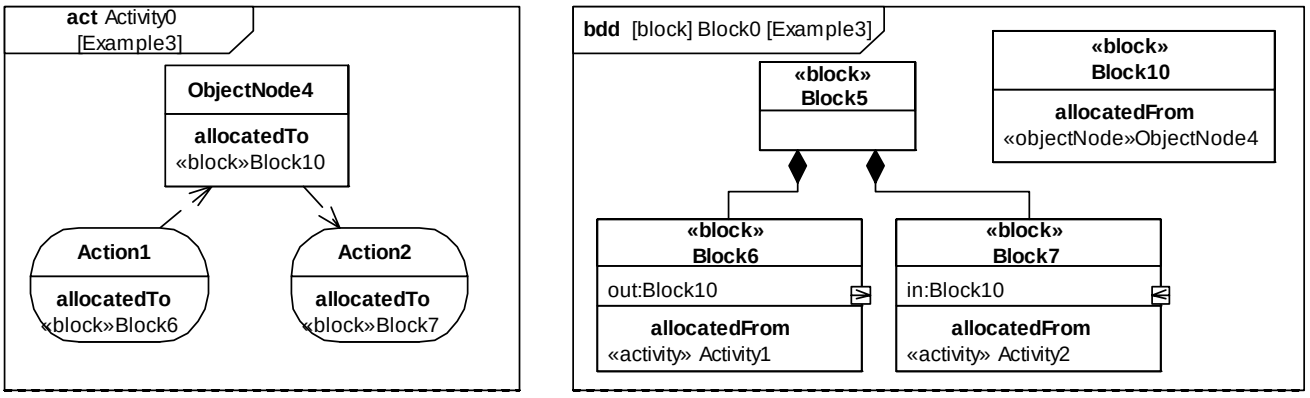


Figure 15.7 - Example of flow allocation from ObjectNode to FlowProperty

15.4.2.1 Allocating Structure

Systems engineers have a 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).

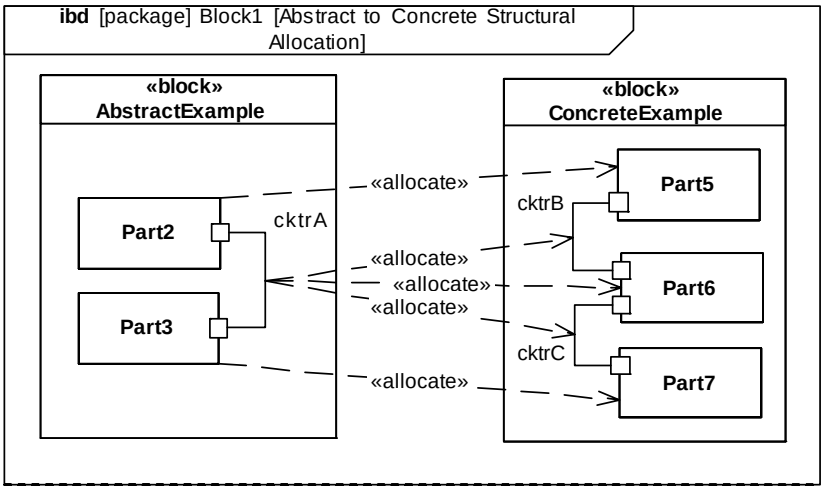


Figure 15.8 - Example of Structural Allocation

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 15.9. This example is consistent with Annex B: Sample Problem.

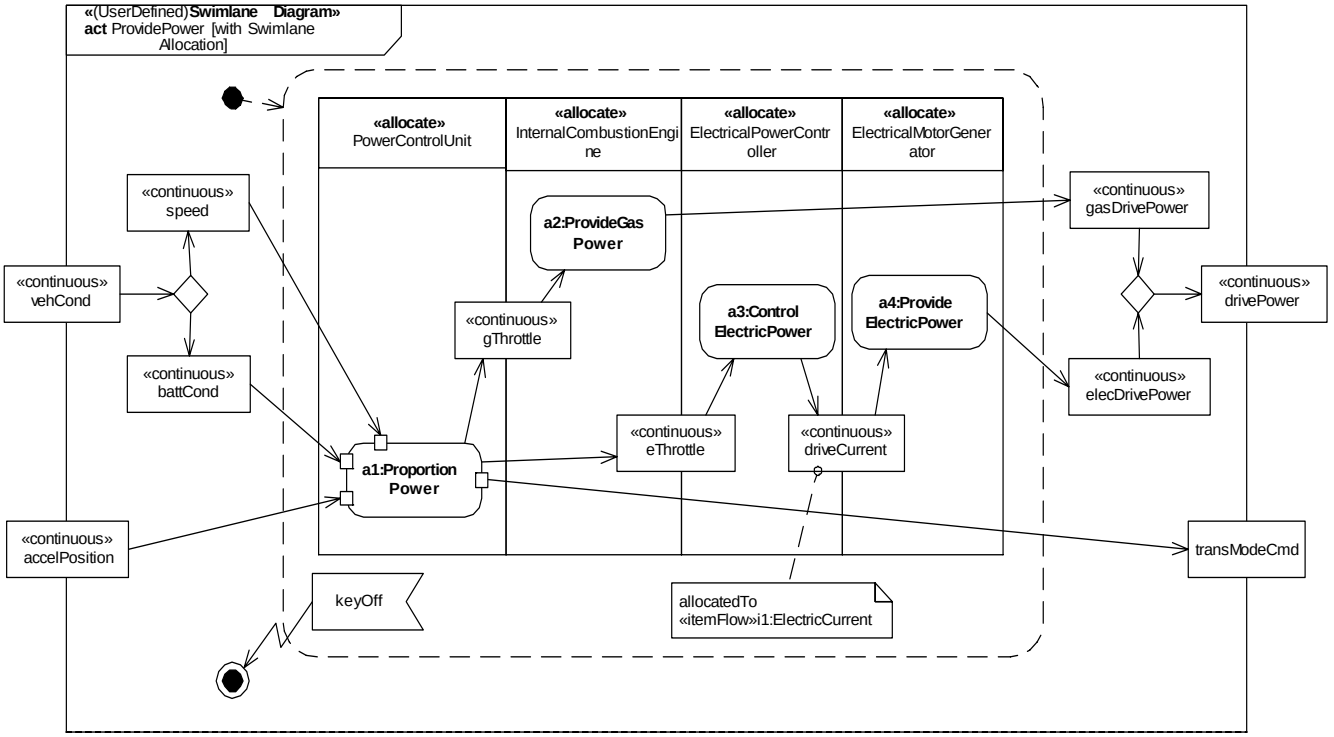


Figure 15.9 - AllocateActivityPartitions (Swimlanes) for HybridSUV Cellarette Example

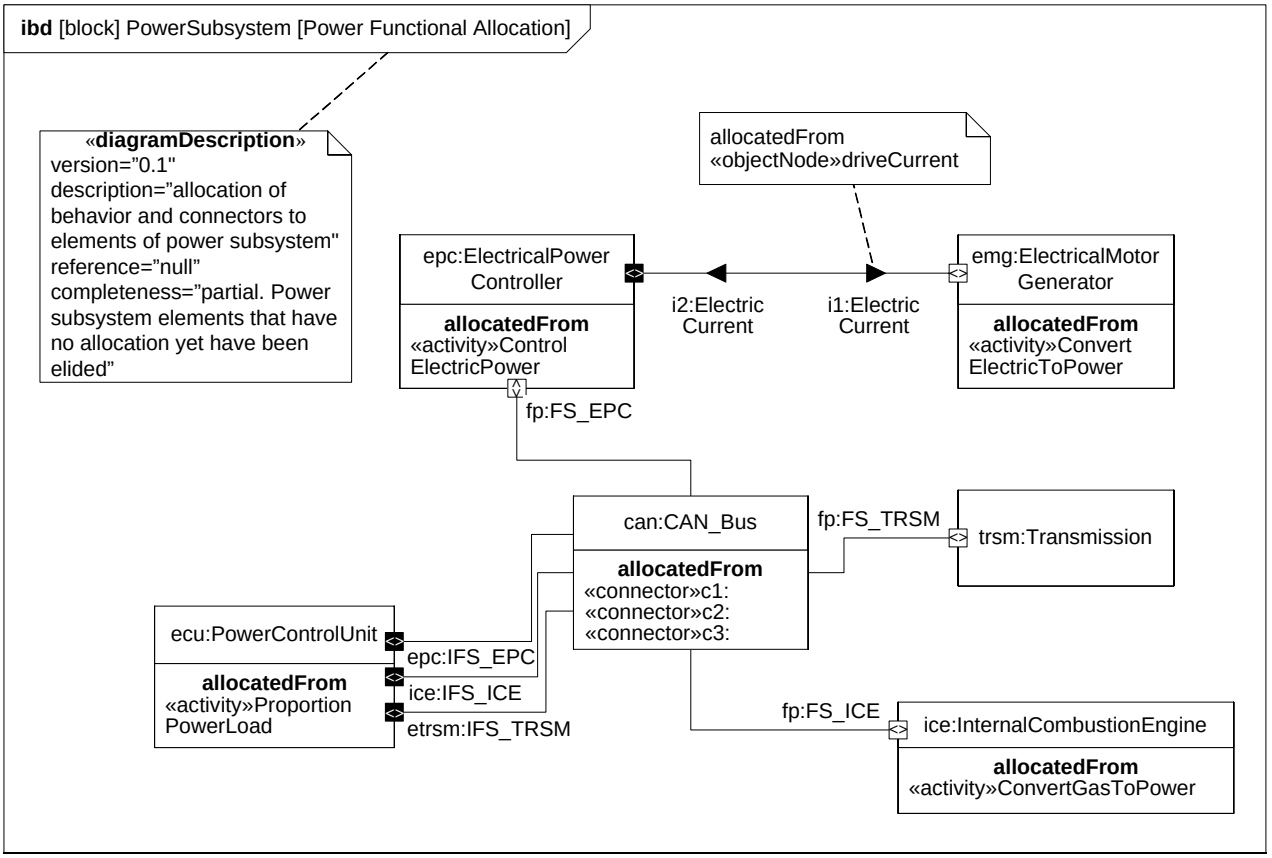


Figure 15.10 - Internal Block Diagram Showing Allocation for HybridSUV Accelerate Example

15.4.3 Tabular Representation

The table shown in Figure 15.11 is provided as a specific example of how the «allocate» dependency may be depicted in tabular form, consistent with the automotive example above.

table [activity] ProvidePower [Allocation Tree for Provide Power Activities]

type	name	end	relation	end	type	name
activity	a1:ProportionPower	from	allocate	to	block	PowerControlUnit
activity	a2:ProvideGasPower	from	allocate	to	block	InternalCombustionEngine
activity	a3:ControlElectricPower	from	allocate	to	block	ElectricalPowerController
activity	a4:ProvideElectricPower	from	allocate	to	block	ElectricalMotorGenerator
objectNode	driveCurrent	from	allocate	to	itemFlow	i1:ElectricCurrent

Figure 15.11 - Allocation Table (Tree) Showing Allocation for Hybrid SUV Cellarette Example

The allocation table can also be shown using a sparse matrix style as in the following example shown in Figure 15.12 :

matrix [activity] ProvidePower [Allocation Tree for Provide Power Activities]

Source	Target				
	PowerControlUnit	InternalCombustionEngine	ElectricalPowerController	ElectricalMotorGenerator	I1:ElectricCurrent
A1:ProportionPower	allocate				
A2:ProvideGasPower		allocate			
A3:ControlElectricPower			allocate		
A4:ProvideElectriPower				allocate	
driveCurrent					allocate

Figure 15.12 - Allocation Matrix Showing Allocation for Hybrid SUV Cellarette 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 chapter 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 re-use 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 re-used 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 re-used requirement(s).

The use of namespace containment to specify requirements hierarchies precludes re-using 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 verifies a requirement. In SysML, a test case is intended to 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 Chapter 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 C: Non-normative Extensions.

16.2 Diagram Elements

16.2.1 Requirements Diagrams

Table 16.1 - Graphical nodes included in Requirement diagrams

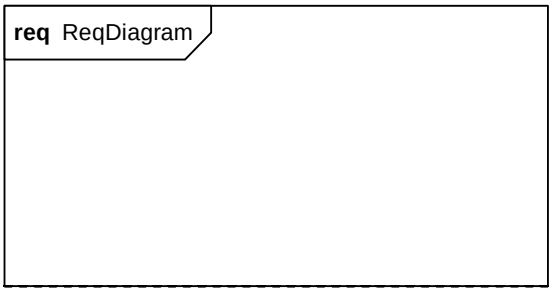
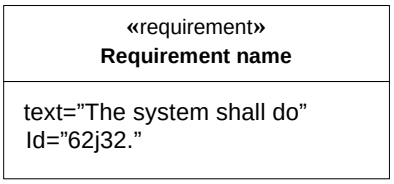
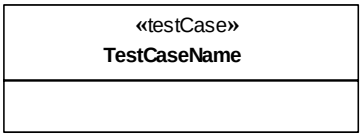
Node Name	Concrete Syntax	Abstract Syntax Reference
Requirement Diagram		**SysML::Requirements:: Requirement, SysML:: ModelElements::Package**
Requirement		**SysML::Requirements:: Requirement**
TestCase		**SysML::Requirements:: TestCase**

Table 16.2 - Graphical paths included in Requirement diagrams

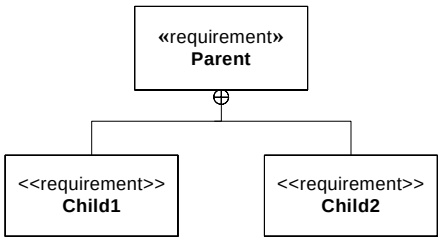
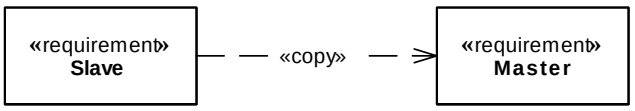
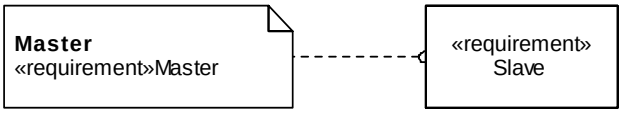
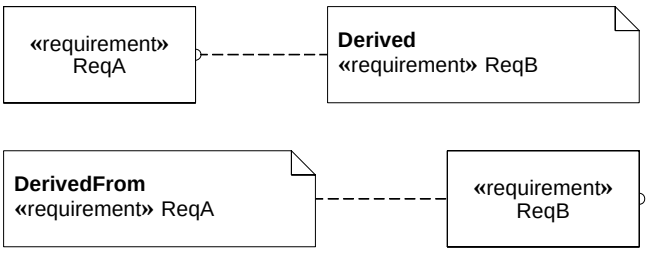
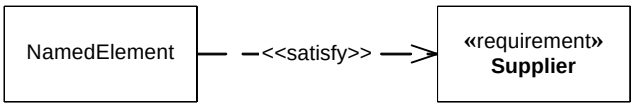
Path Type	Concrete Syntax	Abstract Syntax Reference
Requirement containment relationship		**UML4SysML:: NestedClassifier**
CopyDependency		**SysML::Requirments:: Copy**
MasterCallout		**SysML::Requirments:: Copy**
Derive Dependency		**SysML::Requirments:: DeriveReqt**
DeriveCallout		**SysML::Requirments:: DeriveReqt**
Satisfy Dependency		**SysML::Requirements:: Satisfy**

Table 16.2 - Graphical paths included in Requirement diagrams

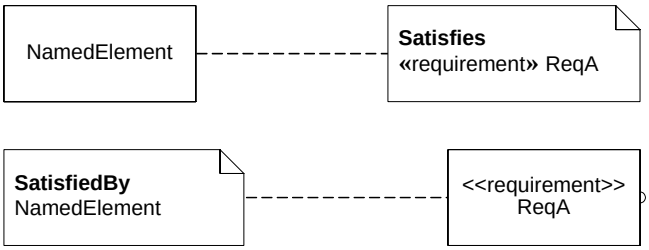
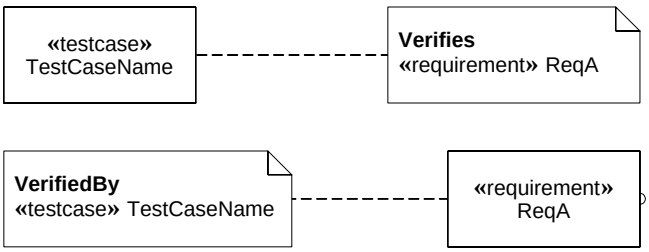
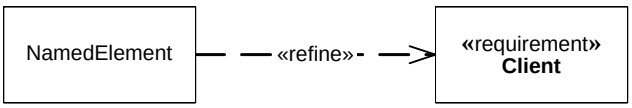
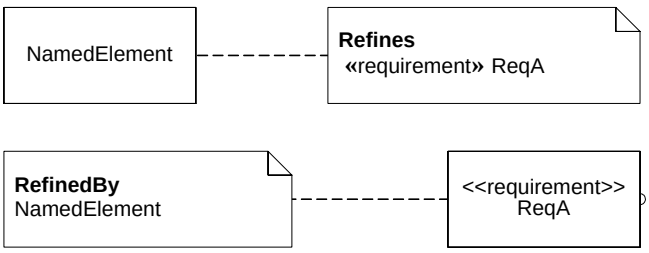
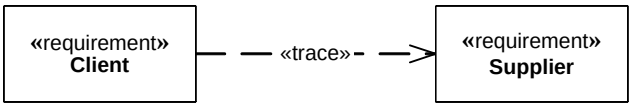
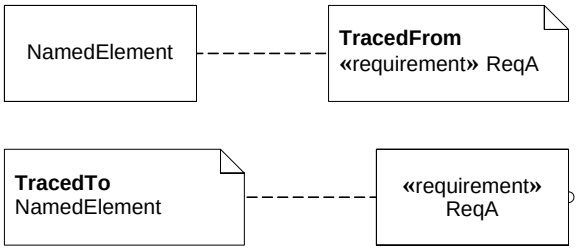
Path Type	Concrete Syntax	Abstract Syntax Reference
SatisfyCallout		**SysML::Requirements::** **Satisfy**
Verify Dependency		**SysML::Requirements::** **Verify**
VerifyCallout		**SysML::Requirements::** **Verify**
Refine Dependency		**UML4SysML::Refine**
RefineCallout		**UML4SysML::Refine**
Trace Dependency		**UML4SysML::Trace**

Table 16.2 - Graphical paths included in Requirement diagrams

Path Type	Concrete Syntax	Abstract Syntax Reference
TraceCallout		**UML4SysML::Trace**

16.3 UML Extensions

16.3.1 Diagram Extensions

16.3.1.1 Requirement Diagram

The Requirements Diagram can only display requirements, packages, other classifiers, test cases, and rationale. The relationships for containment, deriveReqt, satsify, 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 and 16.3.1.3 can be used. The callouts represents 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 (Section 15.4.3 (“Tabular Representation”)). The table should include the source and target elements names (and optionally kinds) and the requirement dependency kind..

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

16.3.2 Stereotypes

Package Requirements

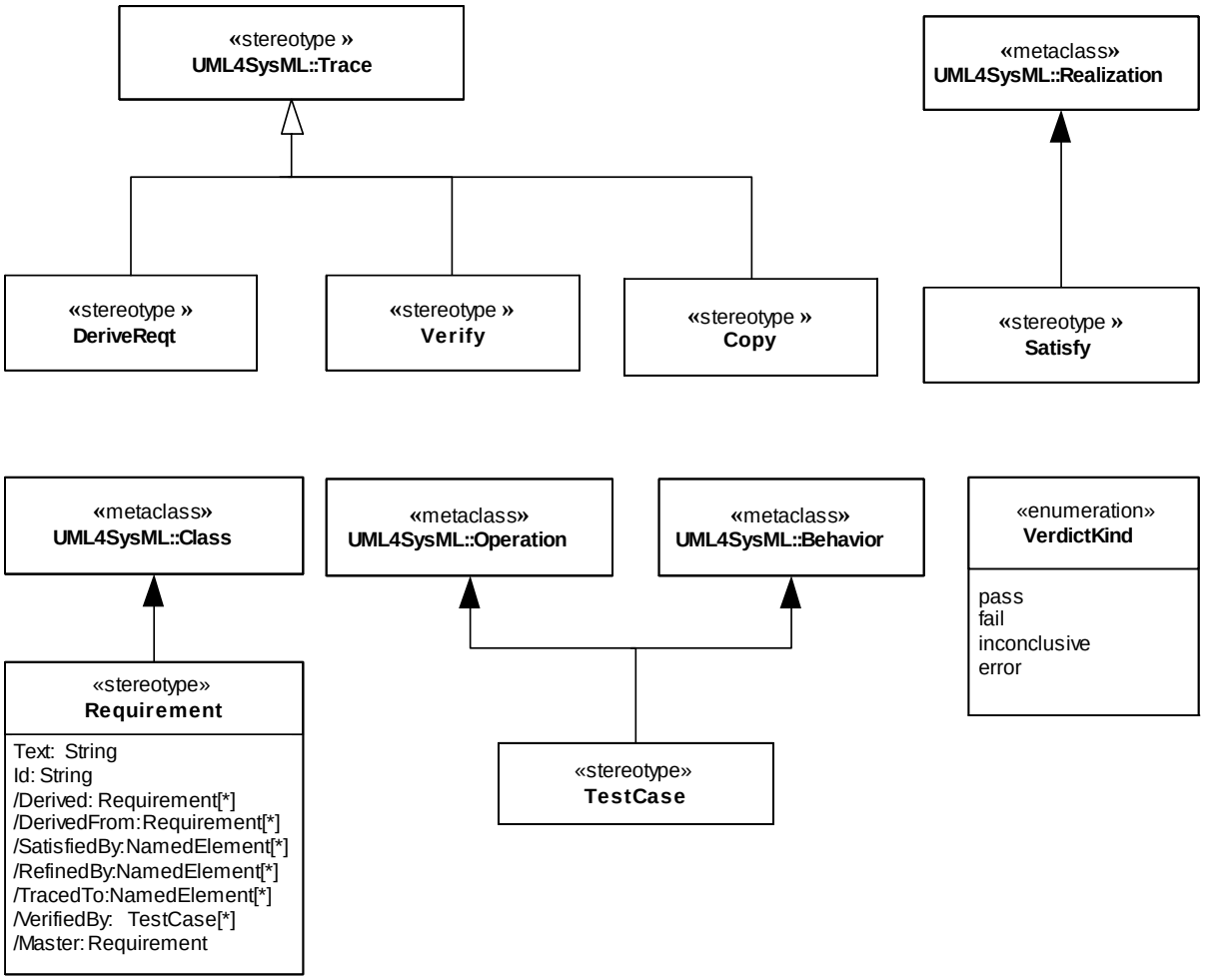


Figure 16.1 - Abstract Syntax for Requirements Stereotypes

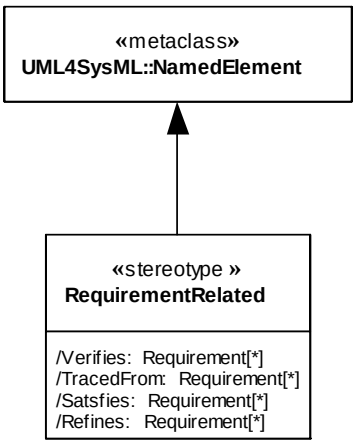


Figure 16.2 - Abstract Syntax for Requirements Stereotypes (cont)

16.3.2.1 Copy (from Requirements)

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 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.

Constraints

[1] A Copy dependency may only be created between two classes that have the "requirement" stereotype, or a sub-type of the "requirement" stereotype applied.

[2] If the supplier requirement has sub-requirements, copies of the sub-requirements are made recursively in the context of the client requirement and Copy dependencies are created between each sub-requirement and the associated copy.

[3] The text property of the client requirement is constrained to be a read only copy of the text property of the supplier requirement.

[4] Constraint [3] is applied recursively to all sub-requirements.

16.3.2.2 DeriveReqt (from Requirements)

Description

A dependency relationship 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.

Constraints

[1] The supplier must be an element stereotyped by «requirement» or one of «requirement» subtypes.

[2] The client must be an element stereotyped by «requirement» or one of «requirement» subtypes.

16.3.2.3 Requirement (from Requirements)

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 Class. 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 must be satisfied for the compound requirement to be satisfied. Subrequirements can 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.

Attributes

- text: String
  The textual representation or a reference to the textual representation of the requirement.
- id: String
  The unique id of the requirement.
- /satisfiedBy: NamedElement[*]
  Derived from all elements that are the client of a <<satisfy>> relationship for which this requirement is a supplier.
- /verifiedBy: NamedElement[*]
  Derived from all elements that are the client of a <<verify>> relationship for which this requirement is a supplier.
- /tracedTo: NamedElement[*]
  Derived from all elements that are the client of a <<trace>> relationship for which this requirement is a supplier.
- /derived: Requirement[0..1]
  Derived from all requirements that are the client of a <<deriveReqt>> relationship for which this requirement is a supplier.
- /derivedFrom: Requirement[*]
  Derived from all requirements that are the supplier of a <<deriveReqt>> relationship for which this requirement is a client.
- /refinedBy: NamedElement[*]
  Derived from all elements that are the client of a <<refine>> relationship for which this requirement is a supplier.
- /master: Requirement[0..1
  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.

Constraints

[1] The property *isAbstract* must be set to *true*.

[2] The property *ownedOperation* must be empty.

[3] The property *ownedAttribute* must be empty.

[4] Classes stereotyped by «requirement» may not participate in associations.

[5] Classes stereotyped by «requirement» may not participate in generalizations.

[6] A nested classifier of a class stereotyped by «requirement» must also be stereotyped by «requirement».

16.3.2.4 RequirementRelated (from Requirements)

Description

This stereotype is used to add properties to those elements that are related to requirements via the various dependencies described in Figure 16.1. The property values are shown using call-out notation (i.e., notes) as shown in the diagram element table.

Attributes

- \verifies: Requirement[*]
  Derived from all requirements that are the supplier of a <<verify>> relationship for which this element is a client.
- \satisfies: Requirement[*]
  Derived from all requirements that are the supplier of a <<satisfy>> relationship for which this element is a client.
- \refines: Requirement[*]
  Derived from all requirements that are the supplier of a <<refine>> relationship for which this element is a client.
- \tracedFrom: Requirement[*]
  Derived from all requirements that are the supplier of a <<trace>> relationship for which this element is a client.

16.3.2.5 TestCase (from Requirements)

Description

A method for verifying a requirement is satisfied.

Constraints

[1] The type of return parameter of the stereotyped model element must be VerdictKind. (note this is consistent with the UML Testing Profile).

16.3.2.6 Satisfy (from Requirements)

Description

A dependency relationship 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.

Constraints

[1] The supplier must be an element stereotyped by «requirement» or one of «requirement» subtypes.

16.3.2.7 Verify (from Requirements)

Description

A relationship between a requirement and a test case that can determine whether a system fulfills the requirement. As with other dependencies, the arrow direction points from the (client) test case to the (supplier) requirement.

Constraints

[1] The supplier must be an element stereotyped by «requirement» or one of «requirement» subtypes

[2] The client must be an element stereotyped by «testCase» or one of the «testCase» subtypes.

16.4 Usage Examples

All the examples in this chapter are based on a set of publicly available (on-line) requirement specification from the *National Highway Traffic Safety Administration (NHTSA*.) Excerpts of the original requirement text used to create the models are shown in Figure 16.3. 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 16.3 shows an example of a compound requirement decomposed into multiple subrequirements.

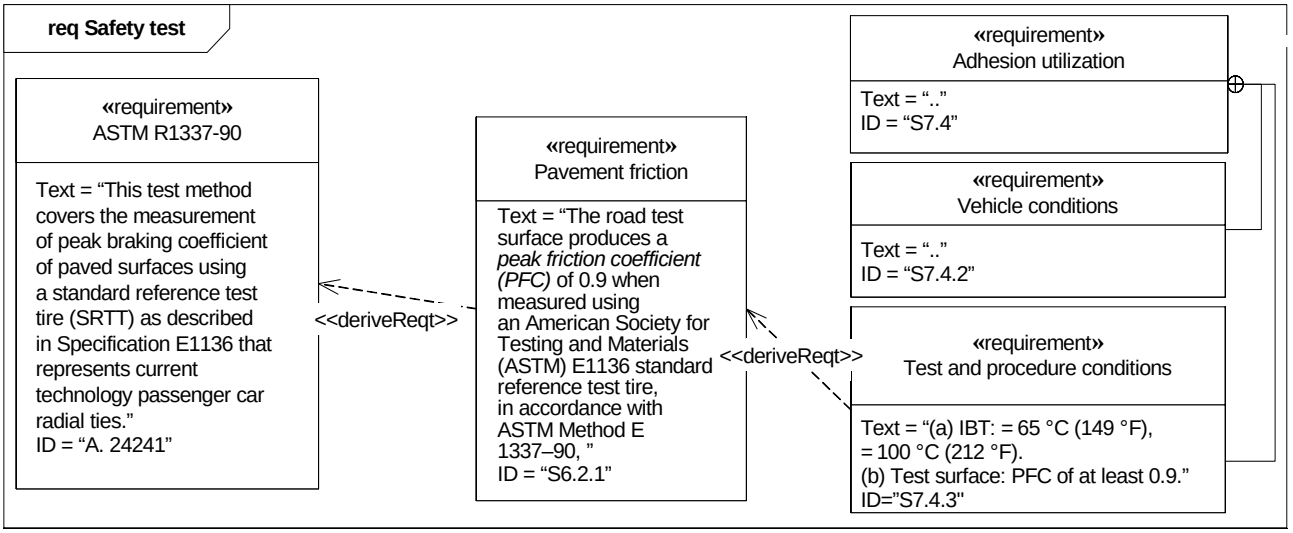


Figure 16.3 - Requirements Derivation

16.4.1.1 Requirements and Design Elements

The diagram in Figure 16.4 shows derived requirements and refers to the design elements that satisfy them. The rational is also shown as a basis for the design solution.

.

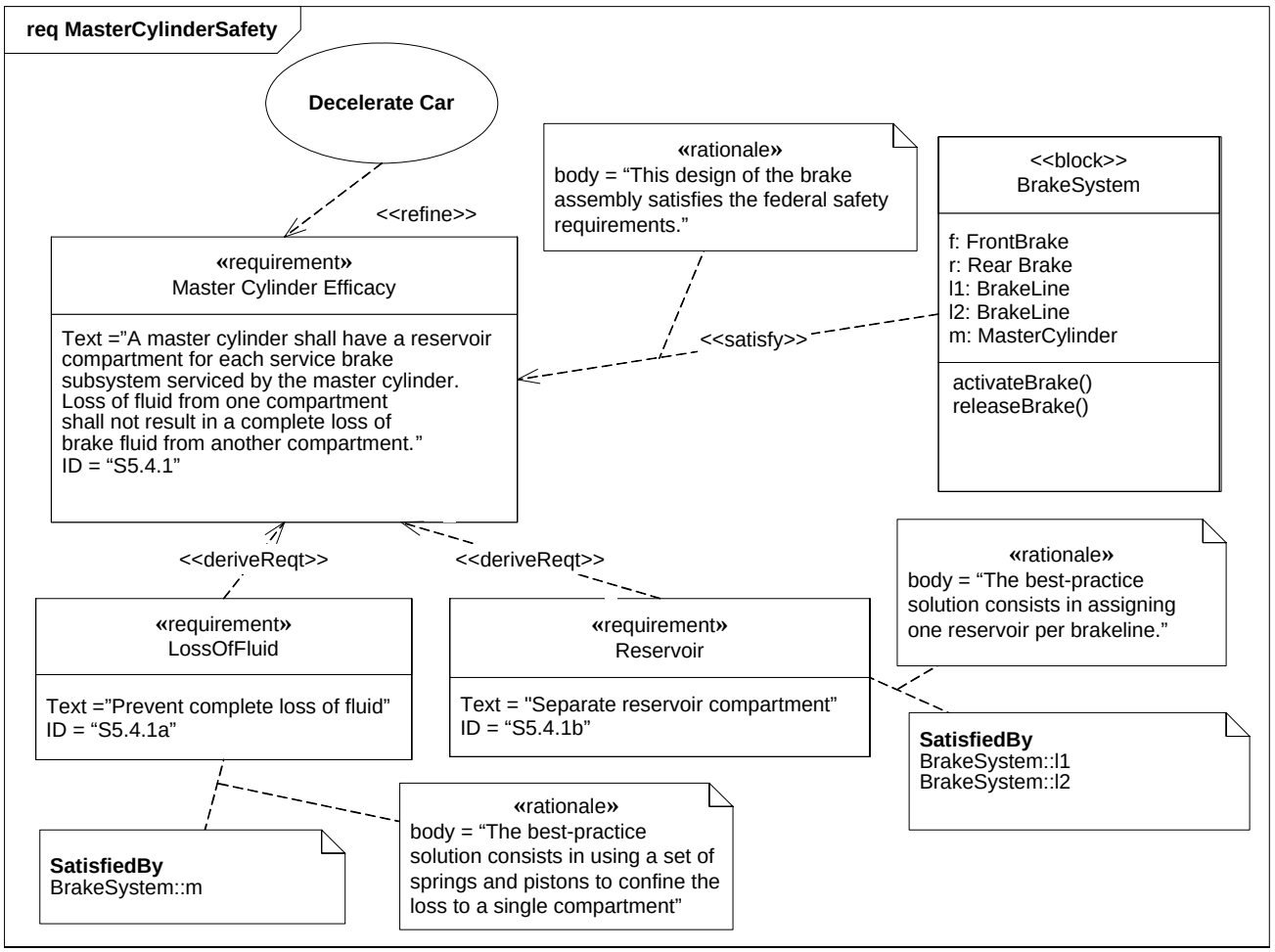


Figure 16.4 - Links between requirements and design

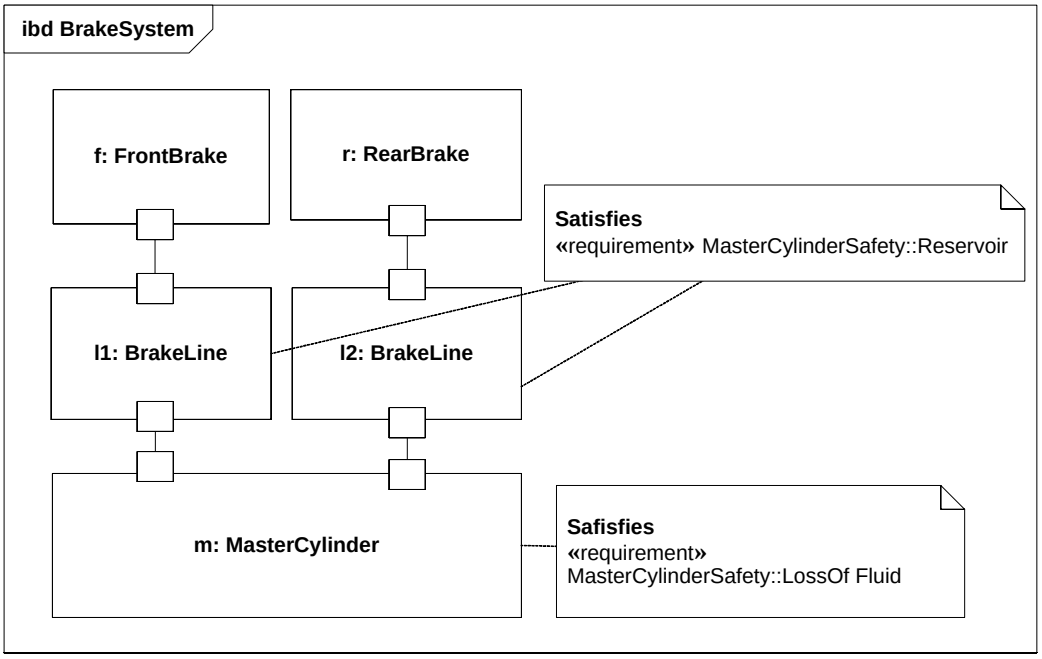


Figure 16.5 - Requirement satisfaction in an internal block diagram.

16.4.1.2 Requirements Reuse

Figure 16.6 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.

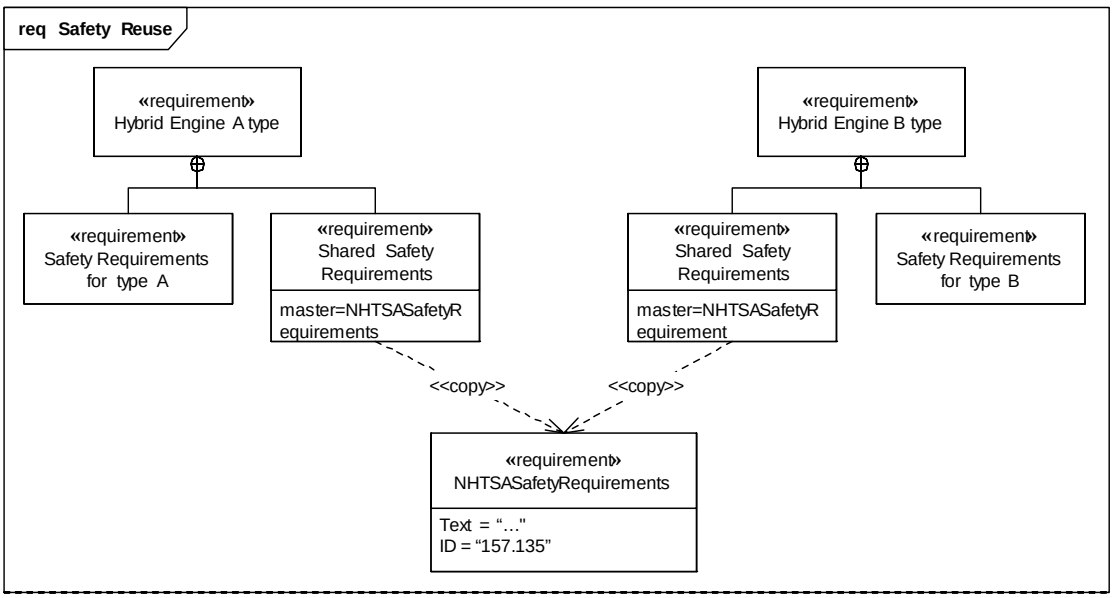


Figure 16.6 - Use of the copy dependency to facilitate reuse

16.4.1.3 Verification Procedure (Test Case)

The example diagram in Figure 16.7 shows how a complex test case, in this example a performance test for a passenger-car brake system, given as a set of steps in text form (see part of the procedure text at the upper right-handside corner of the figure), can be described using another type of diagram representation. The performance test, modeled as a Test Case is linked to a requirement using the «verify»» relationship. Note that the modeling of test case can also be addressed using the UML Testing Profile, available from the Object Management Group.

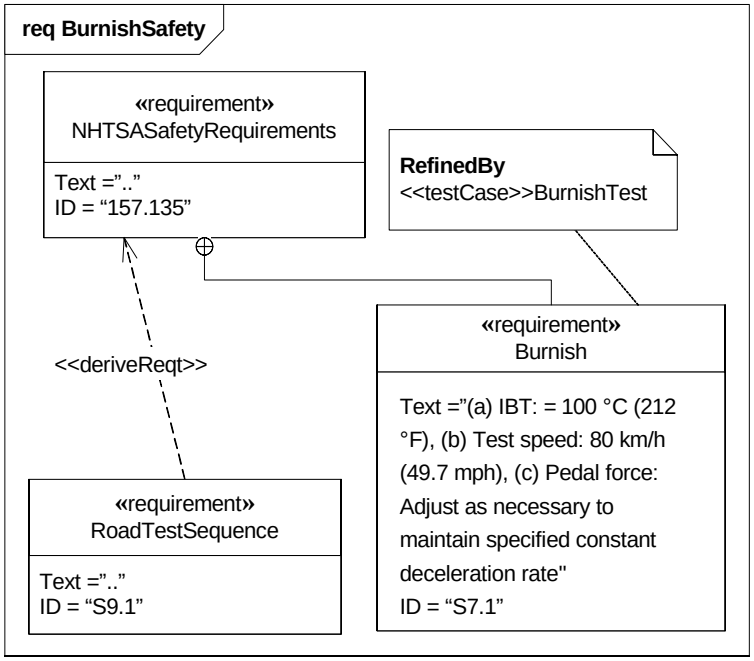


Figure 16.7 - Linkage of a Test Case to a requirement: This figure shows the Requirement Diagram.

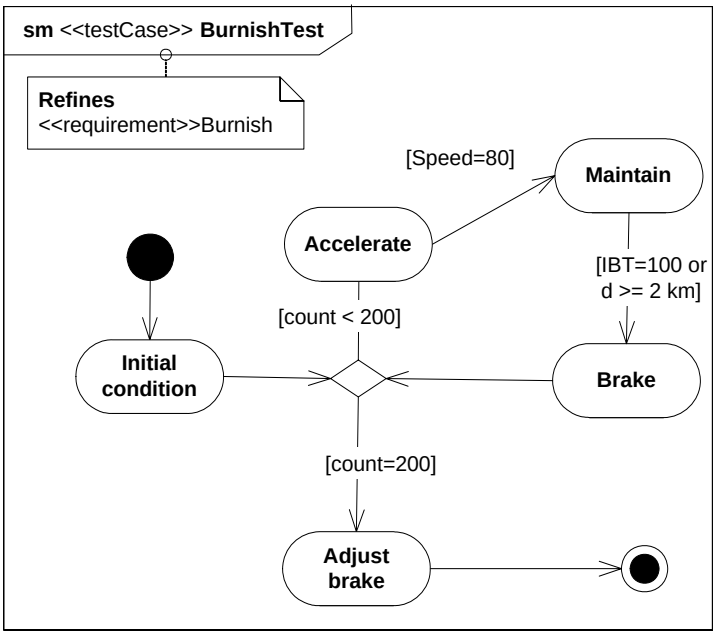


Figure 16.8 - Linkage of a Test Case to a requirement: This figure shows the Test Case as a State Diagram.

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 C: 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.

In addition to ex section 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. This guidelines can be applied to further customize SysML for domain specific applications such as automotive, military, or space systems.

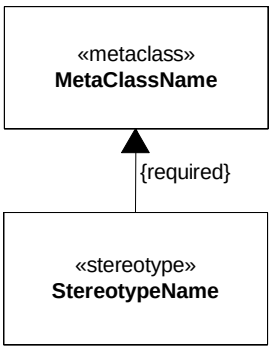
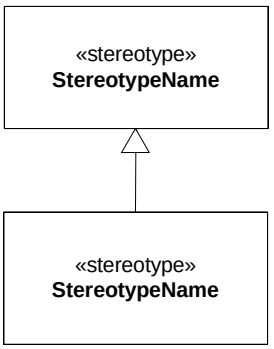
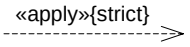
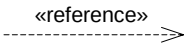
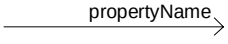
17.2 Diagram Elements

17.2.1 Profile Definition in Class Diagram

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**
Model Library	«modelLibrary» **LibraryName**	**UML::StandardProfileL1**

Table 17.2 - Graphical paths used in profile definition

Path Name	Concrete Syntax	Abstract Syntax Reference
Extension		**UML4SysML::Extension**
Generalization		**UML4SysML::Generalization**
ProfileApplication		**UML4SysML::ProfileApplication**
MetamodelReference		**UML4SysML::PackageImport;** **UML4SysML::ElementImport**
Unidirectional Association		**UML4SysML::Association**

NOTE: In the above table, boolean properties can alternatively be displayed 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.

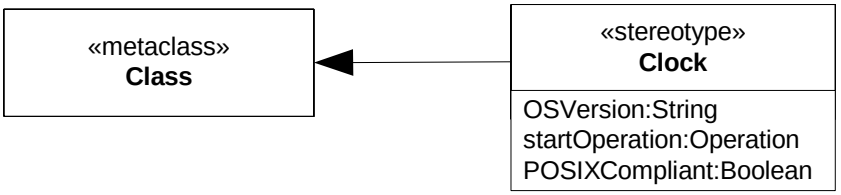


Figure 17.1 - Defining a stereotype

17.2.2 Stereotypes Used On Diagrams

Table 17.3 - Notations for Stereotype Use

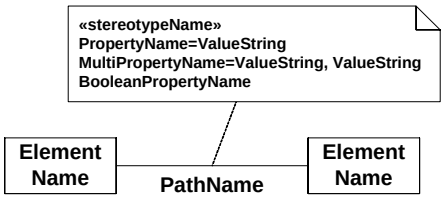
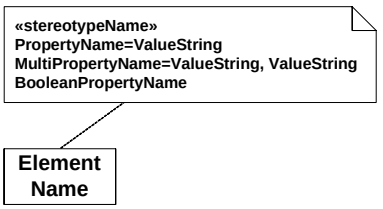
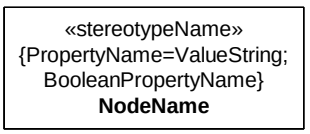
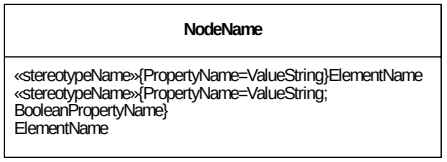
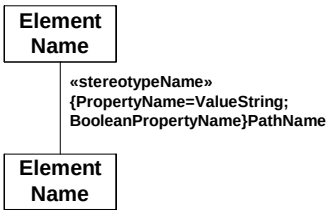
Node Name	Concrete Syntax	Abstract Syntax Reference
StereotypeNote		**UML4SysML::Element**
StereotypeNote		**UML4SysML::Element**
StereotypeInNode		**UML4SysML::Element**
StereotypeInCompartment Element		**UML4SysML::Element**
StereotypeOnEdge		**UML4SysML::Element**

Table 17.3 - Notations for Stereotype Use

Node Name	Concrete Syntax	Abstract Syntax Reference
Stereotype Compartment	**«stereotypeName» NodeName** «stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName	**UML4SysML::Element**

17.2.2.1 StereotypeInNode

Figure 17.2 shows how the stereotype *Clock*, as defined in Figure 17.1, is applied to a class called *AlarmClock*.

Figure 17.2 - Using a stereotype

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.

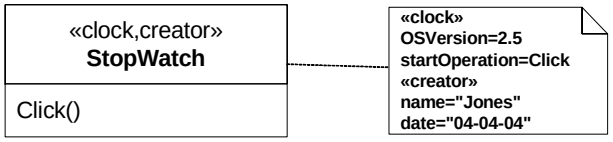


Figure 17.3 - Using stereotypes and showing values

17.2.2.3 StereotypeInCompartment

Finally, the compartment form is shown.

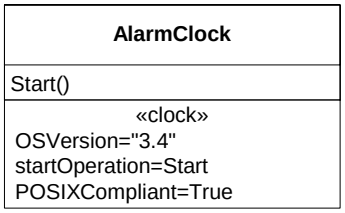


Figure 17.4 - Other notational forms for showing values

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

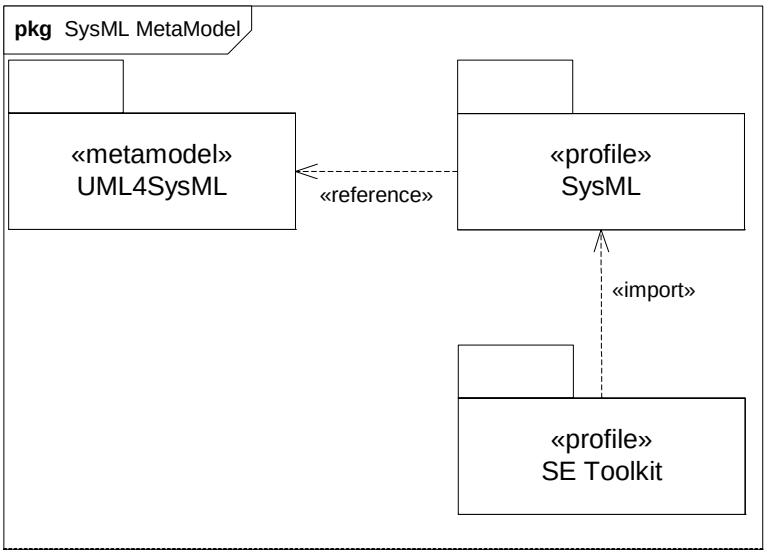


Figure 17.5 - Definition of a profile

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

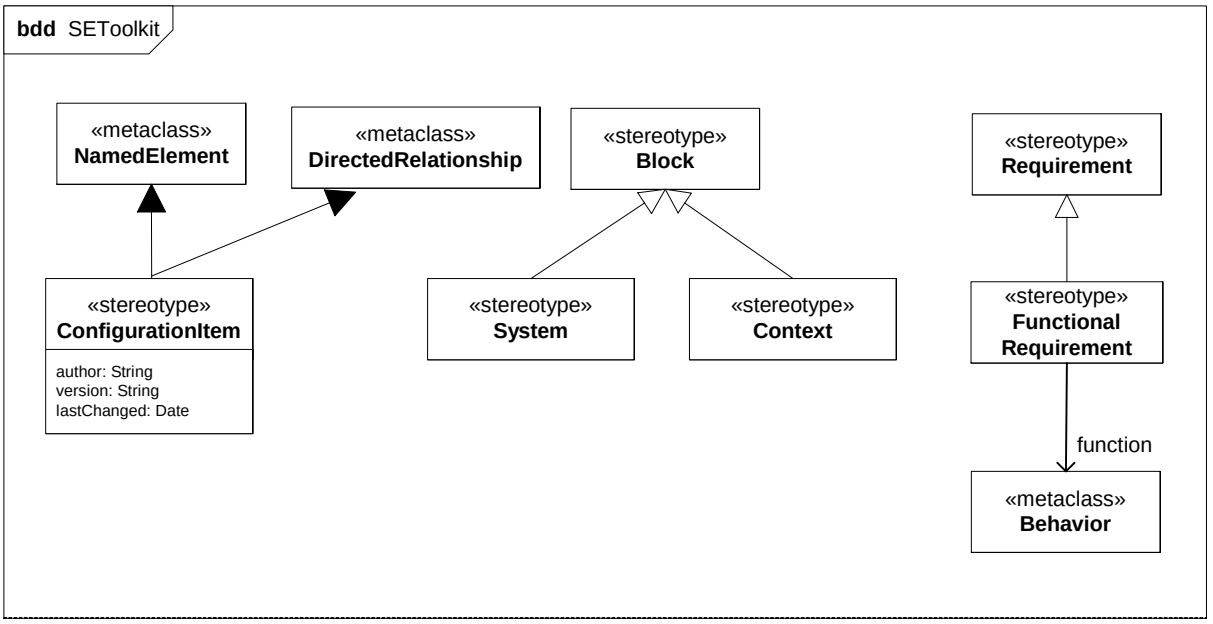


Figure 17.6 - Profile Contents

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 (in this case none) 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

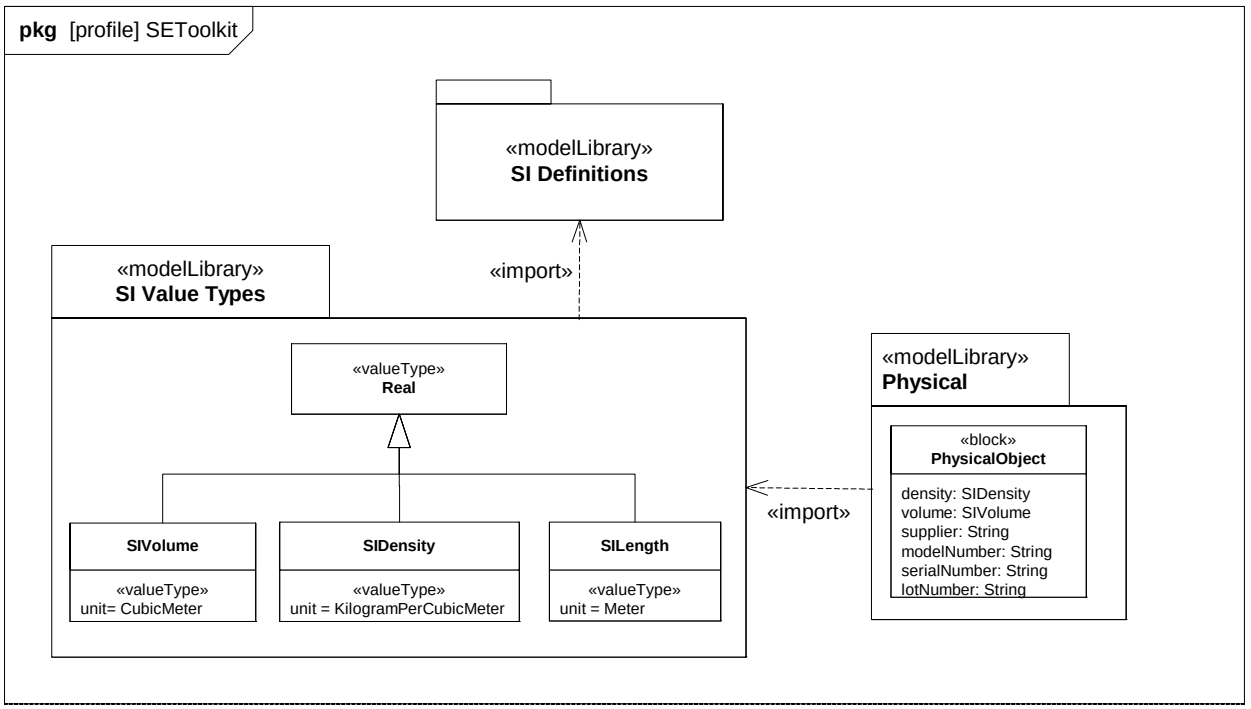


Figure 17.7 - Two model libraries

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 dimensions such as shown in Annex C, Section C.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 an physical object.

17.4.4 Guidance on Whether to Use a Stereotype or Class

This section 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 amongst 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

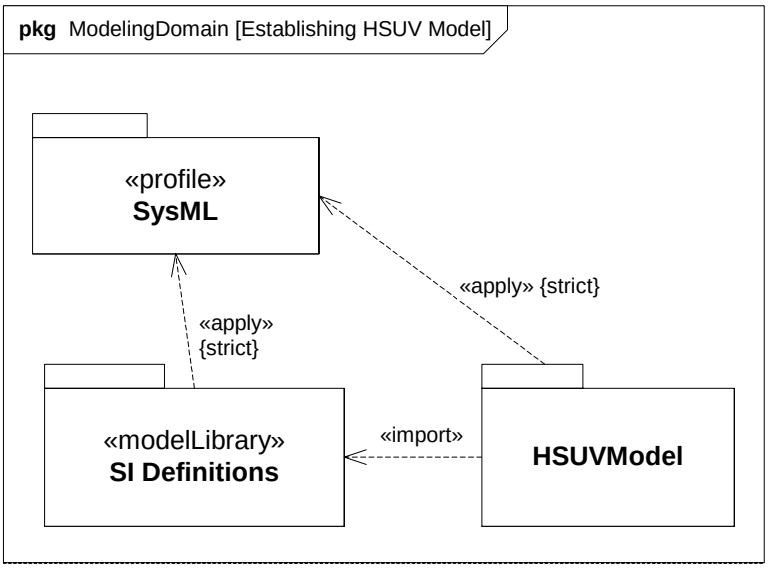


Figure 17.8 - A model with applied profile and imported model library

The HSUVModel is a system 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

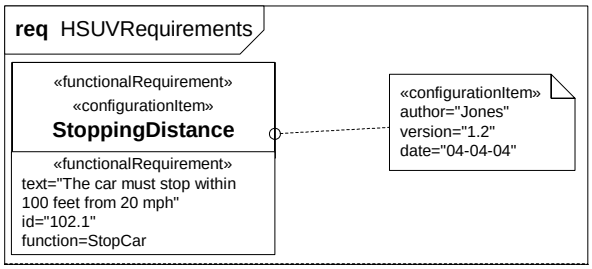


Figure 17.9 - Using two stereotypes on a model element

StoppingDistance has two stereotypes applied, functionalRequirement, that 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

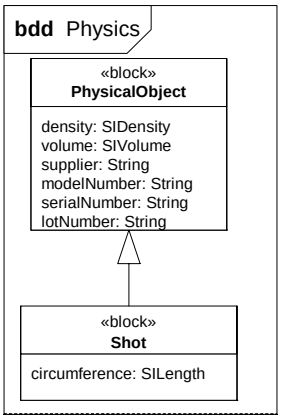


Figure 17.10 - Using model library elements

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.

Part V - Annexes

This section contains the following non-normative annexes for this specification.:

- A - Diagrams
- B - Sample Problem
- C - Non-normative Extensions
- D - Model Interchange
- E - Requirements Traceability
- F - Terms and Definitions
- G - BNF Diagram Syntax Definitions

Annex A: Diagrams

(informative)

A.1 Overview

SysML diagrams contains 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. SysML reuses many of the major diagram types of UML. In some cases, the UML diagrams are strictly re-used such as use case, sequence, state machine, and package diagram, 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 Chapter 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, and deployment 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 behavior diagrams provided adequate coverage for representing behavior without the need to include these diagram types. Two new diagram types have been added to SysML including the requirement diagram and the parametric diagram.

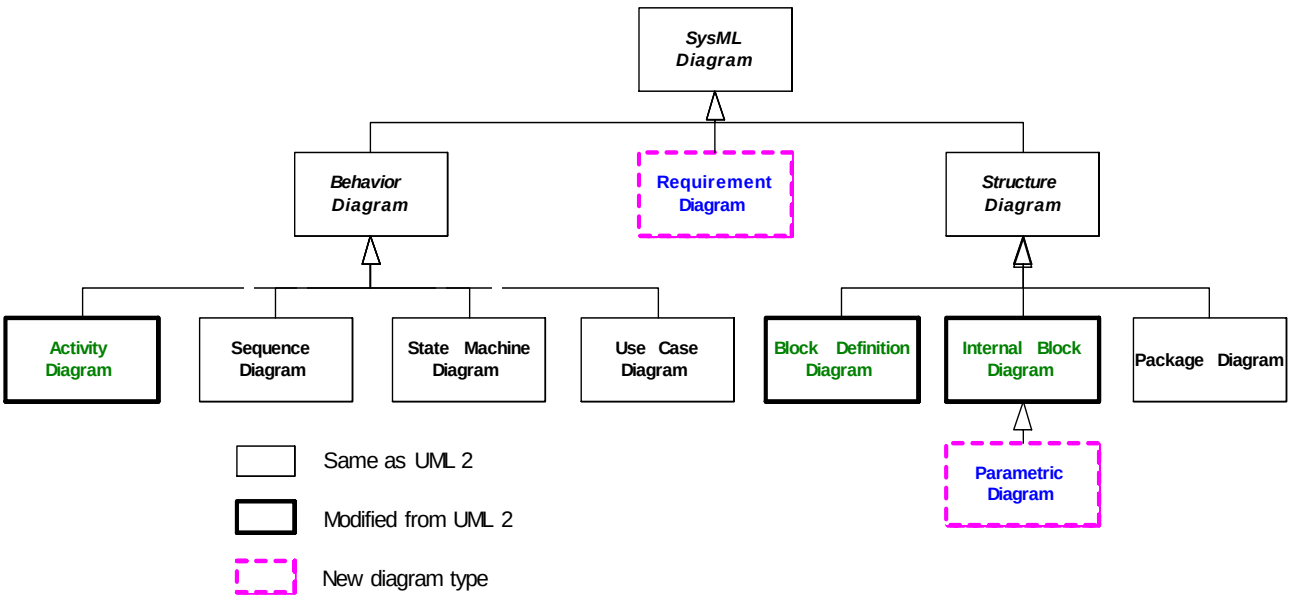


Figure A.1 - SysML Diagram Taxonomy

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 chapter describe what symbols can appear in the diagram, but do not specify the different combinations of symbols that can be used. However, the BNF Diagram Syntax Definitions referred to in the Language Formalism section, is intended to provide the formalism to specify this precisely. At this time, the SST has only implemented the BNF in the three chapters referred to in Annex G.

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 Chapter 17, “Profiles & Model Libraries.” Chapter 16, “Requirements” and Chapter 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 ten SysML diagrams kinds are described in the SysML chapters as indicated below.

- activity diagram - Activities chapter
- block definition diagram - Blocks chapter, Ports and Flows chapter
- internal block diagram - Blocks chapter, Ports and Flows chapter
- package diagram - Model Elements chapter
- parametric diagram - Constraint Blocks chapter
- requirements diagram - Requirements chapter
- state machine diagram - State Machines chapter
- sequence diagram - Interactions chapter
- use case diagram - Use Cases chapter
- Other (allocation tables) - Allocation Chapter

Each SysML diagram has a *frame*, with a *contents area*, a *heading, and a Diagram Description* see Figure A.2

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 can 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 must be provided if it is not contained within default namespace associated with the frame. The top level "Model" name is the highest level namespace for the model elements. The following are 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, and 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 defines 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 optionally include the 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.

SysML diagrams 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 chapter). 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 an 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. [Note: The use of stereotype for diagram usage is not formally part of UML since a diagram is not currently a model element that can be extended. However, the analogy was considered to be of value and adapted for this use.]

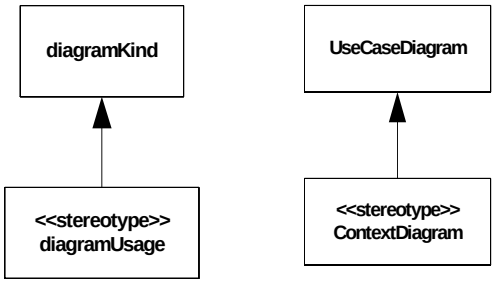


Figure A.3 - Diagram Usages

Some typical diagrams 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. The rake on a model element may 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 - 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.

- 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 provides the capability to represent a document using the UML 2.1 standard stereotype <<document>> applied to the artifact model element. Properties of the artifact can capture information about the document. Use a <<trace>> abstraction to relate the document to model elements. The document can represent text that is contained in the related model elements.
- SysML diagrams including the enhancements described in this section is intended to conform to the Diagram Interchange Standard to facilitate exchange of diagram and layout information. A more formal BNF has been introduced in selected chapters to facilitate diagram interchange, which is referred to in the Language Formalism chapter.
- Tabular 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 representations are often used in systems engineering to represent detailed information 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. The UML superstructure contains a tabular representation of a sequence diagram in an interaction matrix (refer to Superstructure Appendix with interaction matrix). The implementations of tabular representations are defined by the tool implementations and are not standardized in SysML at this time. However, tabular representations may be included in a frame with the heading designator <<**table**>> in bold.
- Graph and tree representations are also an optional alternative notation 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. 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.

Annex B: Sample Problem

(informative)

B.1 Purpose

The purpose of this annex is to illustrate how SysML can support of the specification, analysis, and design of a system using some of the basic features of the language.

B.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 also 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 chapters 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.

B.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 appendix is structured to show each diagram in the context of how it might be used on such a example problem. The first section shows SysML diagrams as they might be used to establish the system context; establishing system boundaries, and top level use cases. The next section 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 section focuses on use of SysML diagrams for capturing and deriving requirements, using diagrams and tables. A section 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 section. A section is then dedicated to illustrating definition and depiction of interfaces and flows in a structural context. The final section focuses on detailed behavior modeling, functional and flow allocation.

B.4 Diagrams

B.4.1 Package Overview (Structure of the Sample Model)

B.4.1.1 Package Diagram - Applying the SysML Profile

As shown in Figure B.1, the HSUVModel is a package that represents the user model. The SysML Profile must 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 must be imported into the user model as indicated.

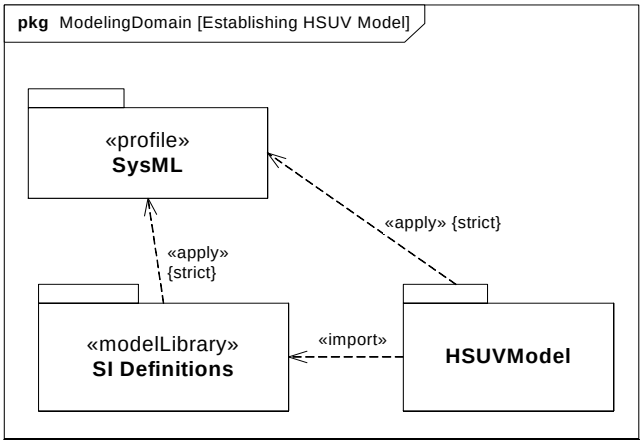


Figure B.1 Establishing the User Model by Importing and Applying SysML Profile & Model Library (Package Diagram)

Figure B.2 details the specification of units and valueTypes employed in this sample problem.

.

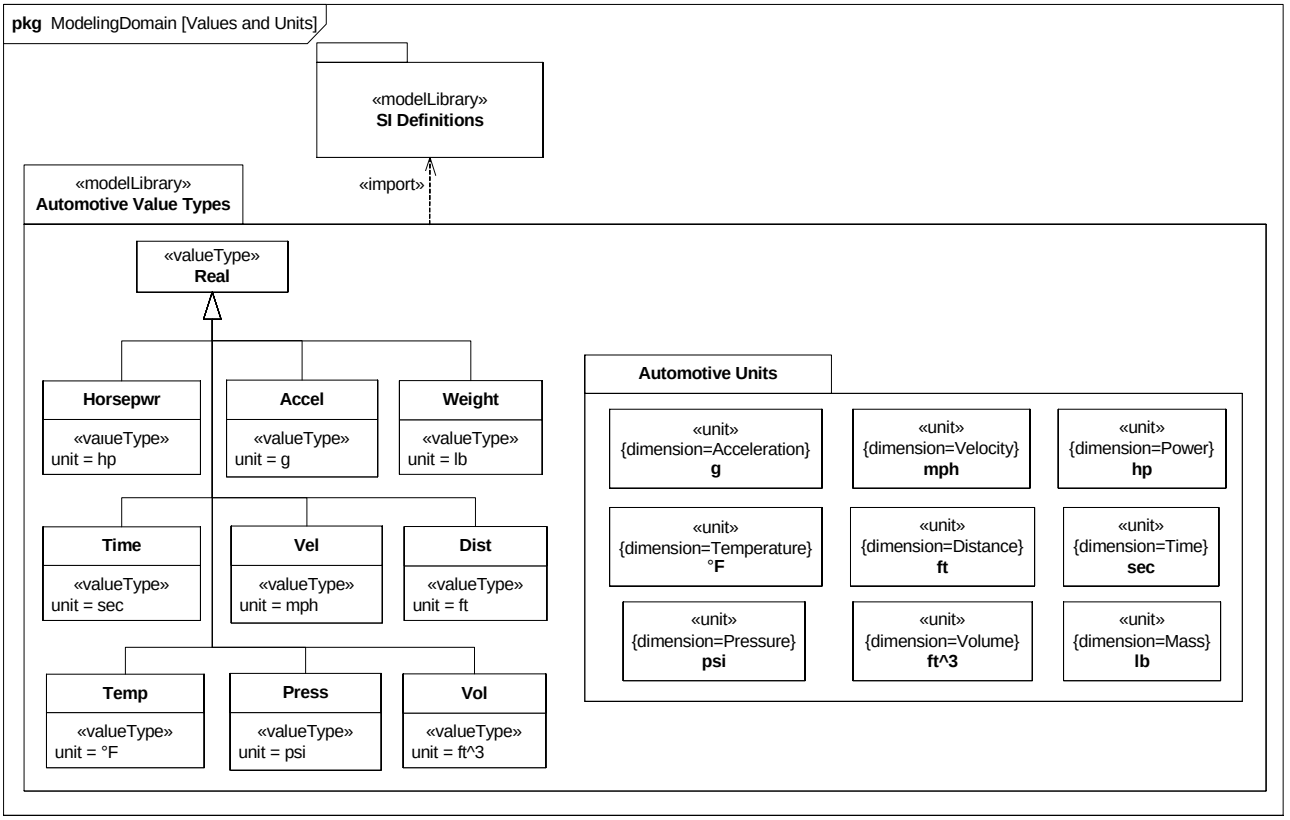


Figure B.2 - Defining valueTypes and units to be Used in the Sample Problem

B.4.1.2 Package Diagram - Showing Package Structure of the Model

The package diagram (Figure B.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 «access» 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.

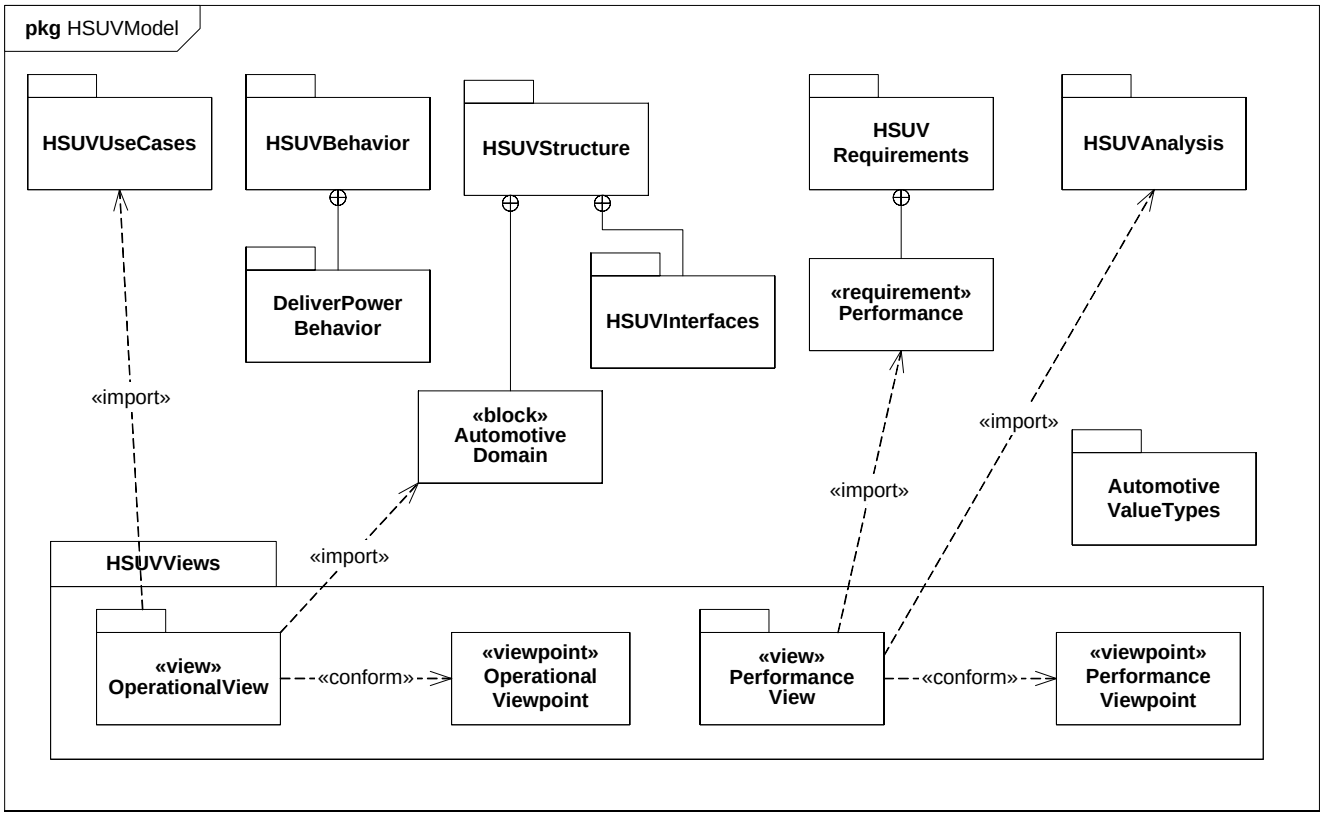


Figure B.3 - Establishing Structure of the User Model using Packages and Views (Package Diagram)

B.4.2 Setting the Context (Boundaries and Use Cases)

B.4.2.1 Internal Block Diagram - Setting Context

The term “context diagram,” in Figure B.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 B.15.

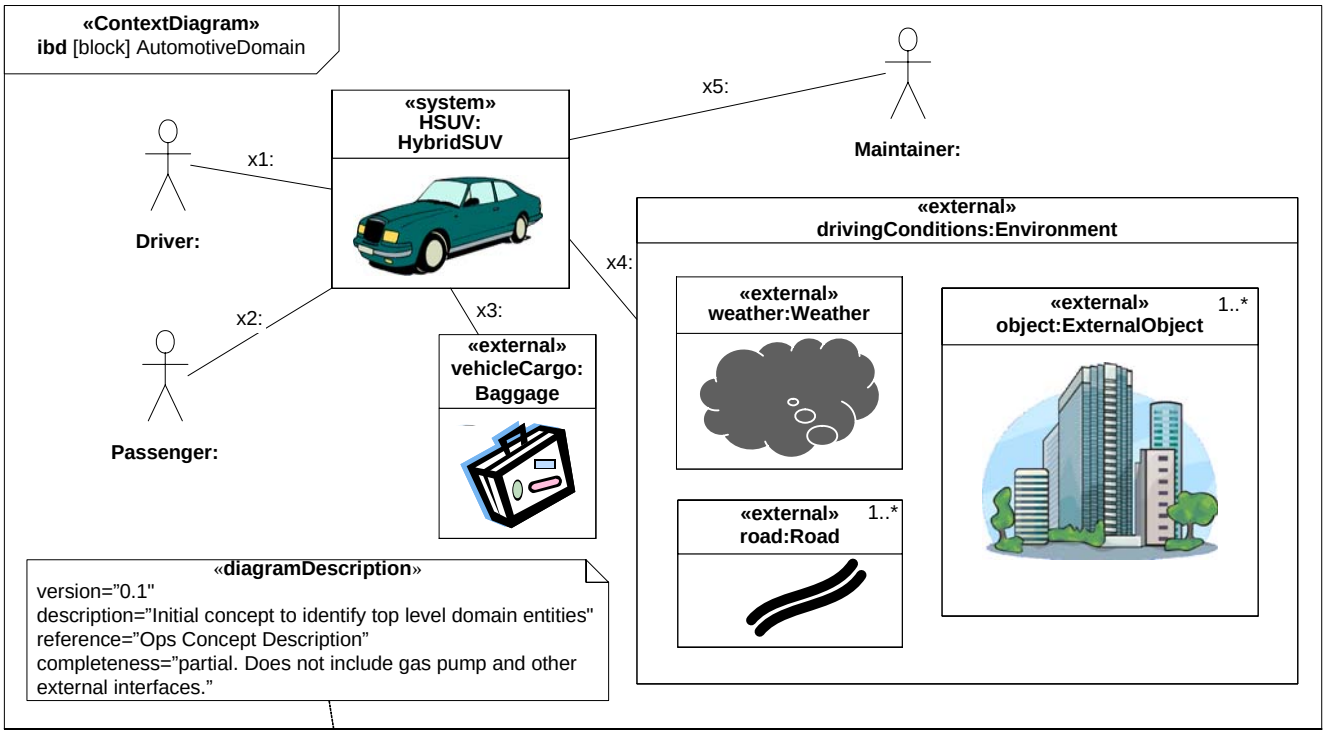


Figure B.4 - Establishing the Context of the Hybrid SUV System using a User-Defined Context Diagram. (Internal Block Diagram) Completeness of Diagram Noted in Diagram Description

B.4.2.2 Use Case Diagram - Top Level Use Cases

The use case diagram for "Drive Vehicle" in Figure B.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.

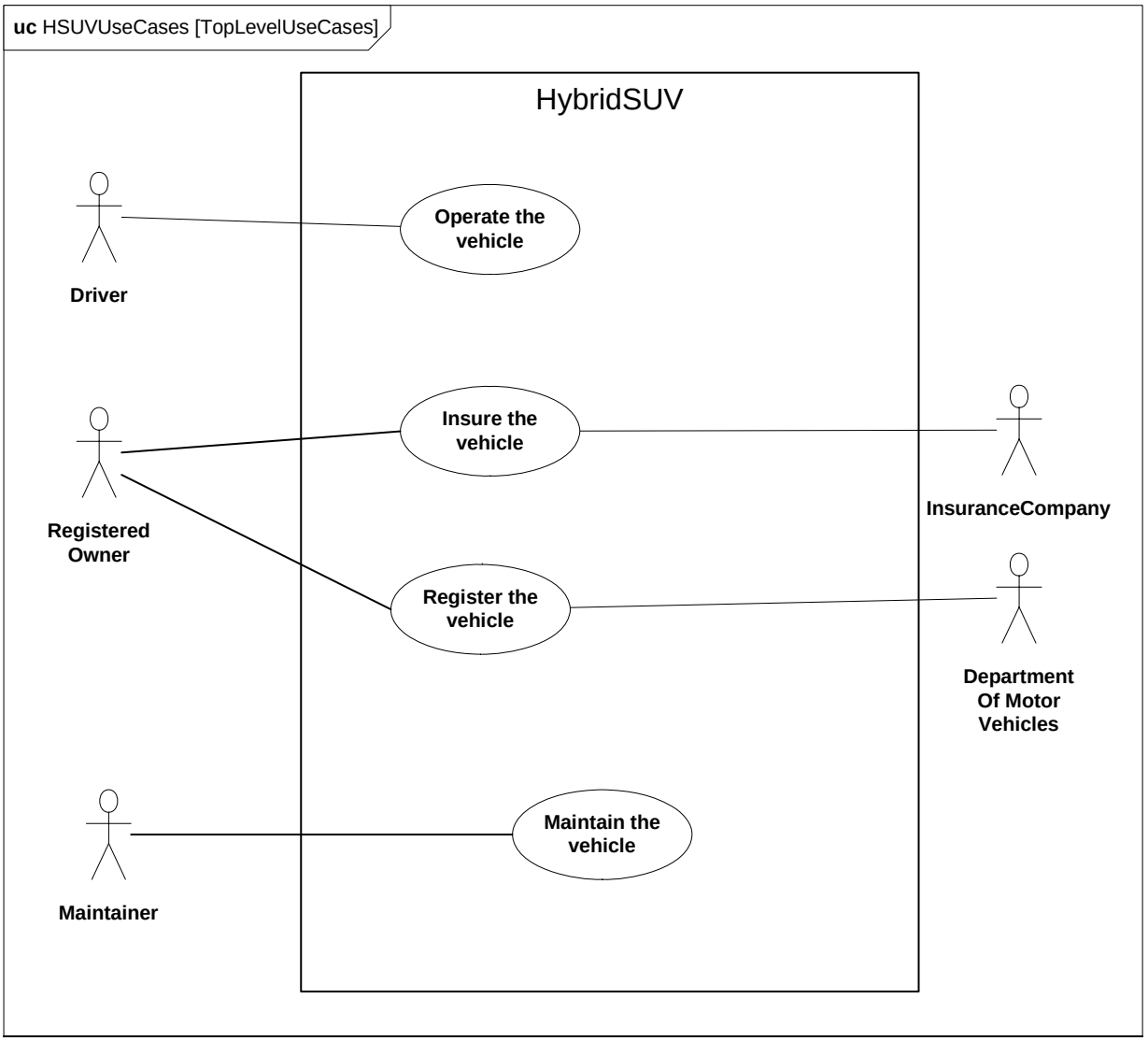


Figure B.5 - Establishing Top Level Use Cases for the Hybrid SUV (Use Case Diagram)

B.4.2.3 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.

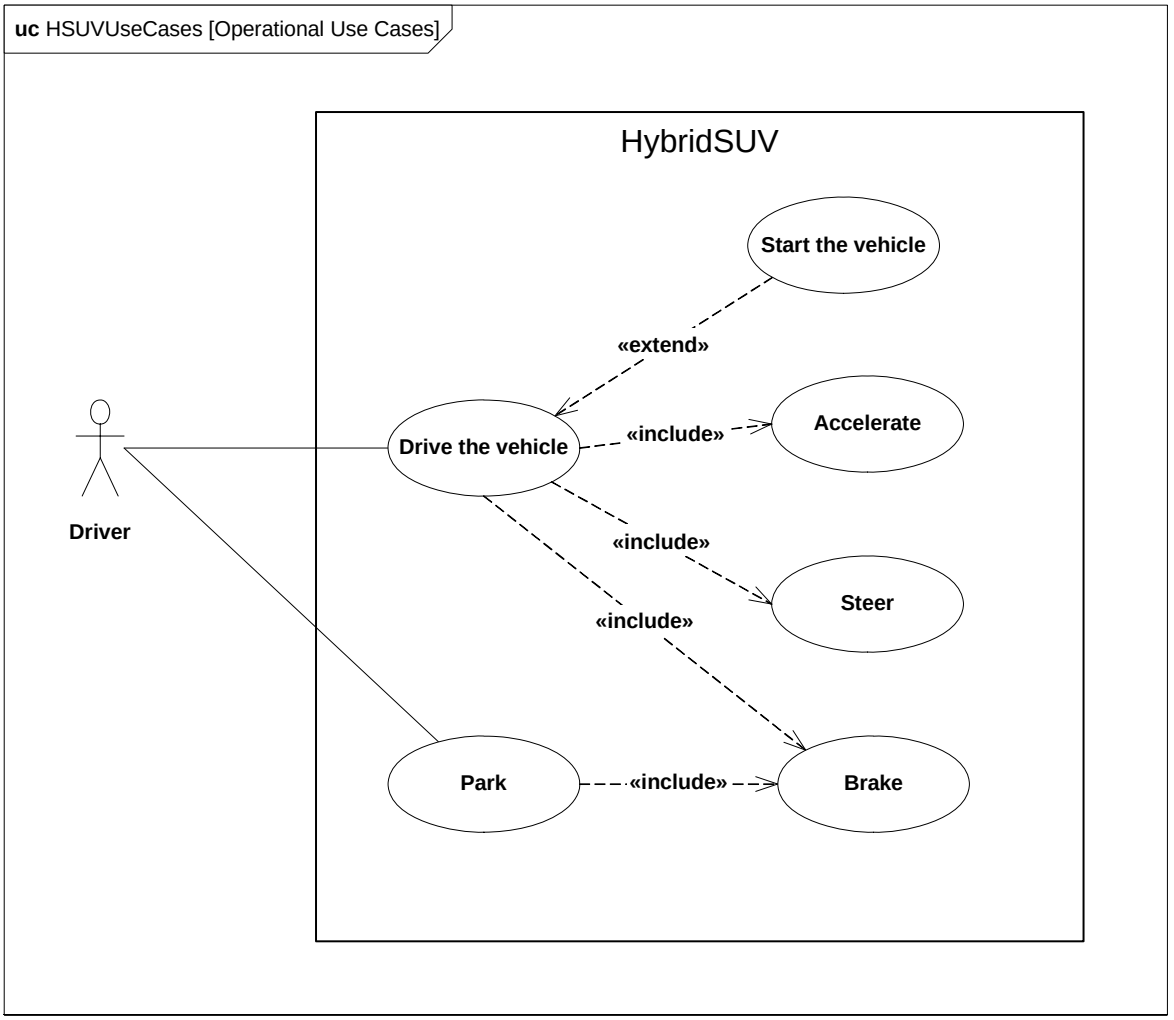


Figure B.6 - Establishing Operational Use Cases for "Drive the Vehicle" (Use Case Diagram)

B.4.3 Elaborating Behavior (Sequence and State Machine Diagrams)

B.4.3.1 Sequence Diagram - Drive Black Box

Figure B.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 section are expressed in OCL, and relate to the states of the HybridSUV block, as shown in Figure B.8.

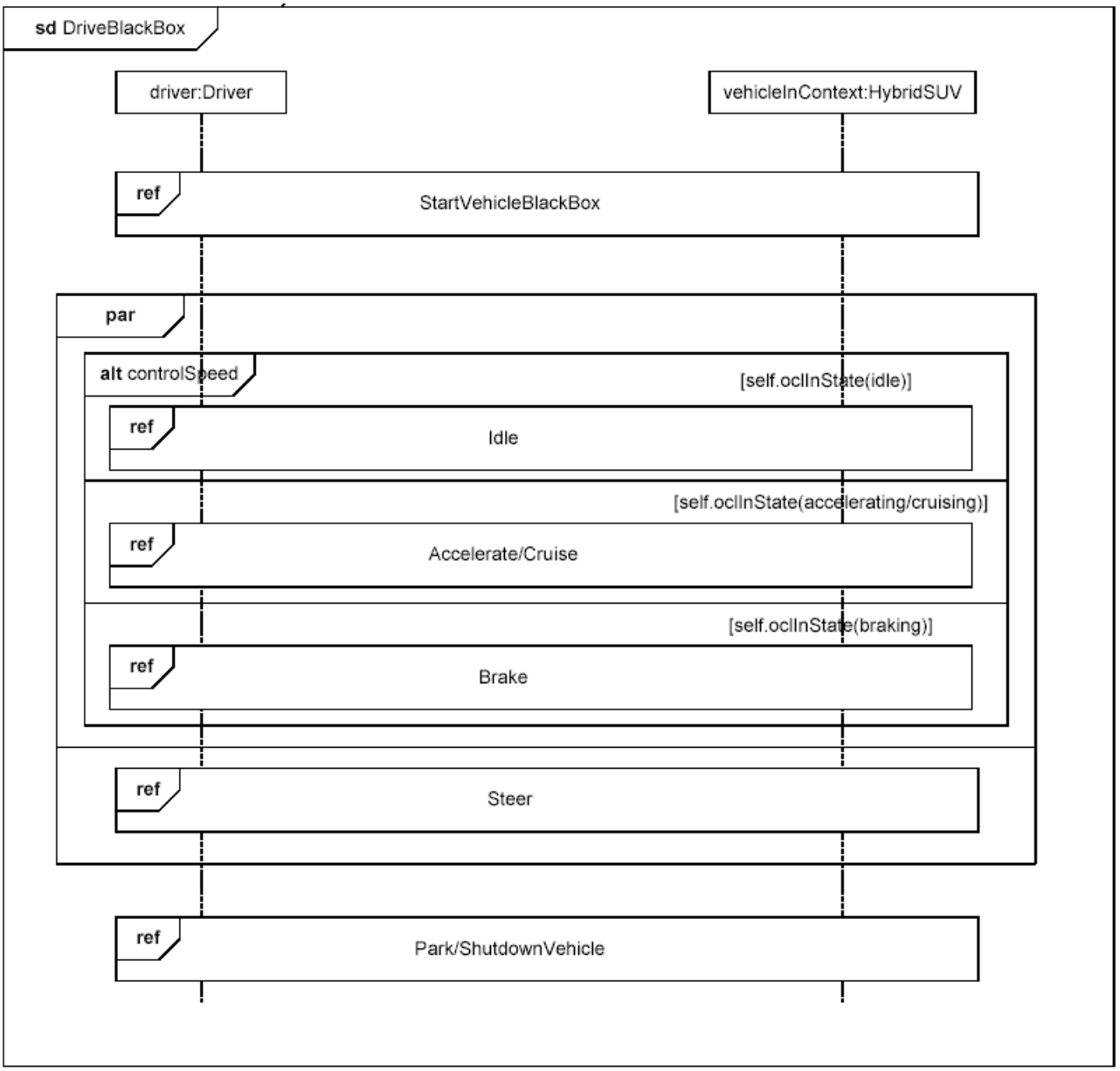


Figure B.7 - Elaborating Black Box Behavior for the "Drive the Vehicle" Use Case (Sequence Diagram)

B.4.3.2 State Machine Diagram - HSUV Operational States

Figure B.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 B.7. Also note that this state machine refines the requirement "PowerSourceManagment," which will be elaborated in the requirements section of this sample problem. This diagram expresses only the nominal states. Exception states, like "acceleratorFailure," are not expressed on this diagram.

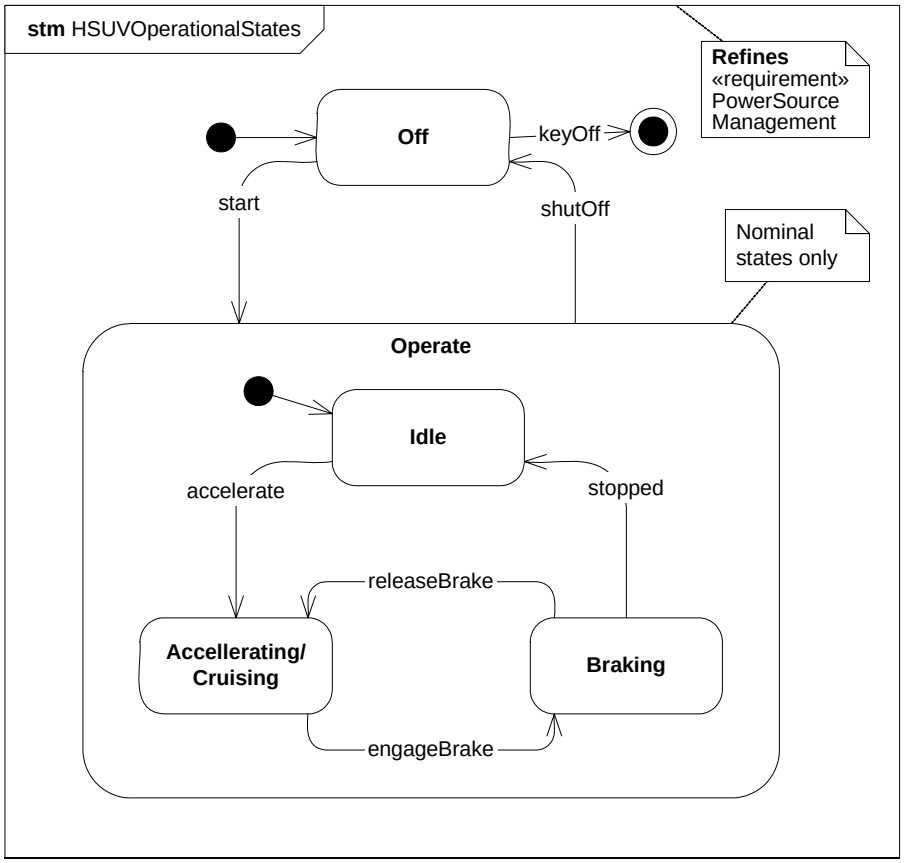


Figure B.8 - Finite State Machine Associated with "Drive the Vehicle" (State Machine Diagram)

B.4.3.3 Sequence Diagram - Start Vehicle Black Box & White Box

The Figure B.9 shows a "black box" interaction, but references "StartVehicleWhiteBox" (Figure B.10), which will decompose the lifelines within the context of the HybridSUV block.

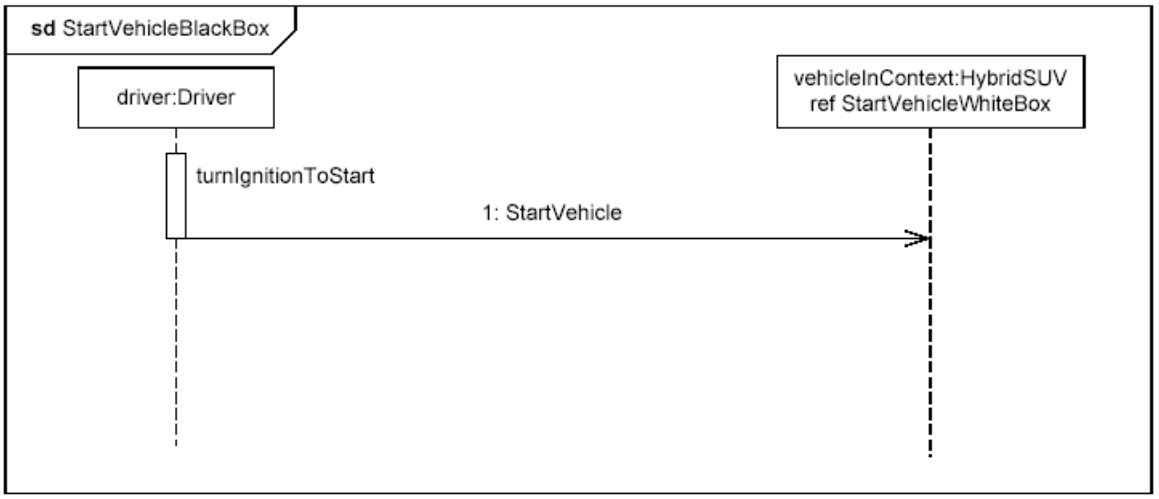


Figure B.9 - Black Box Interaction for "StartVehicle", referencing White Box Interaction (Sequence Diagram)

The lifelines on Figure B.10 ("whitebox" sequence diagram) need to come from the Power System decomposition. This now begins to consider parts contained in the HybridSUV block.

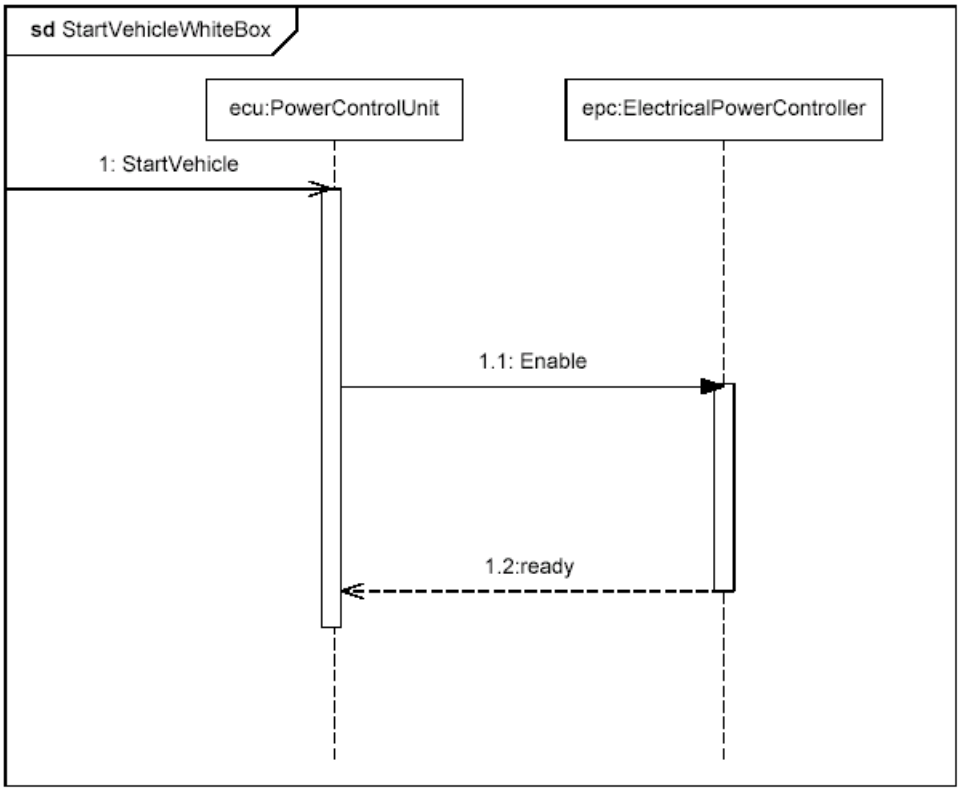


Figure B.10 - White Box Interaction for "StartVehicle" (Sequence Diagram)

B.4.4 Establishing Requirements (Requirements Diagrams and Tables)

B.4.4.1 Requirement Diagram - HSUV Requirement Hierarchy

The vehicle system specification contains many text based requirements. A few requirements are highlighted in Figure B.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.

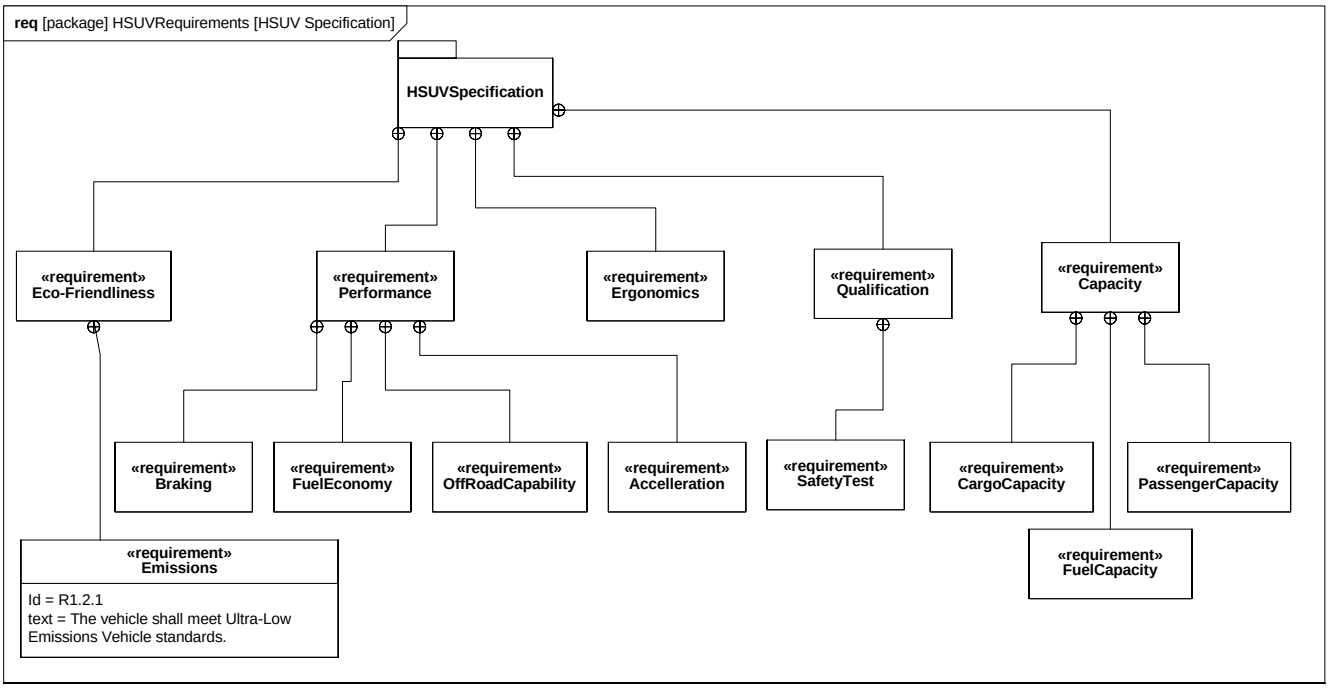


Figure B.11 - Establishing HSUV Requirements Hierarchy (containment) - (Requirements Diagram)

B.4.4.2 Requirement Diagram - Derived Requirements

Figure B.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 B.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."

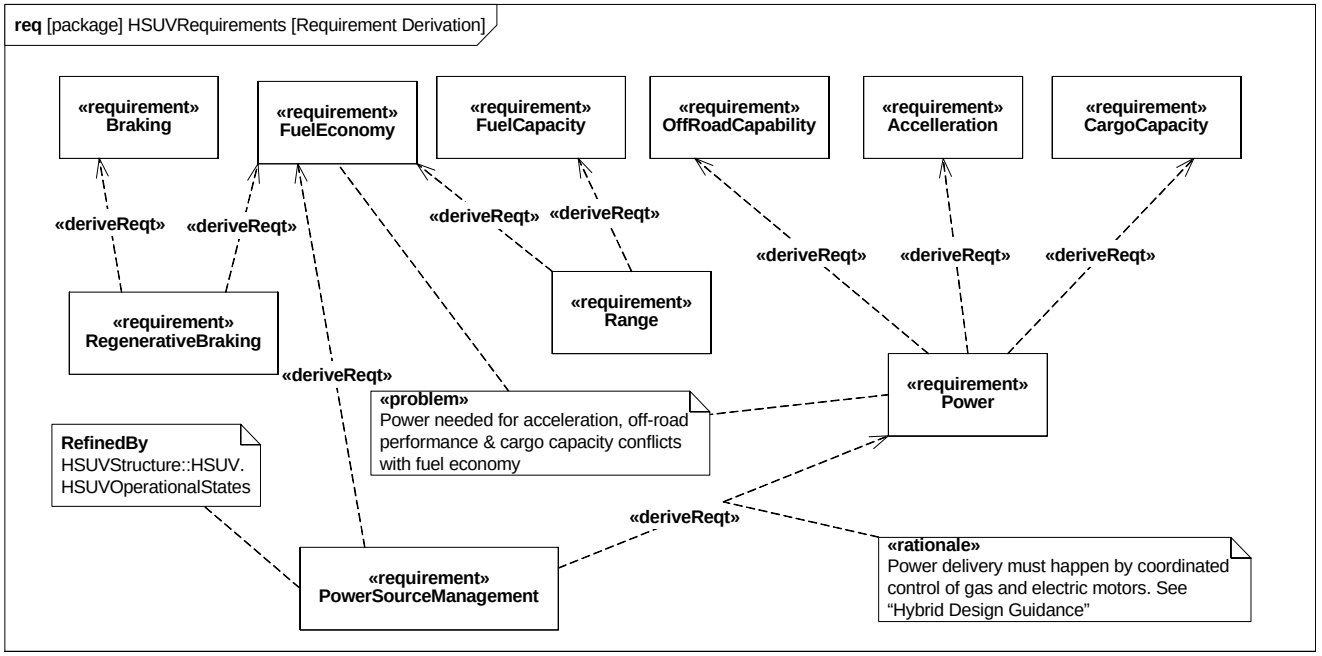


Figure B.12 - Establishing Derived Requirements and Rationale from Lowest Tier of Requirements Hierarchy (Requirements Diagram)

B.4.4.3 Requirement Diagram - Acceleration Requirement Relationships

Figure B.13 focuses on the Acceleration requirement, and relates it to other requirements and model elements. The "refineReqt" relation, introduced in Figure B.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.

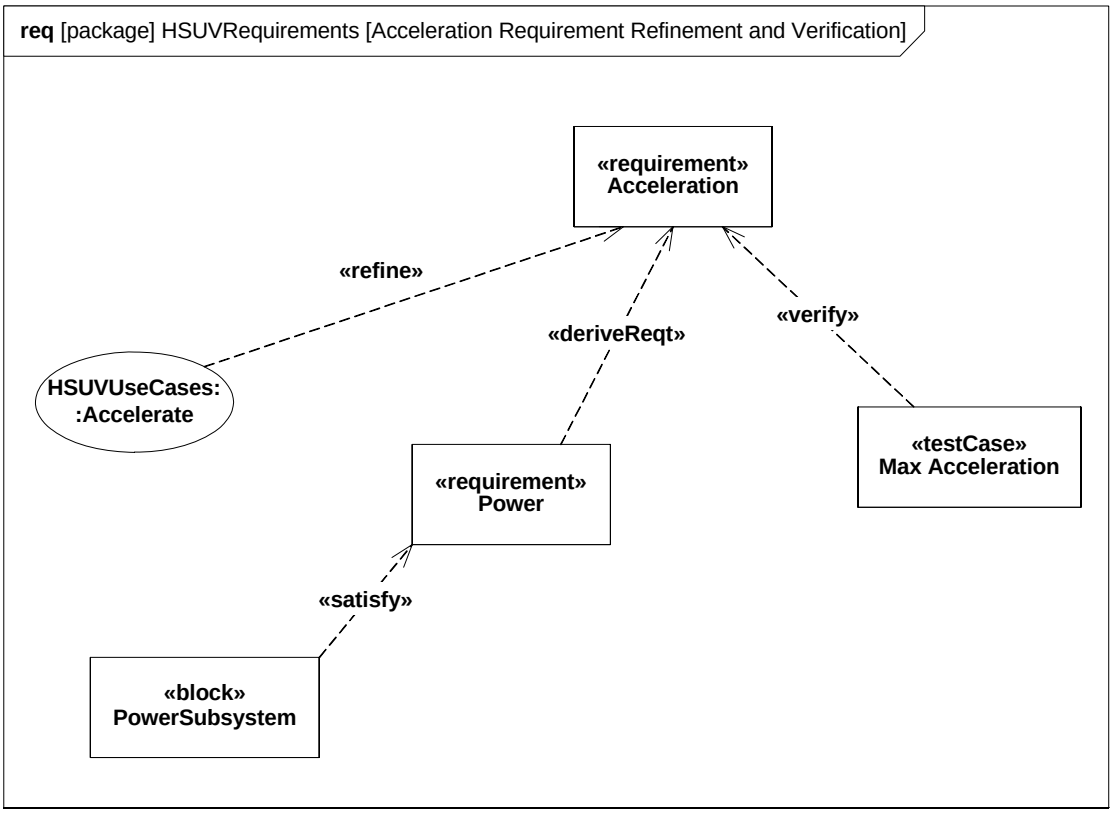


Figure B.13 - Acceleration Requirement Relationships (Requirements Diagram)

B.4.4.4 Table - Requirements Table

Figure B.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 B.14 - Requirements Relationships Expressed in Tabular Format (Table)

B.4.5 Breaking Down the Pieces (Block Definition Diagrams, Internal Block Diagrams)

B.4.5.1 Block Definition Diagram - Automotive Domain

Figure B.15 provides definition for the concepts previously shown in the context diagram. Note that the interactions DriveBlackBox and StartVehicleBlackBox (described inSection B.4.3, “Elaborating Behavior (Sequence and State Machine Diagrams),” on page 177) are depicted as owned by the AutomotiveDomain block.

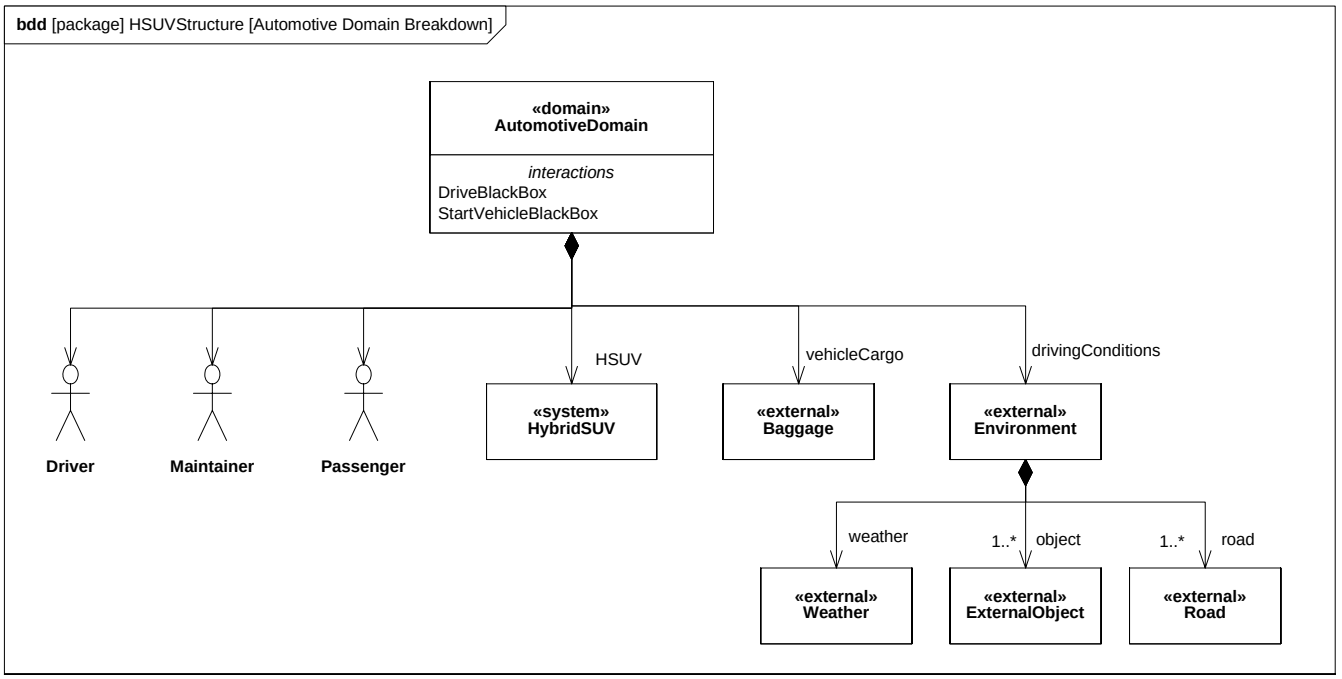


Figure B.15 - Defining the Automotive Domain (compare with Figure B.4) - (Block Definition Diagram)

B.4.5.2 Block Definition Diagram - Hybrid SUV

Figure B.16 defines components of the HybridSUV block Note that the BrakePedal and WheelHubAssembly are used by, but not contained in, the PowerSubsystem block.

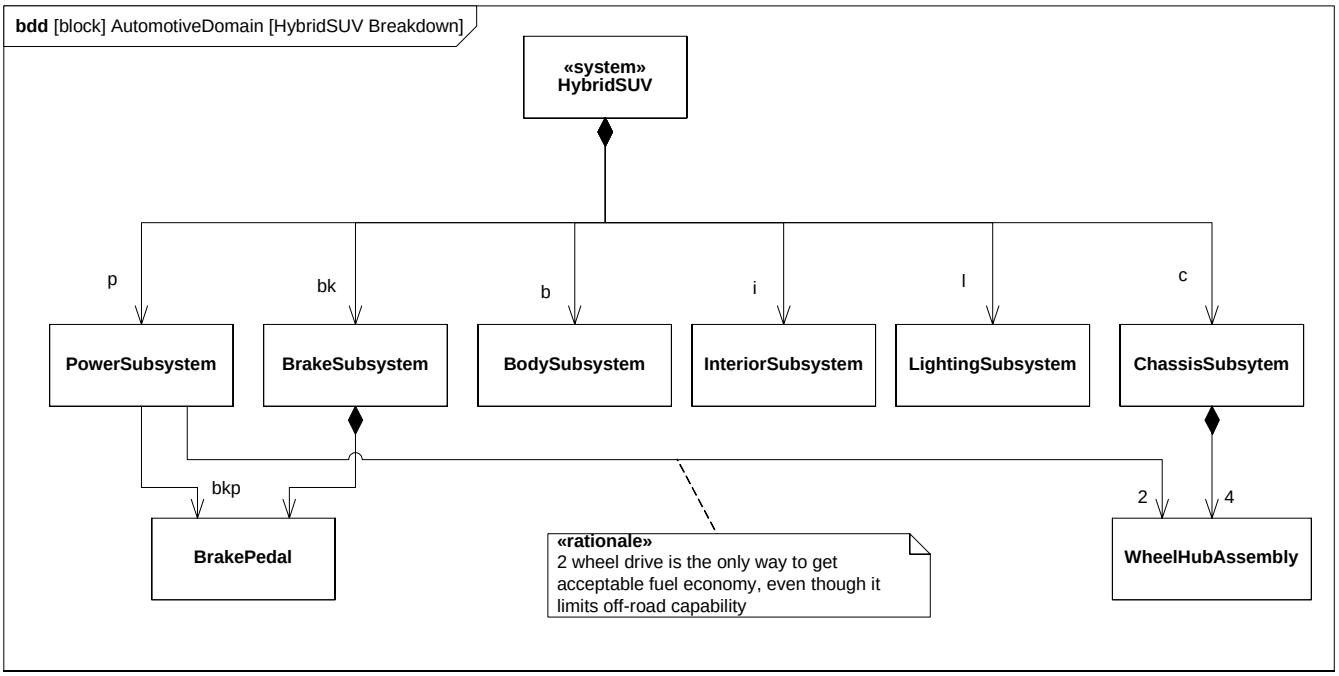


Figure B.16 - Defining Structure of the Hybrid SUV System (Block Definition Diagram)

B.4.5.3 Internal Block Diagram - Hybrid SUV

Figure B.17 shows how the top level model elements in the above diagram are connected together in the HybridSUV block.

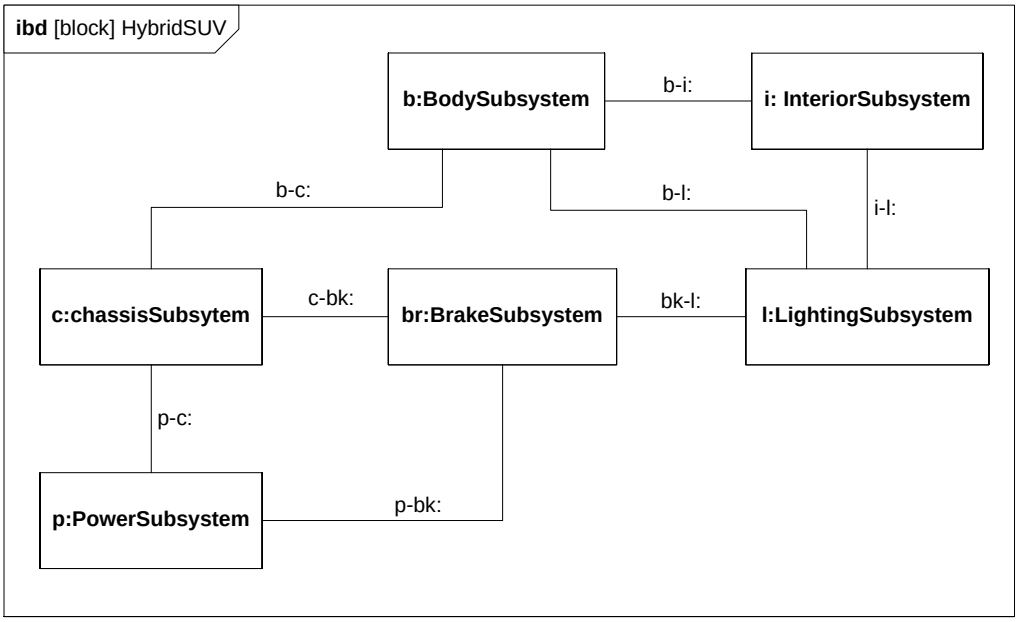


Figure B.17 - Internal Structure of Hybrid SUV (Internal Block Diagram)

B.4.5.4 Block Definition Diagram - Power Subsystem

Figure B.18 defines the next level of decomposition, namely the components of the PowerSubsystem block. Note how the of white diamond (composition) on FrontWheel and BrakePedal denotes the same “use-not-composition” kind of relationship previously shown in Figure B.16.

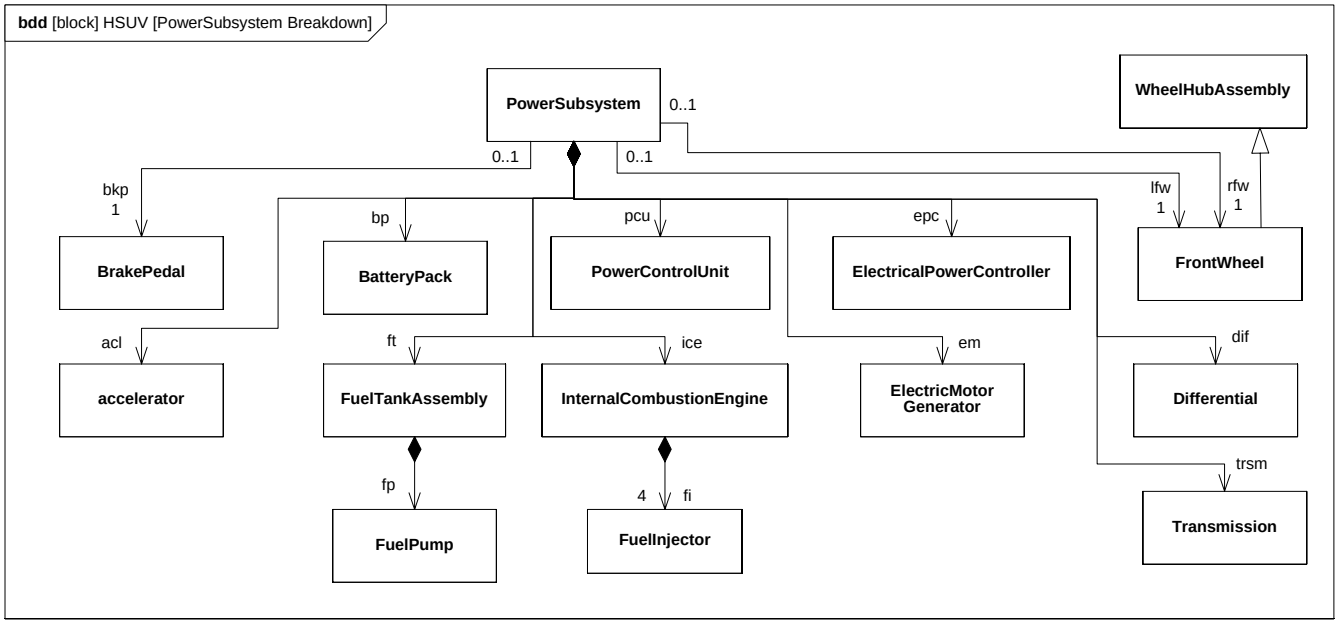


Figure B.18 - Defining Structure of Power Subsystem (Block Definition Diagram)

B.4.5.5 Internal Block Diagram for the “Power Subsystem”

Figure B.19 shows how the parts of the PowerSubsystem block, as defined in the diagram above, are used. It shows «connectors» between parts, «clientServerPorts», «flowPorts», «atomicFlowPorts», and «itemFlows». The dashed borders on FrontWheel and BrakePedal denote the “use-not-composition” relationship depicted elsewhere in Figure B.16 and Figure B.18.

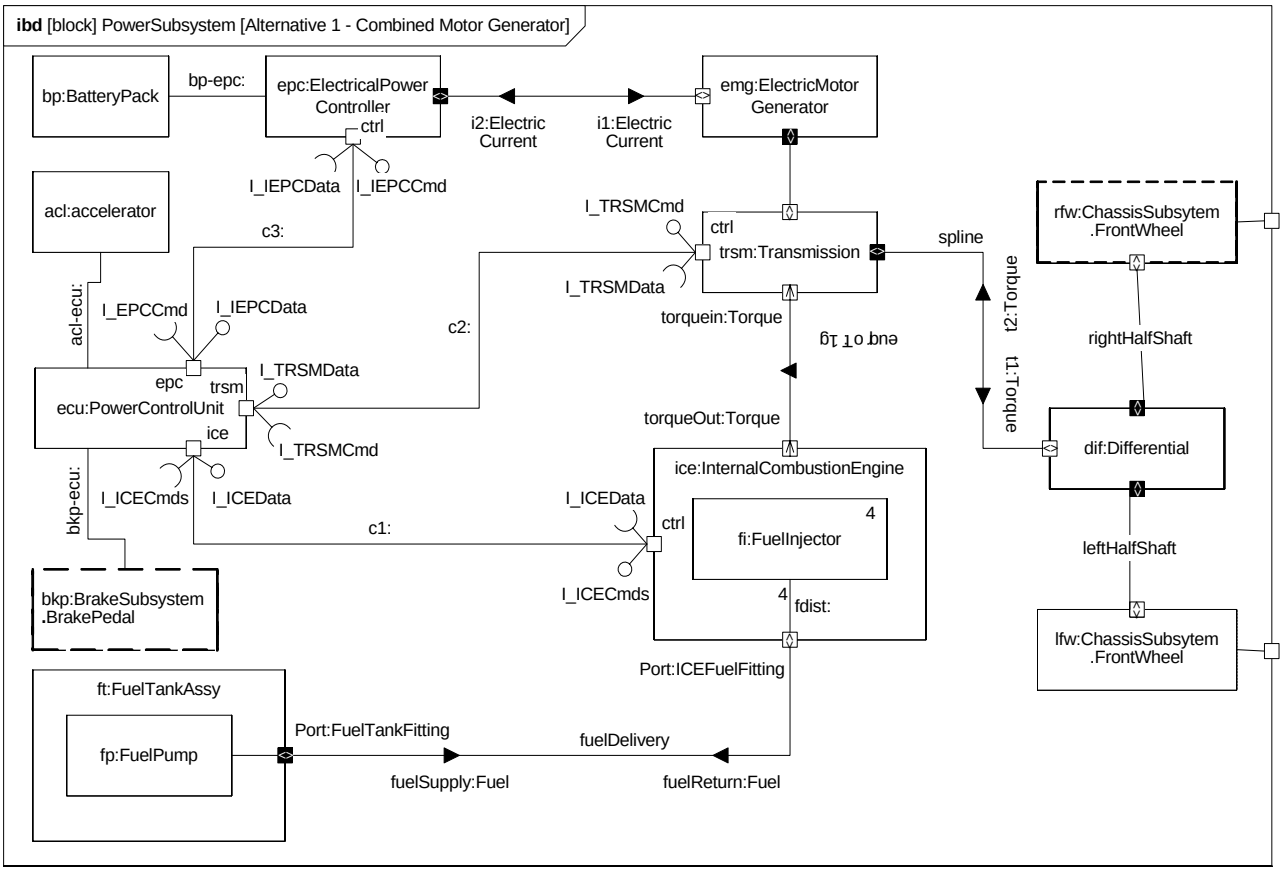


Figure B.19 - Internal Structure of the Power Sybsystem (Internal Block Diagram)

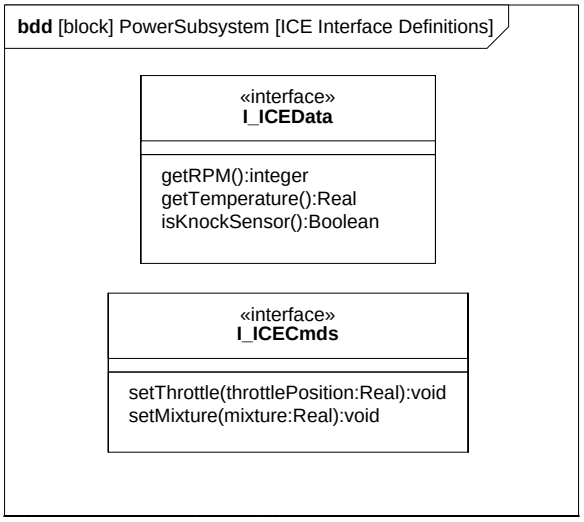


Figure B.20 - Interfaces Typing StandardPorts Internal to the Power Subsystem (Block Definition Diagram)

Figure B.20 provides definition of the interfaces applied to Standard Ports associated with connector c1 in Figure B.19.

B.4.6 Defining Ports and Flows

B.4.6.1 Block Definition Diagram - ICE Interface

For purposes of example, the StandardPorts and related point-to-point connectors in Figure B.19 are being refined into a common bus architecture. For this example, FlowPorts have been used to model the bus architecture. Figure B.21 is an incomplete first step in the refinement of this bus architecture, as it begins to identify the flow specification for the InternalCombustionEngine, the Transmission, and the ElectricalPowerController..

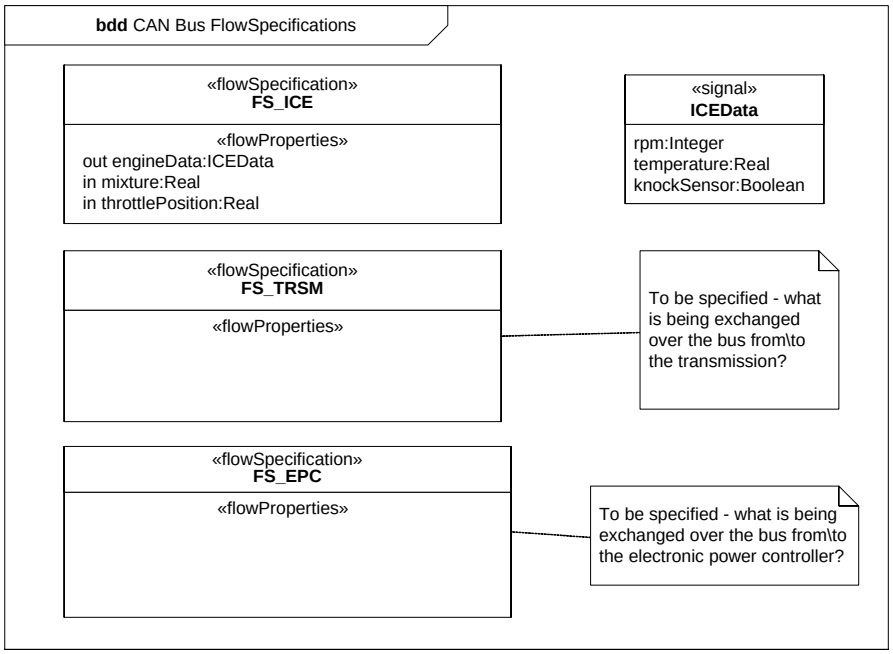


Figure B.21 - Initially Defining Flow Specifications for the CAN Bus (Block Definition Diagram)

B.4.6.2 Internal Block Diagram - CANbus

Figure B.22 continues the refinement of this Controller Area Network (CAN) bus architecture using FlowPorts. The explicit structural allocation between the original connectors of Figure B.19 and this new bus architecture is shown in Figure B.36.

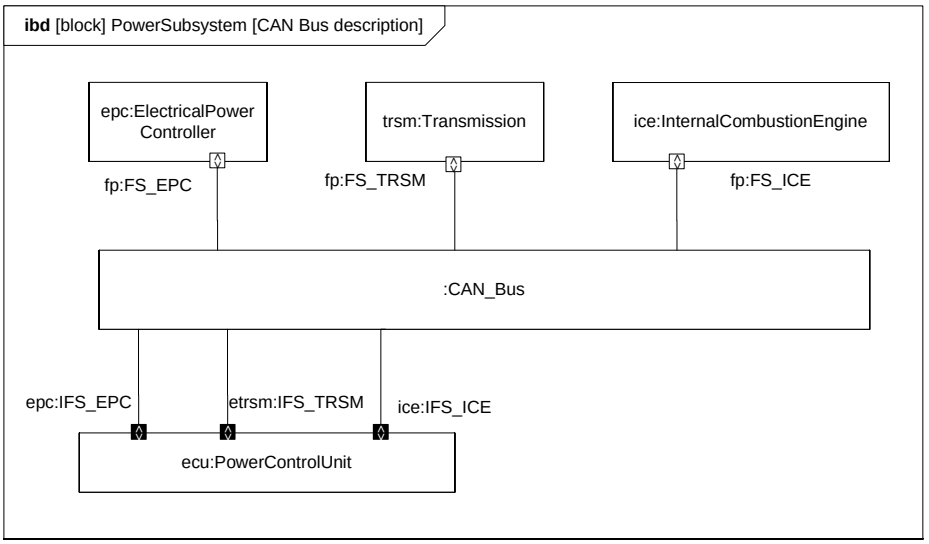


Figure B.22 - Consolidating Interfaces into the CAN Bus. (Internal Block Diagram)

B.4.6.3 Block Definition Diagram - Fuel Flow Properties

The FlowPorts on the FuelTankAssembly and InternalCombustionEngine (as shown in Figure B.19) are defined in Figure B.23.

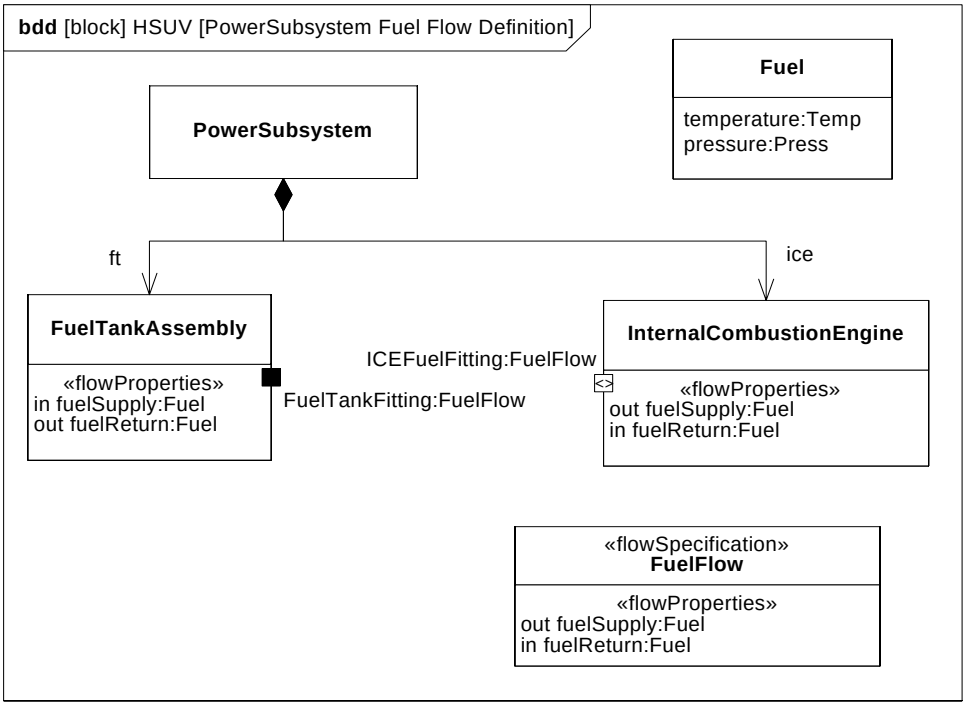


Figure B.23 - Elaborating Definition of Fuel Flow. (Block Definition Diagram)

B.4.6.4 Parametric Diagram - Fuel Flow

Figure B.24 is a parametric diagram showing how fuel flowrate is related to FuelDemand and FuelPressure value properties.

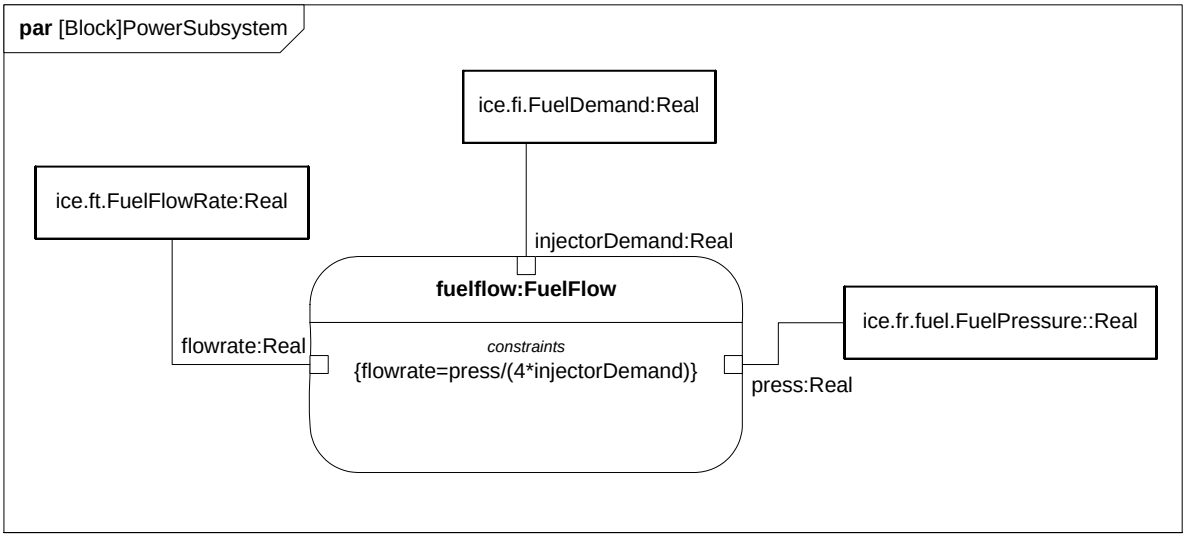


Figure B.24 - Defining Fuel Flow Constraints (Parametric Diagram)

B.4.6.5 Internal Block Diagram - Fuel Distribution

Figure B.25 shows how the connectors fuelDelivery and fdist on Figure B.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.

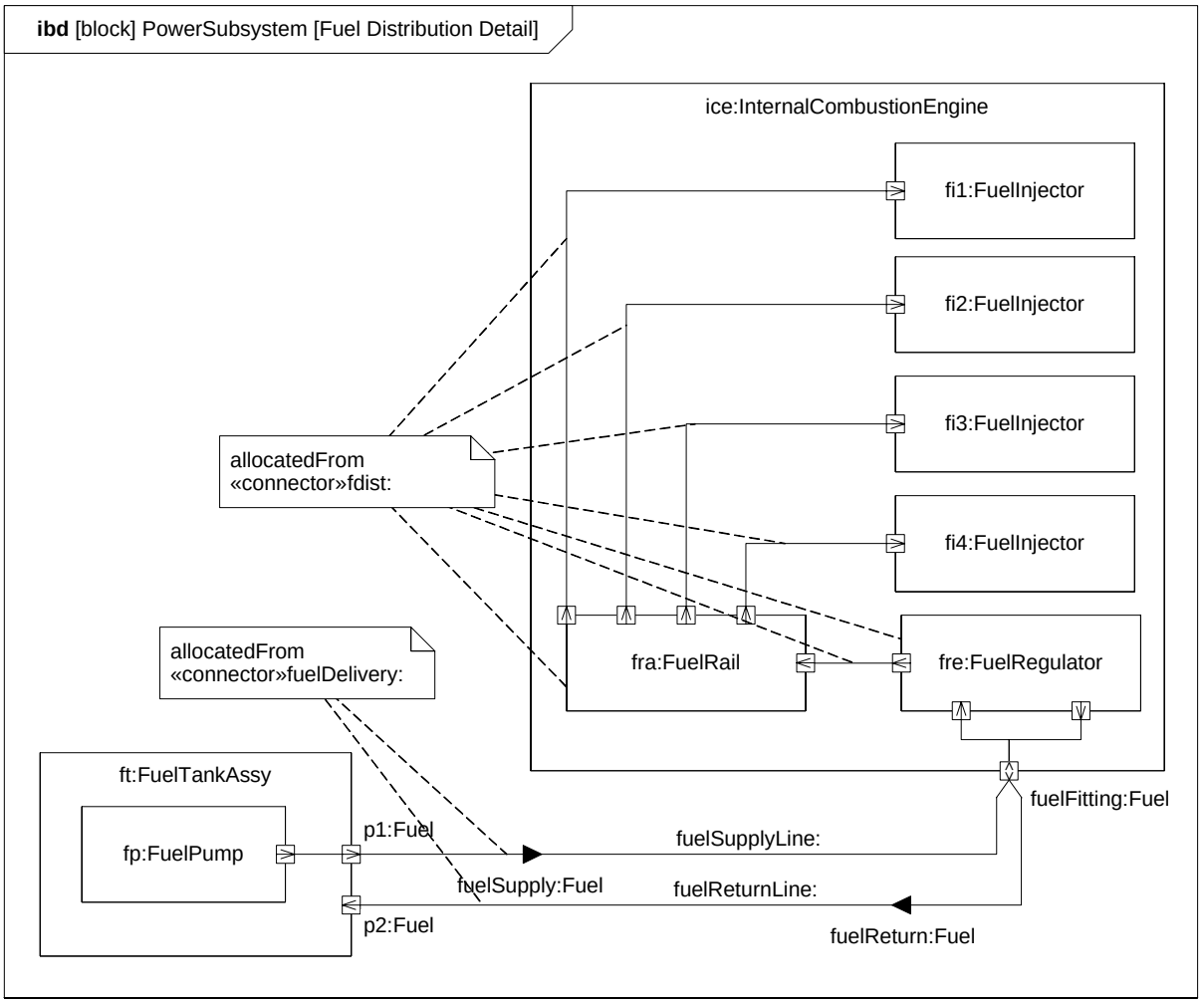


Figure B.25 - Detailed Internal Structure of Fuel Delivery Subsystem (Internal Block Diagram)

B.4.7 Analyze Performance (Constraint Diagrams, Timing Diagrams, Views)

B.4.7.1 Block Definition Diagram - Analysis Context

Figure B.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.

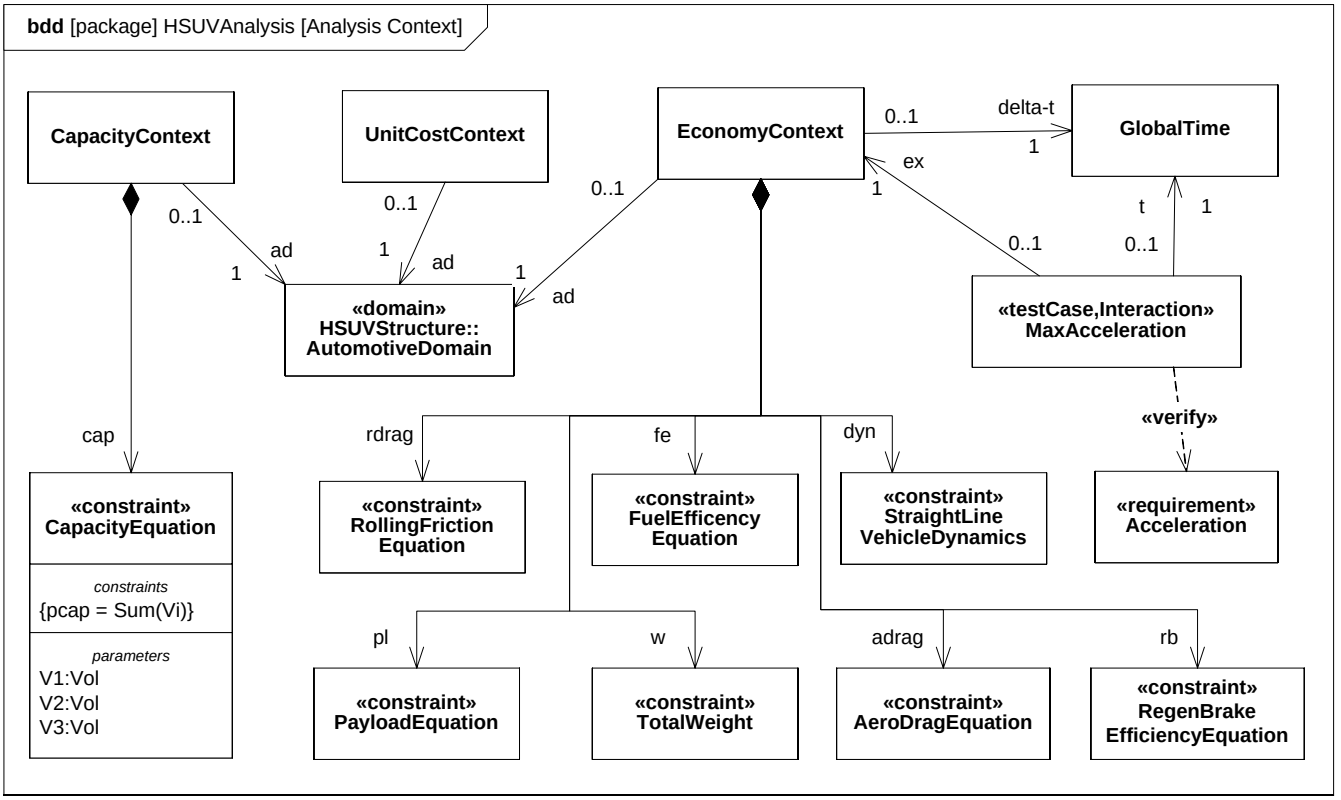


Figure B.26 - Defining Analyses for Hybrid SUV Engineering Development (Block Definition Diagram)

B.4.7.2 Package Diagram - Performance View Definition

Figure B.27 shows the user-defined Performance Viewpoint, and the elements that populate the HSUV specific PerformanceView. The PerformanceView itself may contain of a number of diagrams depicting the elements it contains.

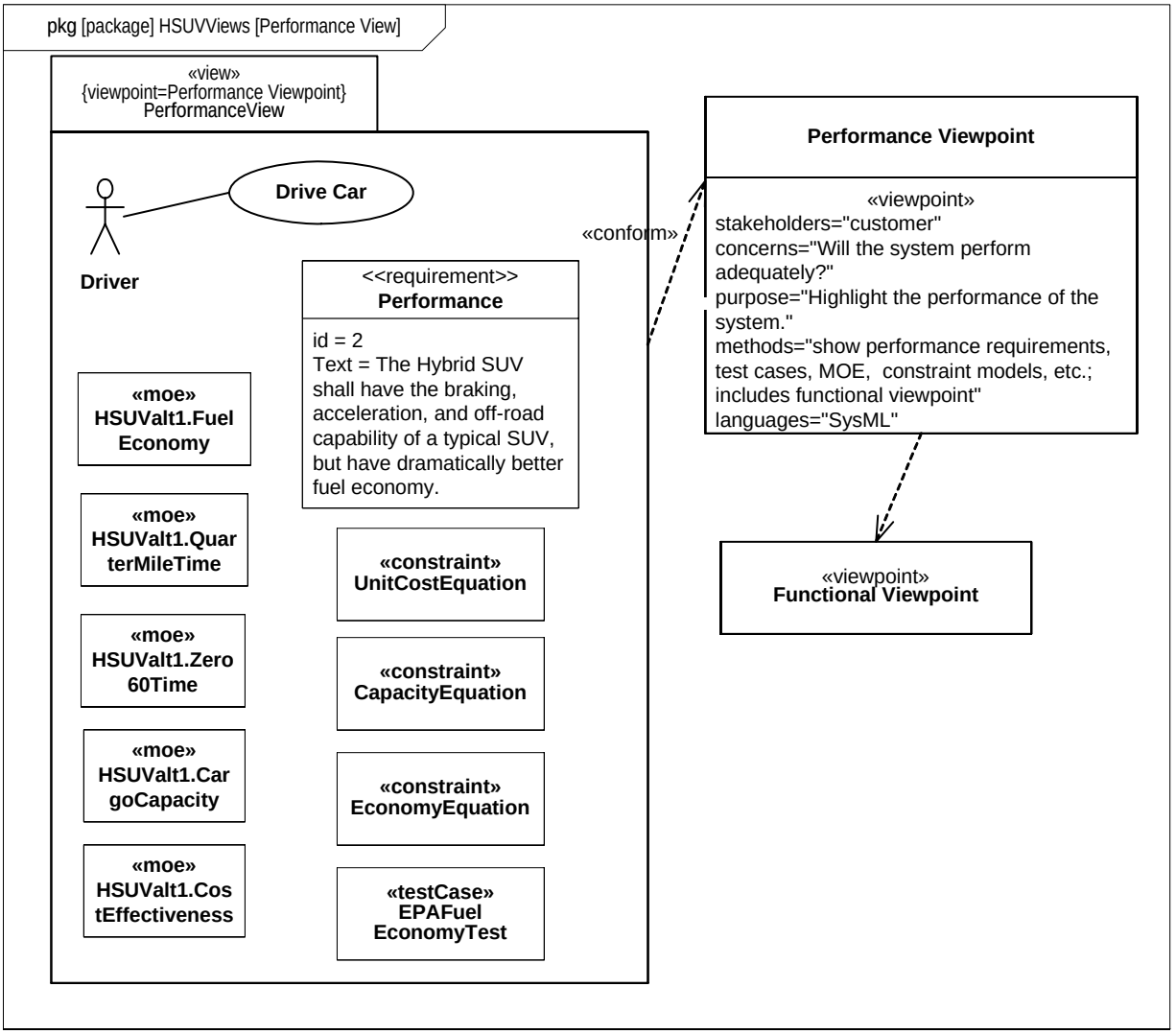


Figure B.27 - Establishing a Performance View of the User Model (Package Diagram)

B.4.7.3 Parametric Diagram - Measures of Effectiveness

Measure of Effectiveness is a user defined stereotype. Figure B.28 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.

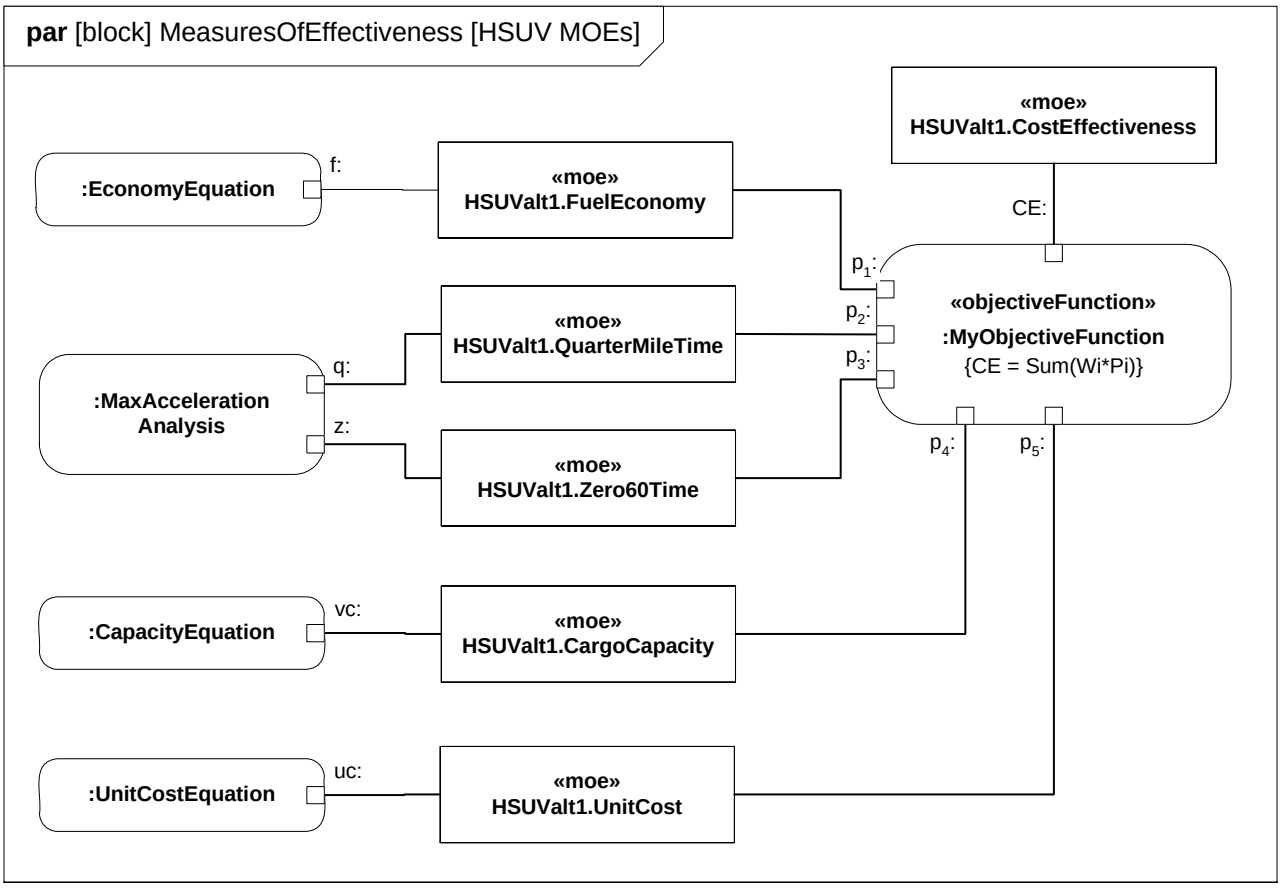


Figure B.28 - Defining Measures of Effectiveness and Key Relationships (Parametric Diagram)

B.4.7.4 Parametric Diagram - Economy

Since overall fuel economy is a key requirement on the HSUV design, this example applies significant detail in assessing it. Figure B.29 shows the constraint blocks and properties necessary to evaluate fuel economy.

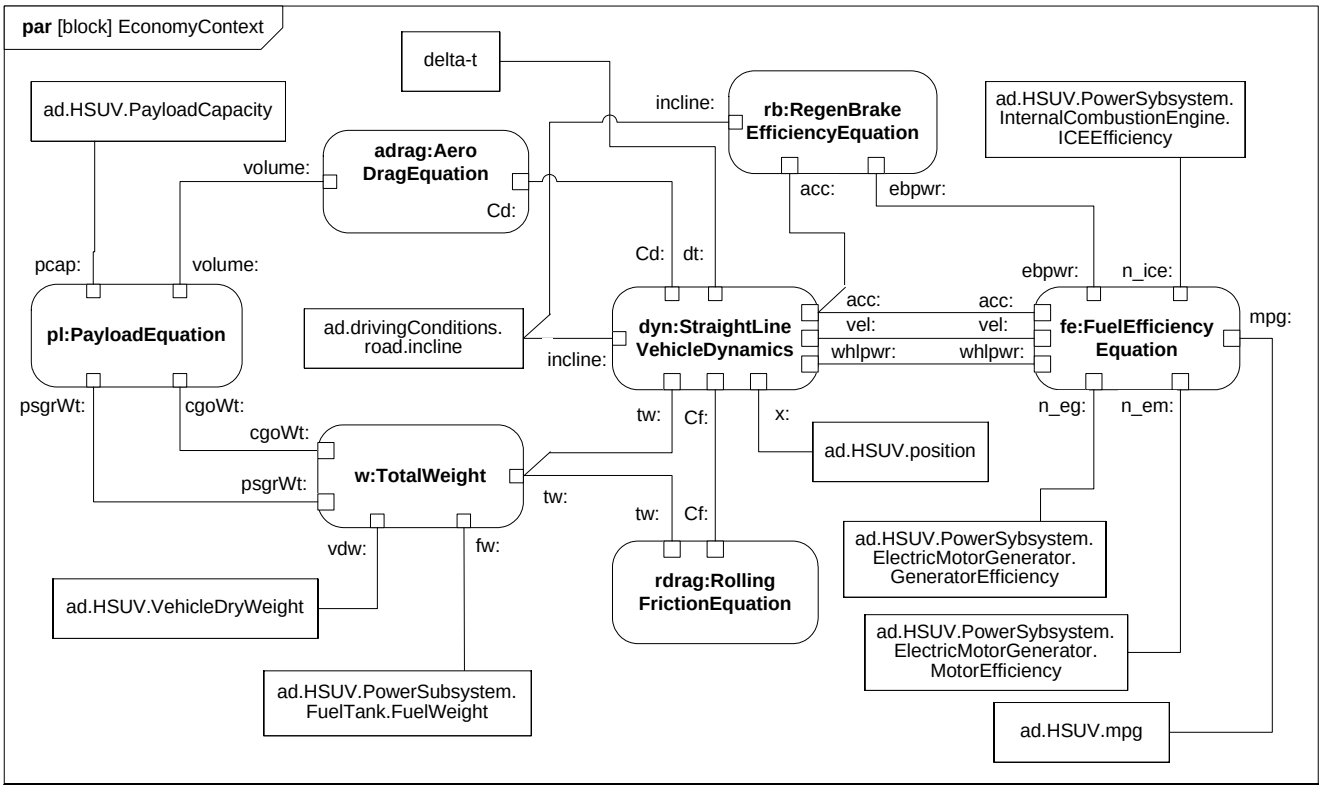


Figure B.29 - Establishing Mathematical Relationships for Fuel Economy Calculations (Parametric Diagram)

B.4.7.5 Parametric Diagram - Dynamics

The StraightLineVehicleDynamics constraint block from Figure B.29 has been expanded in Figure B.30. ConstraintNotes are used, which identify each constraint using curly brackets {}. In addition, Rationale has been used to explain the meaning of each constraint maintained.

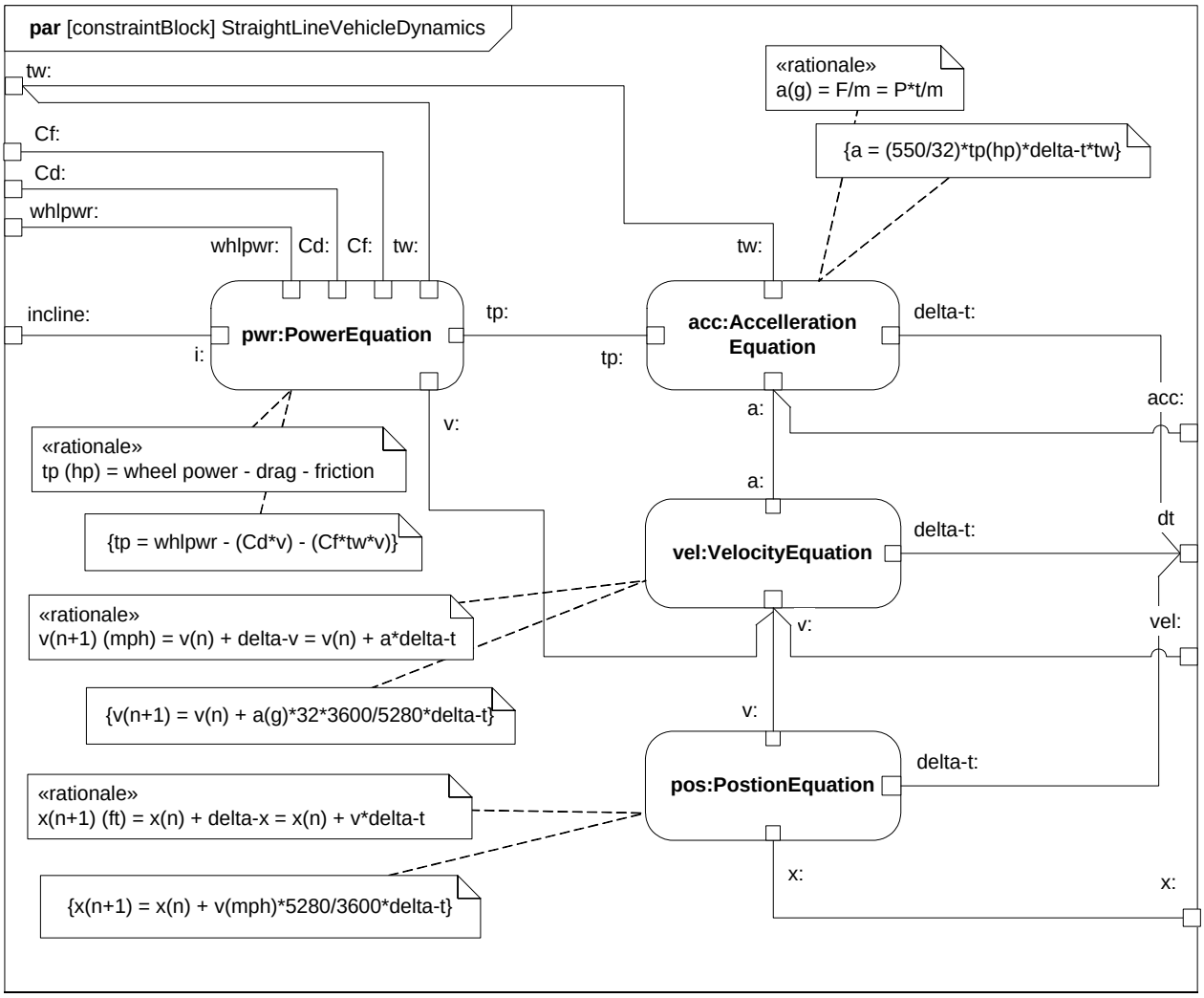


Figure B.30 - Straight Line Vehicle Dynamics Mathematical Model (Parametric Diagram)

The constraints and parameters in Figure B.30 are detailed in Figure B.31 in Block Definition Diagram format.

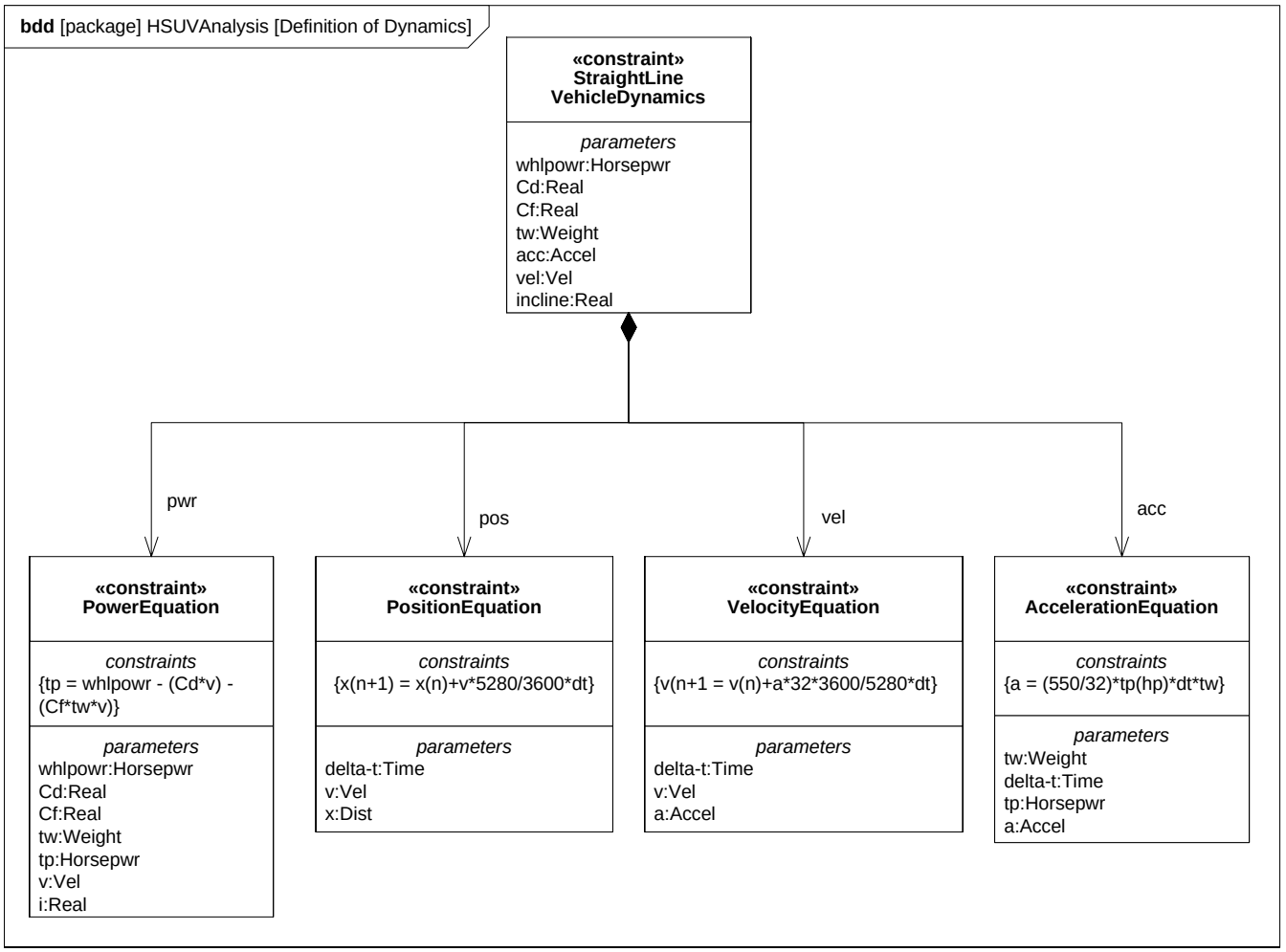


Figure B.31 - Defining Straight-Line Vehicle Dynamics Mathematical Constraints (Block Definition Diagram)

Note the use of valueTypes originally defined in Figure B.2.

B.4.7.6 (Non-Normative) Timing Diagram - 100hp Acceleration

Timing diagrams, while included in UML 2.1, are not directly supported by SysML. For illustration purposes, however, the interaction shown in Figure B.32 was generated based on the constraints and parameters of the StraightLineVehicleDynamics constraintBlock, as described in the Figure B.30. It assumes a constant 100hp at the drive wheels, 4000lb gross vehicle weight, and constant values for Cd and Cf.

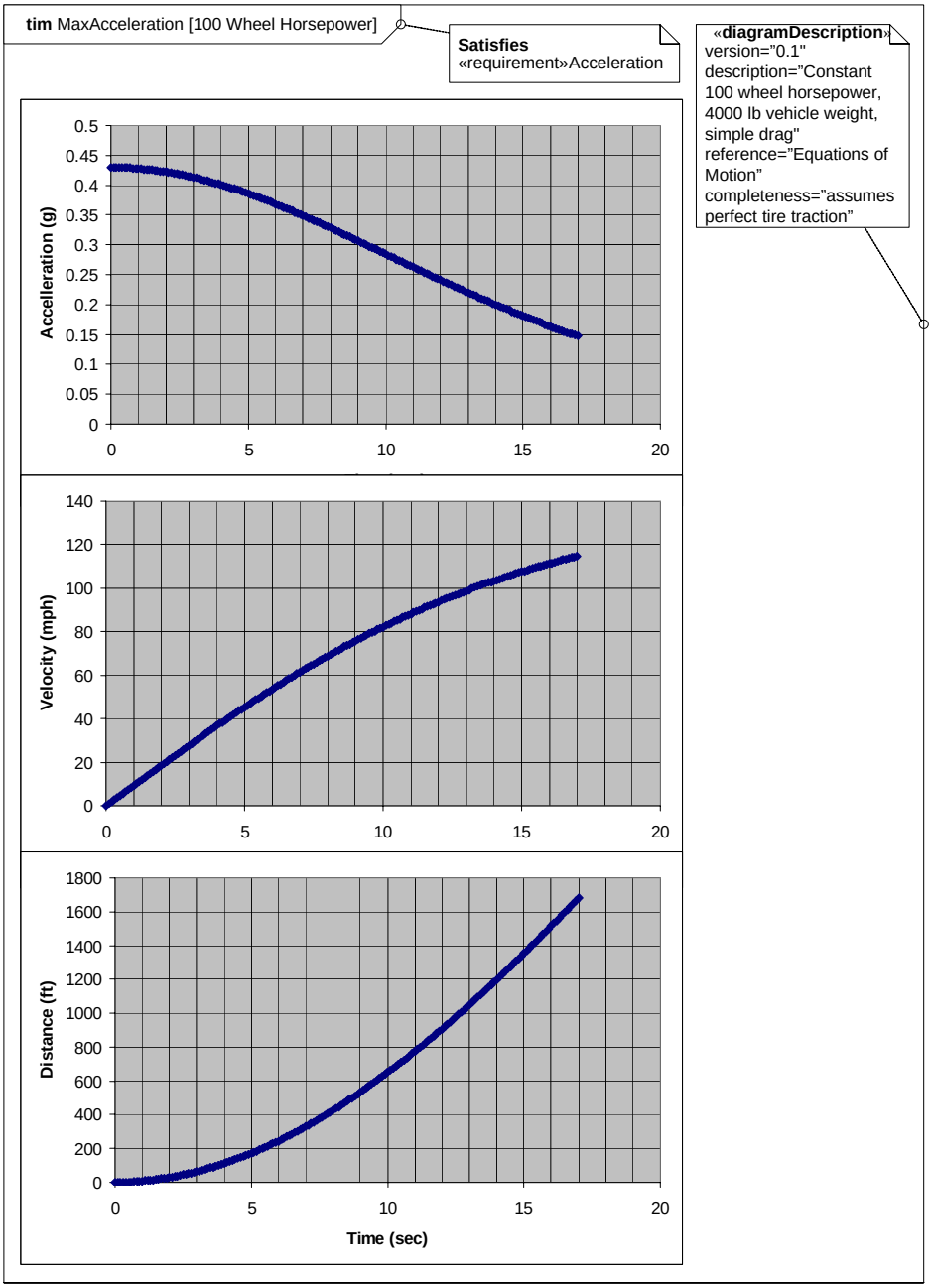


Figure B.32 - Results of Maximum Acceleration Analysis (Timing Diagram)

B.4.8 Defining, Decomposing, and Allocating Activities

B.4.8.1 Activity Diagram - Acceleration (top level)

Figure B.33 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.

.

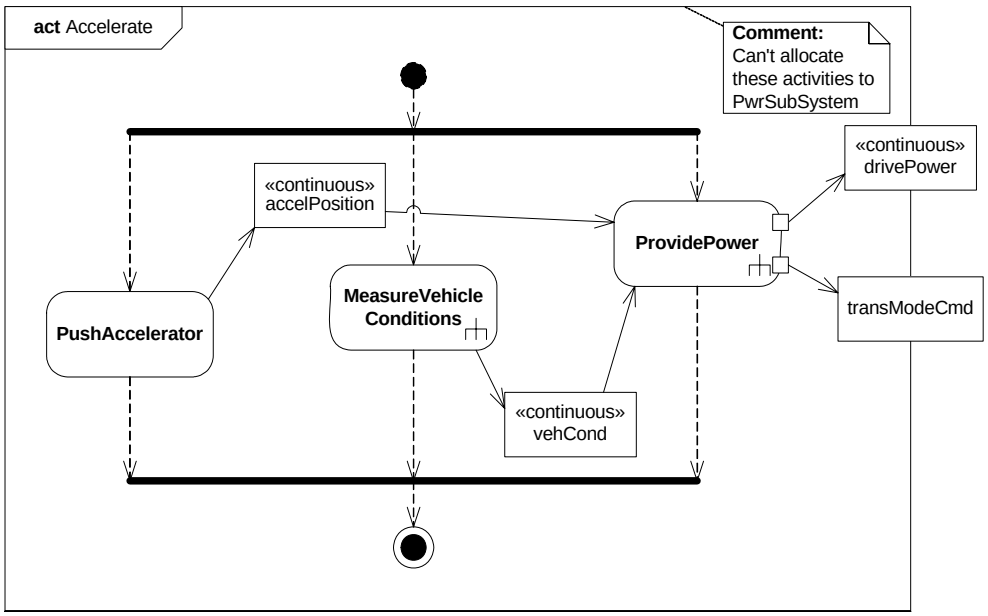


Figure B.33 - Behavior Model for "Accelerate" Function (Activity Diagram)

B.4.8.2 Block Definition Diagram - Acceleration

Figure B.34 defines a decomposition of the activities and objectFlows from the activity diagram in Figure B.33.

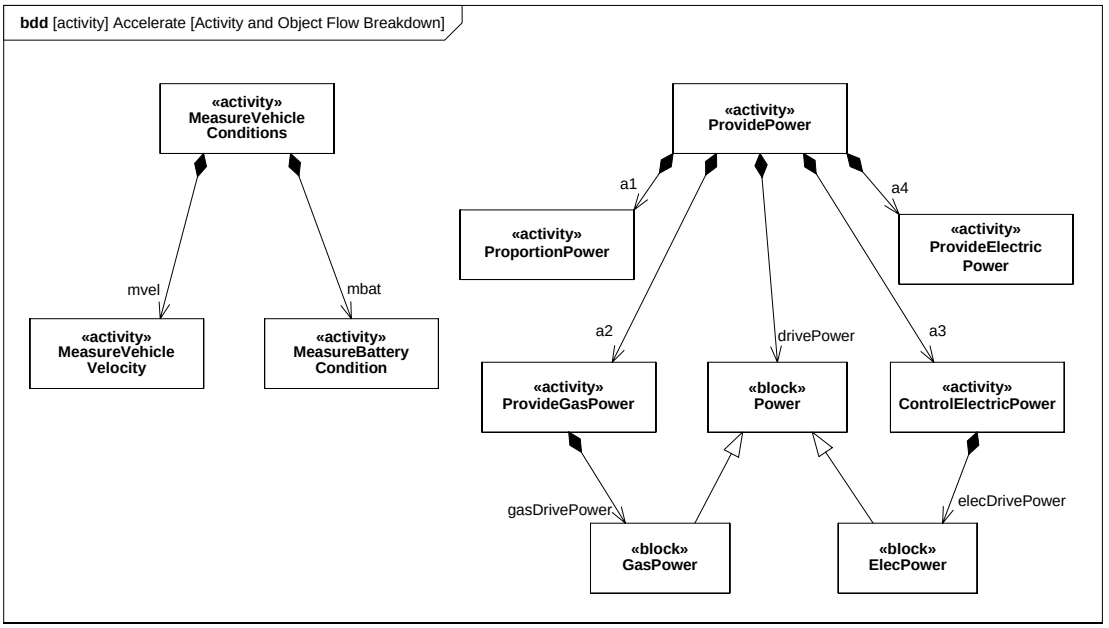


Figure B.34 - Decomposition of "Accelerate" Function (Block Definition diagram)

B.4.8.3 Activity Diagram (EFFBD) - Acceleration (detail)

Figure B.35 shows the ProvidePower activity, using the decomposed activities and objectFlows from Figure B.34. 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.

.

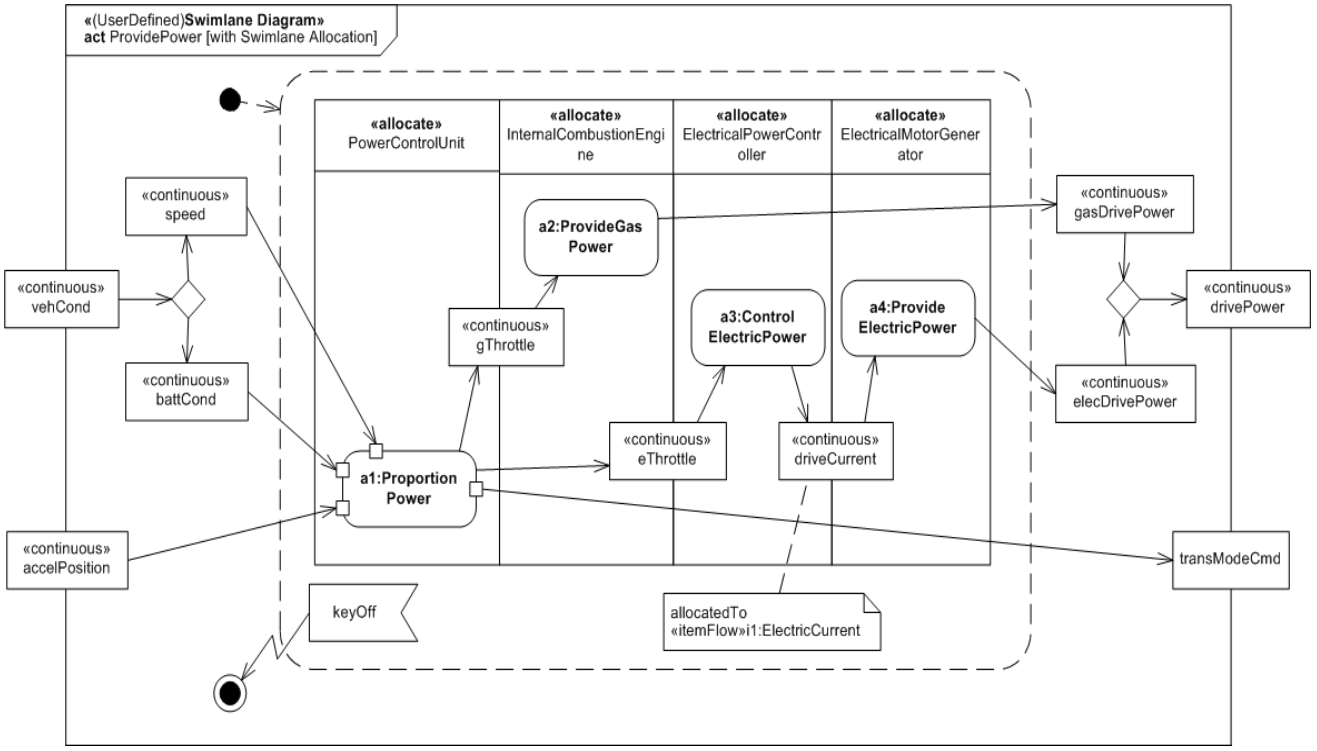


Figure B.35 - Detailed Behavior Model for "Provide Power" (Activity Diagram)
Note hierarchical consistency with Figure B.33.

B.4.8.4 Internal Block Diagram - Power Subsystem Behavioral and Flow Allocation

Figure B.36 depicts a subset of the PowerSubsystem, specifically showing the allocation relationships generated in Figure B.35.

.

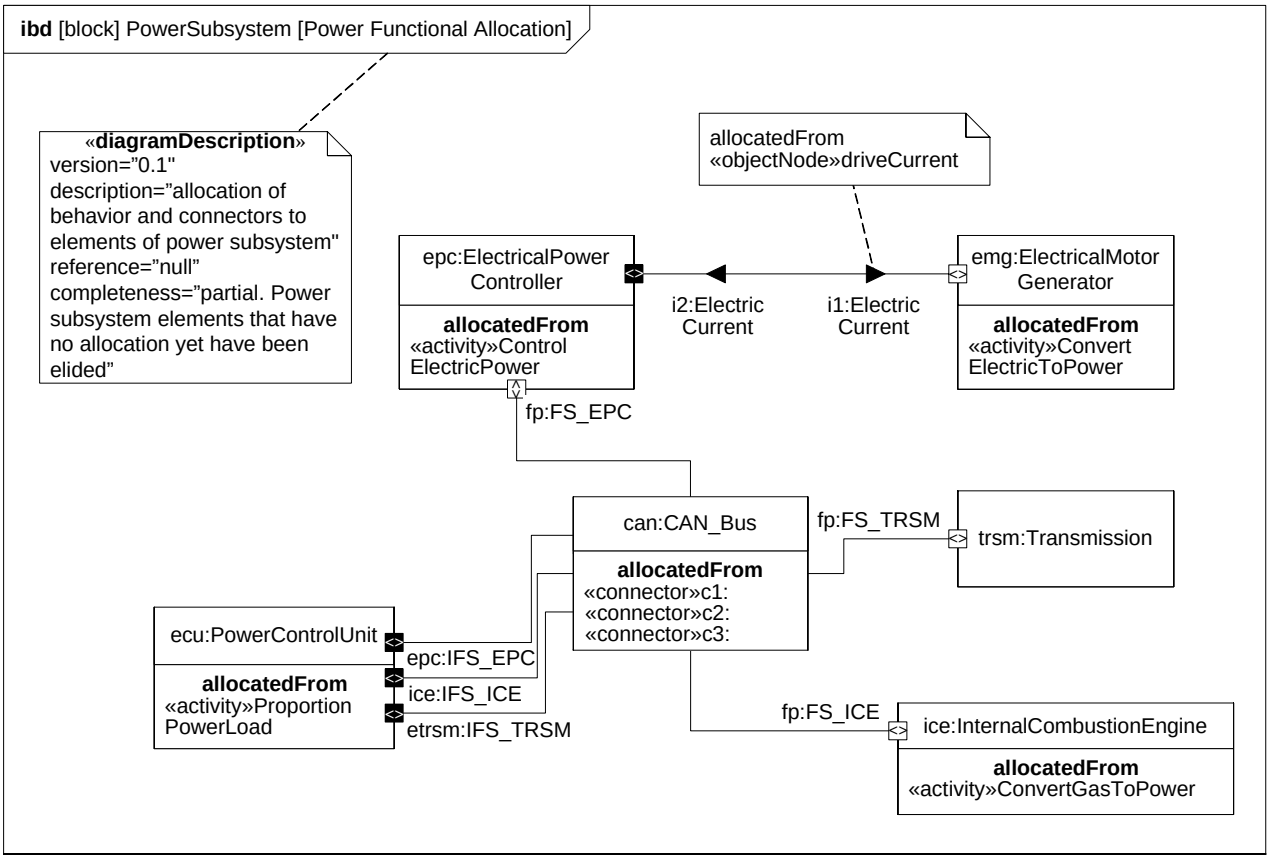


Figure B.36 - Flow Allocation to Power Subsystem (Internal Block Diagram)

B.4.8.5 Table - Acceleration Allocation

Figure B.37 shows the same allocation relationships shown in Figure B.36, but in a more compact tabular representation.

.

Table [activity] ProvidePower [Allocation Tree for Provide Power Activities]

type	name	end	relation	end	type	name
activity	a1:ProportionPower	from	allocate	to	block	PowerControlUnit
activity	a2:ProvideGasPower	from	allocate	to	block	InternalCombustionEngine
activity	a3:ControlElectricPower	from	allocate	to	block	ElectricalPowerController
activity	a4:ProvideElectricPower	from	allocate	to	block	ElectricalMotorGenerator
objectNode	driveCurrent	from	allocate	to	itemFlow	i1:ElectricCurrent

Figure B.37 - Tabular Representation of Allocation from "Accelerate" Behavior Model to Power Subsystem (Table)

B.4.8.6 Internal Block Diagram: Property Specific Values - EPA Fuel Economy Test

Figure B.38 shows a particular Hybrid SUV (VIN number) satisfying the EPA fuel economy test. Serial numbers of specific relevant parts are indicated.

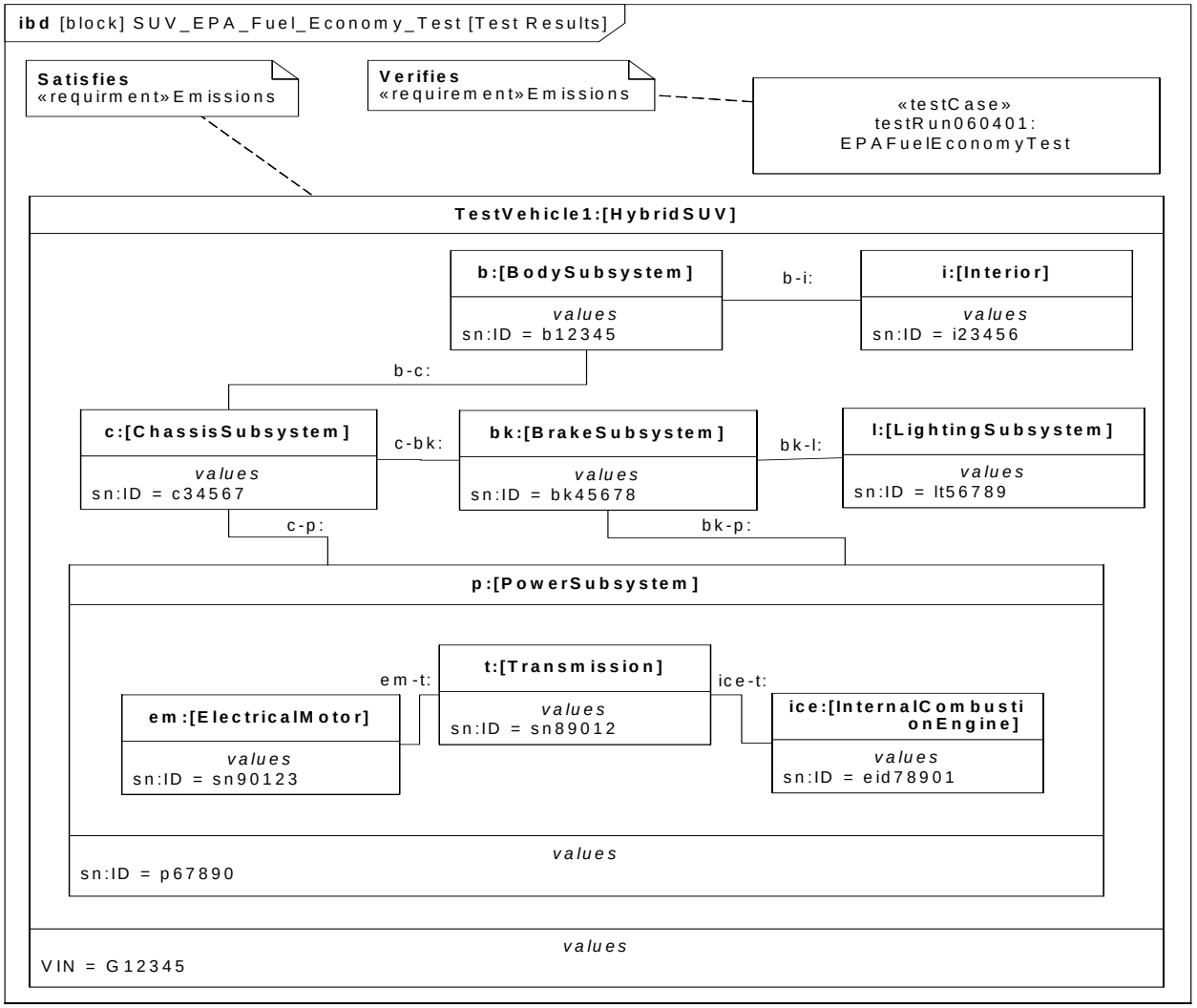


Figure B.38 - Special Case of Internal Block Diagram Showing Reference to Specific Properties (serial numbers)

Annex C: Non-normative Extensions

(informative)

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 specification is organized. Stereotypes in this section are specified using a tabular format, consistent with how non-normative stereotypes are specified in the UML 2.1 Superstructure specification. Model libraries are specified using the guidelines provided in the Profiles & Model Libraries chapter of this specification.

C.1 Activity Diagram Extensions

C.1.1 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].

C.1.2 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 C.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 must conform to the following constraints:

[1] (On Activity) Activities do not have partitions.

[2] (On Activity) All decisions, merges, joins and forks are well-nested. In particular, each decision and merge are 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 require exactly one control edge coming into them, and exactly one control edge coming out, except when using parameter sets.

[4] (Execution constraint) All control is enabling.

[5] (On ControlFlow) All control flows into an action target a pin on the action that has isControl = true.

[6] (On ObjectNode) Ordering is first-in first out, ordering = FIFO.

[7] (On ObjectNode) Object flow is never used for control, isControlType = false, except for pins of parameters in parameter sets.

[8] (On Parameter) Parameters take and produce no more than one item, multiplicity.upper =1.

[9] (On Parameter) Output parameters produce exactly one value, multiplicity.lower = 1. The «optional» stereotype cannot be applied to parameters.

[10] (On Parameter) Parameters cannot be streaming or exception.

[11] (On ParameterSet) Parameter sets only apply to output parameters.

[12] (On ParameterSet) Parameter sets only apply to control. Parameters in parameter sets must have pins with isControlType = true.

[13] (On ParameterSet) Parameter sets have exactly one parameter, and it must 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 behaviour or operation must be in parameter sets.

[15] (On ActivityEdge) Edges cannot have time constraints.

[16] The following SysML stereotypes cannot 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 C.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.

C.1.3 Stereotype Examples

Figure C.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 Section C.1.2, for example, that the data outputs on all functions are required and that queuing is FIF.

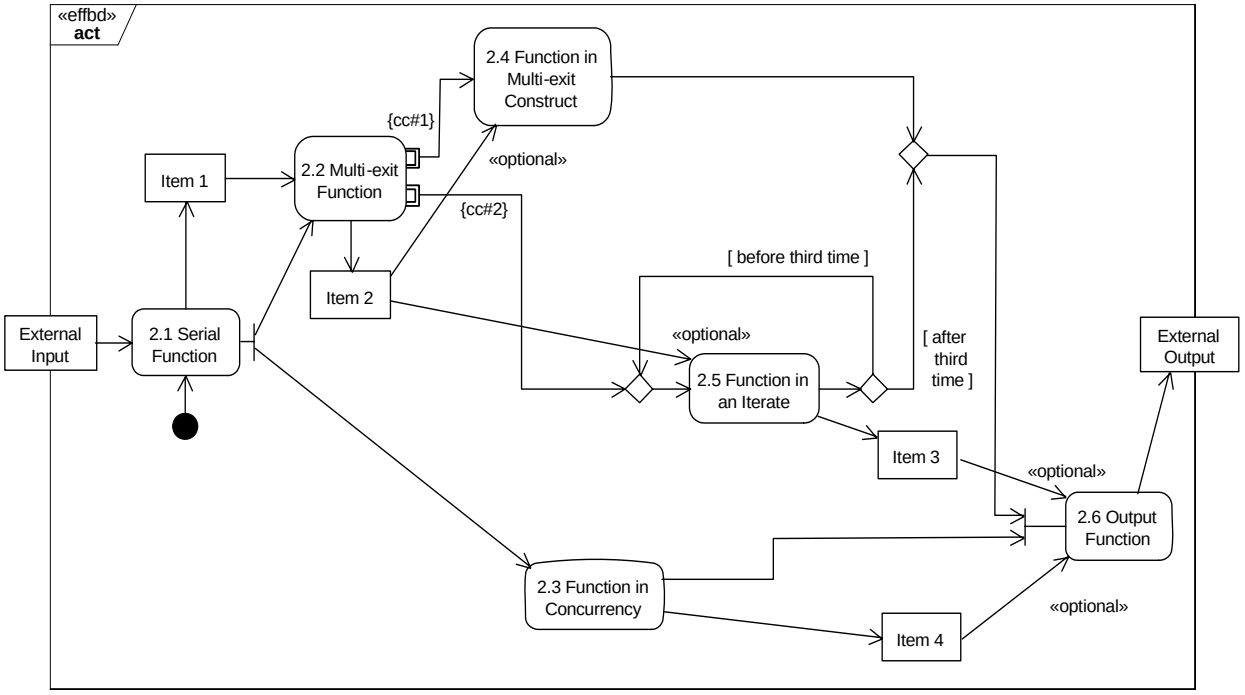


Figure C.1 - Example activity with *«effbd» stereotype applied*

Figure C.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.

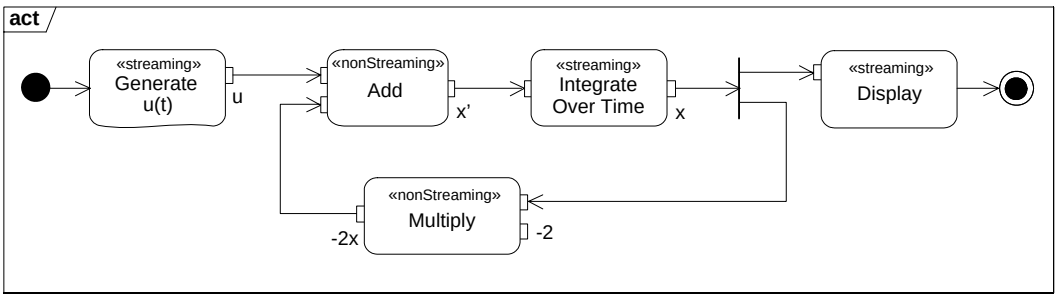


Figure C.2 - Example activity with *«streaming» and «nonStreaming» stereotypes applied to subactivities.*

C.2 Requirements Diagram Extensions

C.2.1 Overview

This section describe an example of a non-normative extension for a requirements profile.

C.2.2 Stereotypes

This section 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 C.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 C.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, connector, 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 quantitatively measures the extent to which a system, or a system part, satisfies a required capability or condition.

Table C.3 - Additional Requirement Stereotypes

Stereotype	Base Class	Properties	Constraints	Description
«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 C.4 provides the definition of the non-normative enumerations that are used to type properties of "extendedRequirement" stereotype of Figure C.3.

Table C.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.

C.2.3 Stereotype Examples

Figure C.3 shows the use of several sub-types 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.

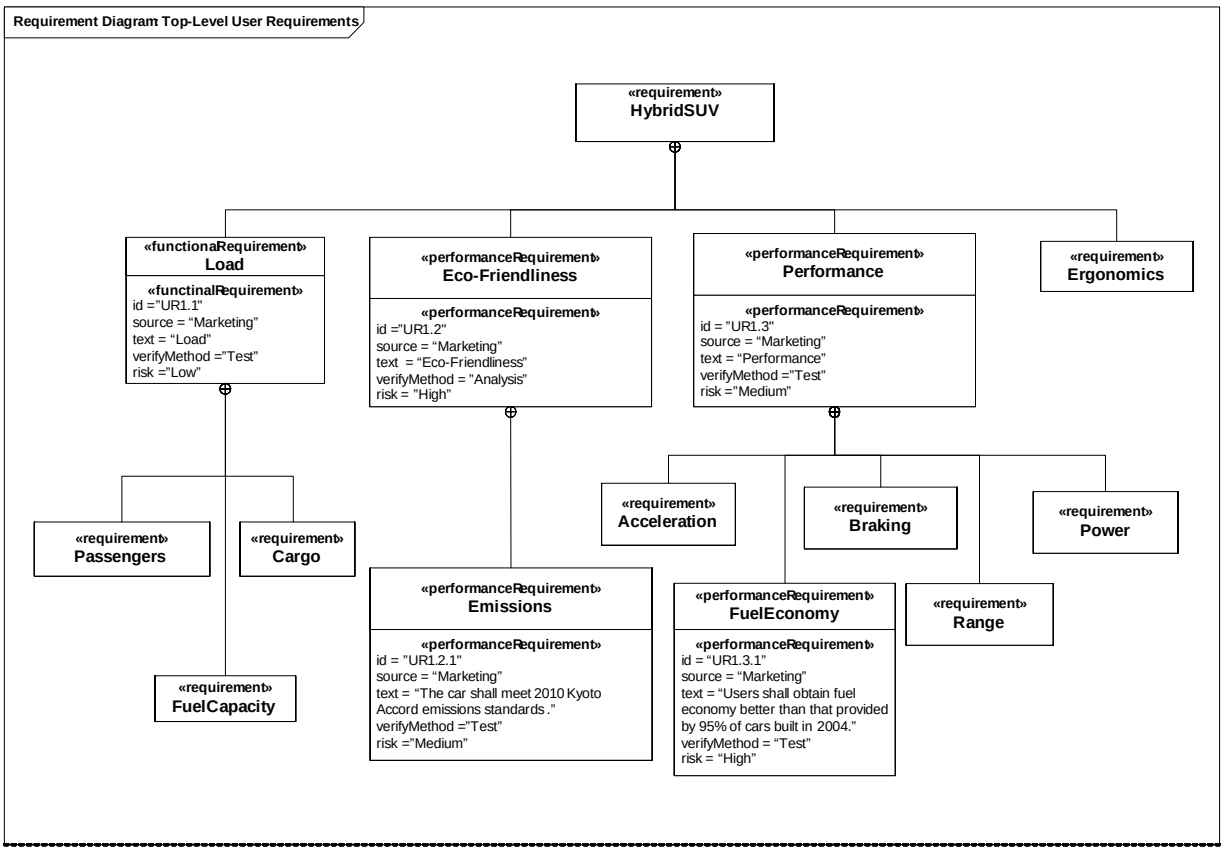


Figure C.3 - Example extensions to Requirement

C.3 Parametric Diagram Extensions for Trade Studies

C.3.1 Overview

This section describes a non-normative extension of a parametric diagram (refer to the Constraint Blocks chapter) 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 measures of effectiveness.

This section 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.

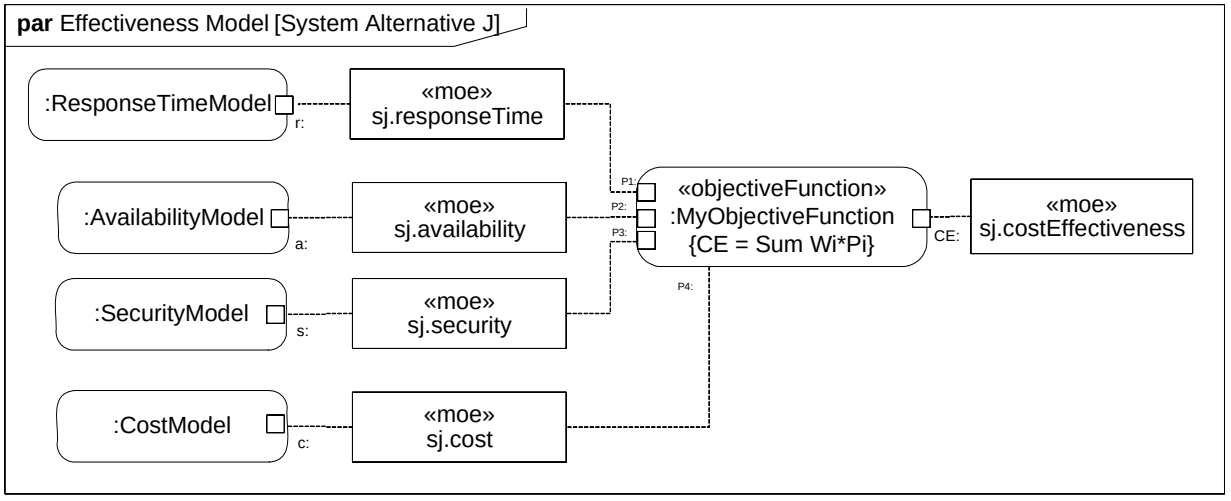
C.3.2 Stereotypes.

Table C.5 - Stereotypes for Measures of Effectiveness

Stereotype	Base Class	Properties	Constraints	Description
«objectiveFunction»	«ConstraintBlock» or «ConstraintProperty»	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»	«blockProperty»	N/A	N/A	A measure of effectiveness (moe) represents a parameter whose value is critical for achieving the desired mission cost effectiveness.

C.3.3 Stereotype Examples

In this example, operational availability, mission response time, and security effectiveness each represent moe's 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 avialability, mission response time, security effectiveness and life cycle cost to determine an overall cost effectiveness for each alternative. It is assumed that the moe's refer to the values for system alternative j (sj).



C.4 Model Library for Dimensions and Units

The dimensions and units in this section are a subset of units defined by the International System of Units (SI) as defined in NIST Special Publication 330 (available from the NIST Reference on Constants, Units and Uncertainty at http://physics.nist.gov/cuu/Units/units.html).

.

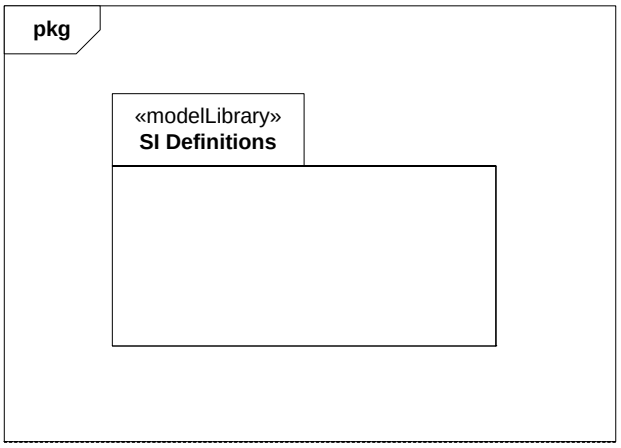


Figure C.4 - SI Definitions model library

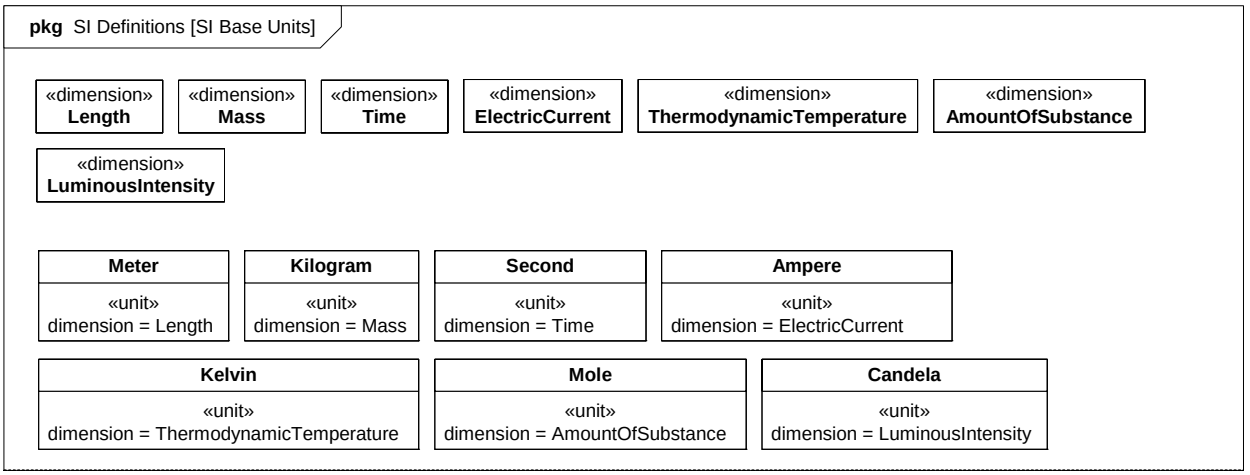


Figure C.5 - SI Base Units

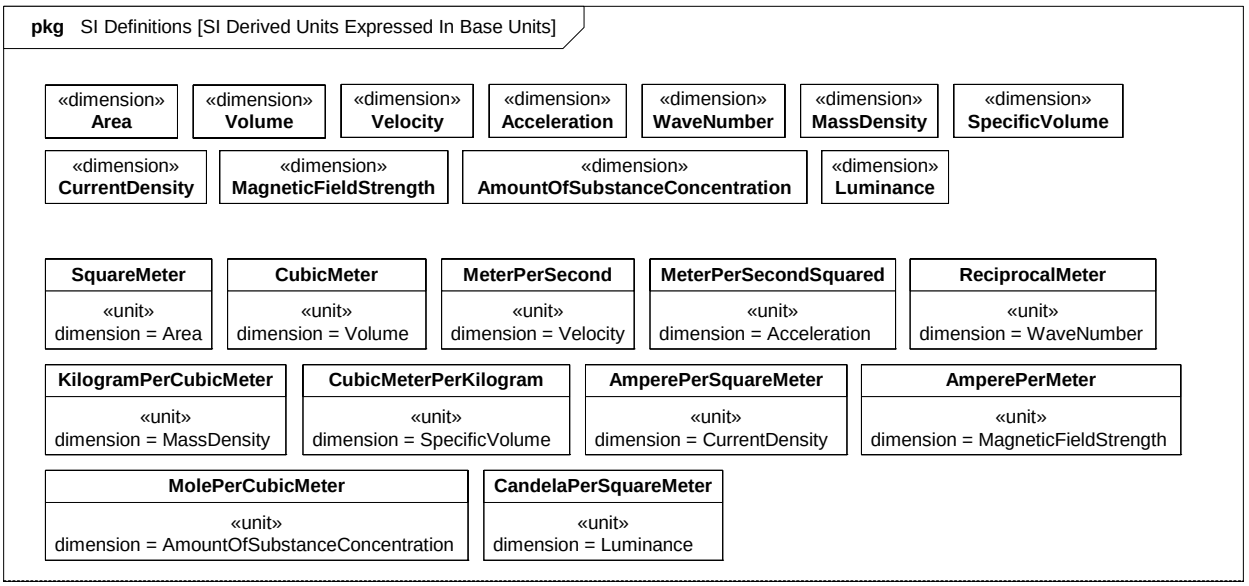


Figure C.6 - SI Derived Units Expressed In Base Units

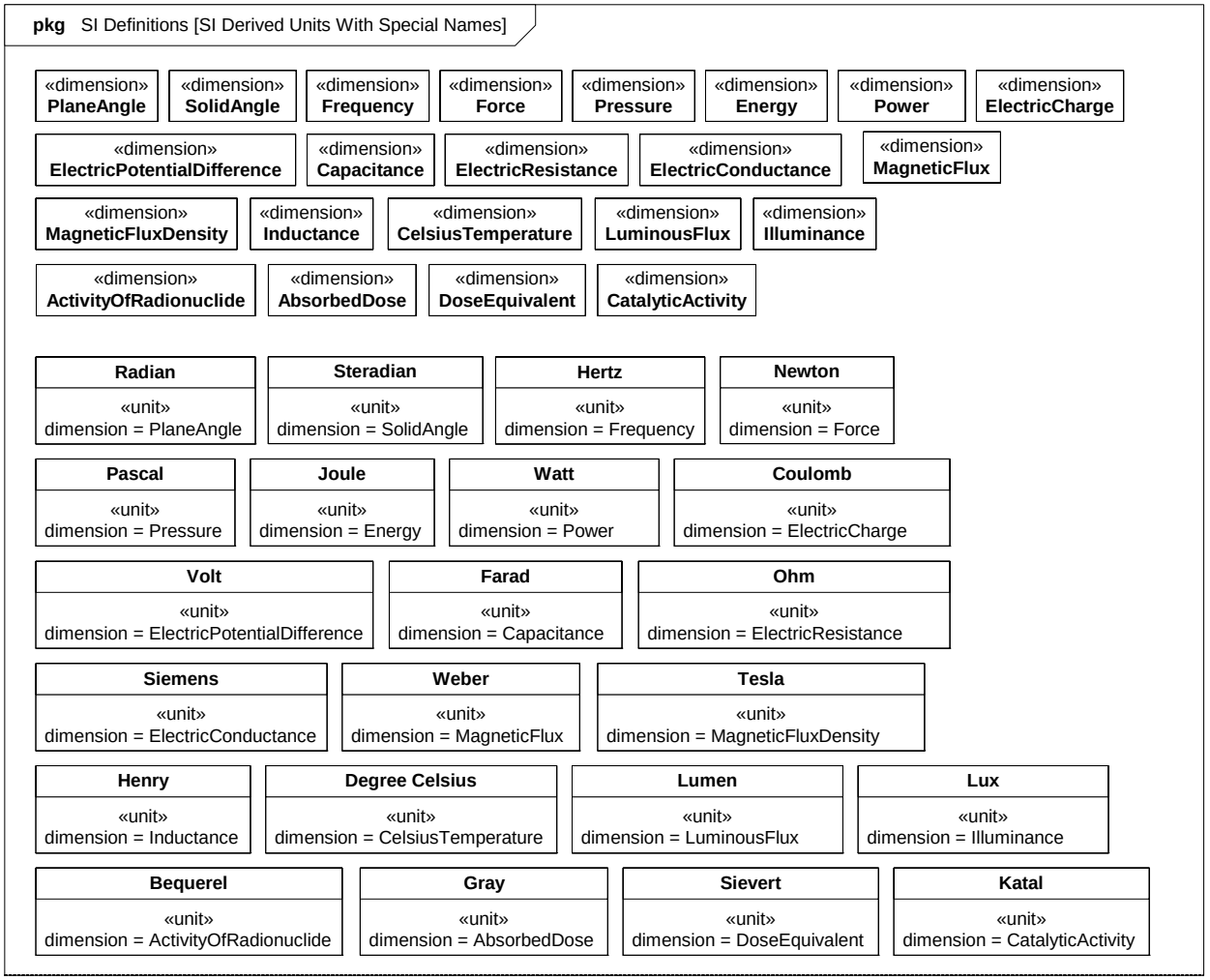


Figure C.7 - SI Derived Units With Special Names

C.5 Distribution Extensions

C.5.1 Overview

This section describes a non-normative extension to provide a candidate set of distributions (see "DistributedProperty" on page 48). It consists of a profile containing stereotypes that can be used to specify distributions for properties of blocks.

C.5.2 Stereotypes

Package Distributions

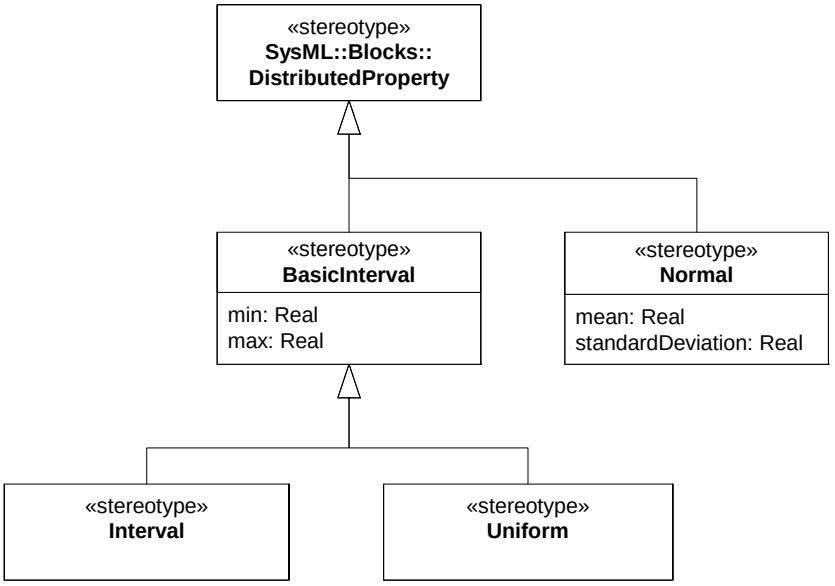


Figure C.8 - Basic distribution stereotypes

Table C.6 - 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 standardDeviation:Real	N/A	Normal distribution - constant probability between min and max

C.5.3 Usage Example

Figure C.9 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.

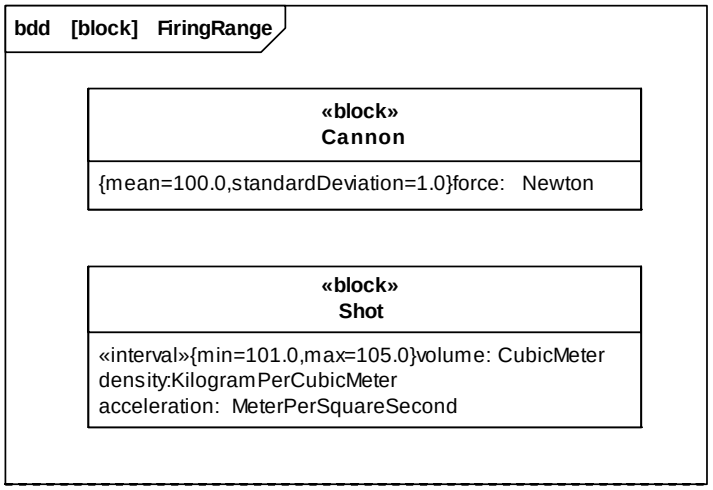


Figure C.9 - Distribution Example

Annex D: Model Interchange

(informative)

D.1 Overview

This annex describes several 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 and design (AP233), which is one of the series of STEP (Standard for the Exchange of Product Model Data) neutral data schemas for representing engineering data. Other model interchange approaches are possible, but the ones described in this annex are expected to be the primary ones supported by SysML.

D.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, bi-directional 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.

D.3 XMI Serialization of SysML

UML 2.0 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) 2.1 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 2.1 representation of the SysML profile (i.e., the UML Profile for SysML) is provided as a support document to this specification (refer to ad/2006-03-02). 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.

D.4 Overview of AP233

AP233 is not finalized at this time, so this section reflects the background and current status of the AP233 work.

AP233 is a neutral data schema for representing systems engineering data. AP233 is being standardized under the ISO TC-184 (Technical Committee on Industrial Automation Systems and Integration), SC4 (Subcommittee on Industrial Data Standards), and is part of the larger STEP effort, which provides standardized models and infrastructure for the exchange of product model data.

D.4.1 Scope of AP233

AP233 will include support for describing:

- requirement
- functional
- structure
- physical structure & allocation
- configuration & traceability
- project & data management

An IDEF activity that shows the scope of AP233 information requirements is available at http://public.ap233.org/AAM/AAM_AP233-Issue-1.pdf. Additional details on AP233 can be found at http://public.ap233.org/.

D.4.2 AP233 Development Approach & Status

AP233 and several other STEP *application protocols* are being built using a modular architecture. This enables the same information model to be reused across disciplines and life cycle stages. In the STEP Modular Architecture these reusable information models are called *application modules*, or more informally simply modules. AP233 will consist of a number of modules that together will satisfy the scope of the requirements stated above. Support for several of systems engineering viewpoints within the scope of AP233 already exist as the result of the development of other application protocols and will simply be reused in AP233. When existing STEP modules do not provide needed capabilities, new modules are being defined as part of AP233 development. Since AP233 is part of STEP, it is easy to relate systems engineering data to that of other engineering disciplines over the lifecycle of a system and to related product models.

Figure D-1 provides an overview of the modules planned to satisfy the scope of AP233 requirements and also shows the current status of each.

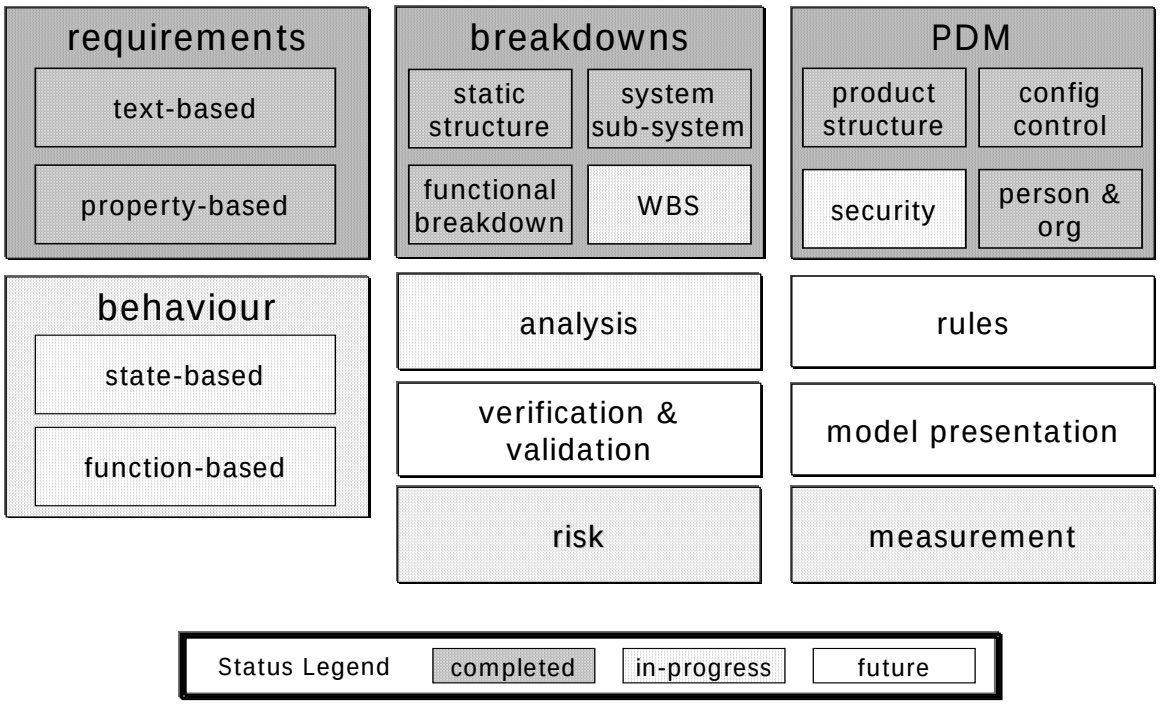


Figure D.1 - AP233 Modules

D.4.3 STEP Architecture, Modeling & Model Interchange Mechanisms

A good understanding of the STEP architecture and its components are required to understand how SysML models will be interchanged using AP233. This section provides an overview of the key elements of STEP that pertain to model interchange.

D.4.3.1 Modular Architecture

The scope of STEP is very large. While a number of STEP modules and *application protocols* have been developed (e.g., product data management, geometry, structural, electrical, and other engineering analysis support) and in use for several years, other area such as AP233 are still being defined and developed.

AP233 and several other STEP application protocols are being built using a modular architecture. This enables the same information model to be reused across disciplines and life cycle stages. In the STEP Modular Architecture these reusable information models are called *application modules*, or more informally simply modules.

For more detail on the STEP architecture see the ISO TC184/SC4 Industrial Data subcommittee web page at http://www.tc184-sc4.org/: and for a more detailed view of where specific STEP parts fit into the architecture is available at http://www.mel.nist.gov/sc5/soap/soapgrf030407.pdf.

D.4.3.2 The Modeling Language for AP233

AP233, like all STEP application protocols, is defined using the EXPRESS modeling language (see ISO 10303-11 Description method: The EXPRESS language reference manual). 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 people;
TYPE year = integer;
END_TYPE;
TYPE person_or_organization = SELECT ( person, organization );
END_TYPE;
ENTITY organization;
name : STRING;
END_ENTITY;
ENTITY building;
address : STRING;
owner : person_or_organization;
END_ENTITY;
ENTITY person
ABSTRACT SUPERTYPE;
spouse : OPTIONAL person;
name : STRING;
birthyear : year;
biological_parents : SET[2:2] of person;
parents : SET[2:?] of person;
END_ENTITY;
ENTITY man
SUBTYPE OF ( person );
sister : SET[0:?] of woman;
END_ENTITY;
ENTITY woman
SUBTYPE OF ( person );
brother : SET[0:?] of man;
END_ENTITY;
END_SCHEMA;
```

An overview of an XML Document Type Definition for the EXPRESS language is available at http://stepmod.sourceforge.net/express_model_spec/. Note however, that the powerful expression language for constraint writing is not addressed by that DTD. EXPRESS expressions are similar in nature to OCL expressions and the two languages have similar expressiveness.

Work is underway to produce and standardize a MOF-based EXPRESS metamodel and EXPRESS/UML mappings. Documentation related to those efforts is available at the exff (Engineering eXchange For Free) web site (http://www.exff.org/express_uml/index.html). Eventually these efforts should allow a formal SysML/AP233 relationship to be standardized within the OMG.

An early draft of one mapping of ISO EXPRESS to UML/XMI is available as an OMG document at http://www.omg.org/cgi-bin/doc?liaison/2003-07-01. Please note that this specification is based on EXPRESS Edition 1, UML 1.4, MOF 1.4 and XMI 1.2.

D.4.3.3 Model Interchange Mechanisms

As part of the STEP series of EXPRESS-based information model, a series of *implementation methods* are also standardized:

- ISO 10303-21 (Part 21), clear text encoding of the exchange structure
- ISO 10303-22 (Part 22), standard data access interface (SDAI) specification
- ISO 10303-25 (Part 25), EXPRESS to OMG XMI binding
- ISO 10303-28 (Part 28), XML representation of EXPRESS schemas and data

A conforming STEP implementation is the combination of a STEP application protocol and one or more of the implementation methods.

SDAI specifies a standard programming interface for access to EXPRESS-based data. SDAI allows the implementors to refer to product data in terms of its conceptual EXPRESS definitions, regardless of the underlying data structure or storage technology. Bindings of the SDAI to C++ (ISO 10303-23), C (ISO 10303-24), Java (ISO 10303-27) provide standardized APIs for accessing EXPRESS-based data.

D.4.4 AP233 - SysML Alignment & Mapping Model

The 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. However there might be differences in breath and scope of AP233 and SysML resulting from the different development life cycles of both activities and the different nature of the modeling frameworks used to define SysML and AP233. To avoid semantical issues in exchanging data between SysML and AP233, a neutral or mapping model of systems engineering concepts will be defined. Thus the mappings between the mapping model and SysML metamodel and the mapping model and AP233 metamodel can be maintained independently. The neutral mapping model will also help to clarify the semantics of the data elements. This approach is illustrated in Figure D-2.

As AP233 and SysML are defined in different modeling frameworks, the AP233 metamodel will be converted to UML to ease the mapping. OMG has started a standardization activity has been started to capture EXPRESS semantics in UML, but a custom mapping will be used until the UML profile for EXPRESS has been adopted. The mapping model will be expressed as a plain MOF model. The mapping model will be defined based on the concepts used and implemented for AP233 and SysML. Another important input is the conceptual systems engineering model maintained by the INCOSE Model Driven System Design Working group. Since development of the mapping model and SysML and AP233 mappings to it is an ongoing maintenance activity, these specifications will be maintained separately and updates will be posted on the SE DSIG web site.

The mapping model can be used as the basis for the models exchange methods discussed in the next section and also for the development of conceptual level API’s, which should ease the usage of AP233 and generation of common test cases for SysML and AP233.

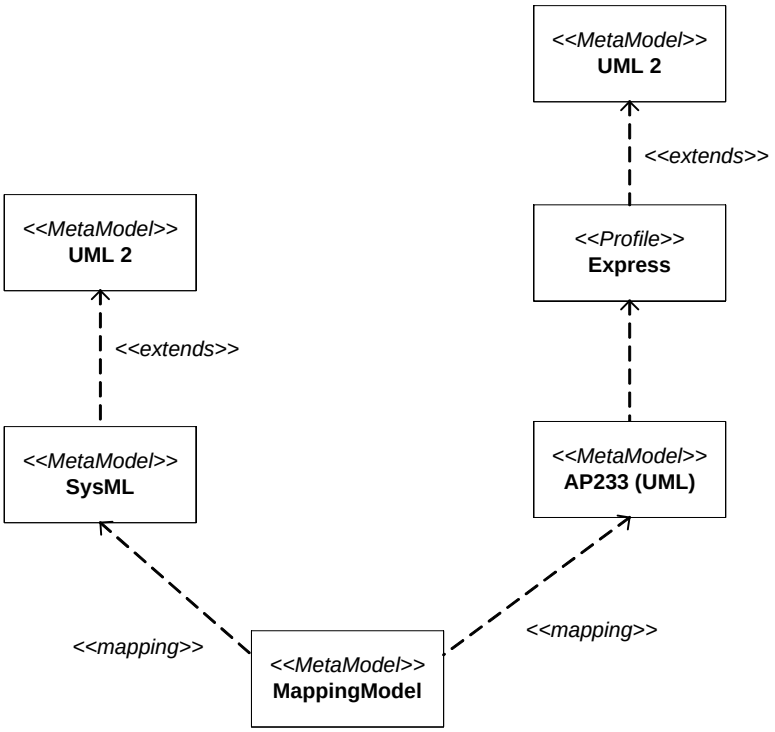


Figure D.2 - Mapping Model

D.4.5 Generic Procedures for SysML and AP233 Model Interchange

D.4.5.1 File-based Exchange

Industrial-strength STEP implementations are typically file exchange-based systems integration processes. As OMG has standardized XMI as its model serialization format, one obvious approach is to use the STEP XML-based file exchange capability (Part 28) by simply translating the model contained in an XMI file into a model based on the AP233 XML Schema. This approach encourages systems integrators and SysML tool vendors to develop interoperable SysML-AP233 exchange capabilities. It is also provides SysML tool vendors with a means to directly export AP233 XML files.

D.4.5.2 API-Driven Model Interchange

Model interchange can be simplified by the use of high-level application program interfaces (APIs) . At the moment, standardized APIs for SysML- or AP233-specific models are not available, but work is underway in the industry to provide implementations of such APIs. Ideally, application level developers can use the same APIs to access backend XML models serialized in either SysML XMI or AP233 XML format, depending on customer needs. When combined, standardized XML serialization formats and high-level APIs will provide a very convenient and interoperable way for SysML tool vendors and systems integrators to exchange SysML and AP233 models. These standardized capabilities will also provide the foundation needed for building a set of Systems Engineering Web Services.

Annex E: Requirements Traceability

(informative)

This annex describes the requirements tracability matrix (RTM) that shows how SysML satisfies the requirements in Sections 6.5 (Mandatory) and 6.6 (Optional) of the UML for SE RFP. The matrix includes columns that correspond to those identified in the first paragraph of Section 6.5 of the RFP and are restated here. The text requirement statement is included in the RFP and was excluded from this annex due to space limitations.

a) The UML for SE requirement number.

b) The UML for SE requirement name (or other letter designator). Note: The reader should refer to the UML for SE RFP for the specific text of the requirement, since there was inadequate room in the table to repeat it here.

c) Describes whether the proposed solution is a full or partial satisfaction of the requirement, or whether there is no solution provided. The section header rows that do not have a text requirement are marked N/A.

d) A description of how SysML addresses the requirement. Note: In some cases, there may be other SysML solutions to satisfy the requirement, but the intent was to describe at least one solution.

e) The specific UML and SysML metaclasses that address the requirement.

f) Reference to the applicable chapter in the SysML specification which addresses e) above. This diagram element tables in the chapter describe the concrete syntax (symbols) that show how the solution to the requirement is represented in diagrams. The usage examples in the chapters along with sample problem in "Annex B: Sample Problem" describe how the solution to the requirement is used in representative examples. Note: The reference to a chapter may require reference to a corresponding chapter in the UML specification. For example, when the blocks chapter is referenced, this may include a combination of the SysML blocks chapter and the UML classes and composite structure chapters.

Table E.1 - Requirement Traceability matrix

UML for SE Req't #	Requirement name	Compl (Y/N, Partial)	Requirement Satisfaction	Metaclass Extension	SysML Diagram Chapter	Ver #
6.5	*Mandatory Requirements*					
6.5.1	*Structure*	N/A	*Structure diagrams include block definition, internal block, and package diagrams*		Structural Constructs	
6.5.1.1	*System hierarchy*	Y	*Block composition (black or white diamond) in a block definition diagram and parts in internal block diagrams are the primary mechanisms for representing system hierarchy.*	*SysML::Block, UML::Association, SysML::Block Property*	*Blocks*	1.0

	a. Subsystem (logical or physical)	*Y*	*Typically represented by a set of logical or physical parts in an internal block diagram that realize one or more system operations. The corresponding sequence diagram and activity diagram with swim lanes can represent a hybrid of structure and behavior.*	SysML::Block, SysML::Block Property	Blocks	1.0
	b. Hardware (i.e., electrical, mechanical, optical)	*Y*	*Represented by a block or part.*	SysML::Block, SysML::Block Property	Blocks	1.0
	c. Software	*Y*	*Represented by a block or part or a UML component.*	SysML::Block, SysML::Block Property, UML::Component	Blocks	1.0
	d. Data	Y	*Represented by a block or part. Refer to input/output requirements in 6.5.2.1.1 and 6.5.2.5 for data flows.*	SysML::Block, SysML::Block Property, SysML::ValueType, UML::DataType	Blocks	1.0
	e. Manual procedure	*Y*	*Represented by a block or part. Can also be represented by the standard UML stereotype <<document>>.*	SysML::Block, SysML::Block Property, UML::Document	Blocks	1.0
	f. User/person	*Y*	*Represented by a block or part. External users are also represented as actors in a use case diagram.*	SysML::Block, SysML::Block Property	Blocks	1.0
	g. Facility	*Y*	*Represented by a block or part.*	SysML::Block, SysML::Block Property	Blocks	1.0
	h. Natural object	*Y*	*Represented by a block or part.*	SysML::Block, SysML::Block Property	Blocks	1.0
	i. Node	Y	*Represented by a block or part.*	*SysML::Block, SysML::Block Property*	Blocks	1.0
6.5.1.2	*Environment*	Y	*Environment is one or more entities that are external to the system of interest and can be represented as a block or part of a broader context. Also, represented as actors in use cases.*	*SysML::Block, SysML::Block Property*	*Blocks,* *Use Case*	1.0

6.5.1.3	*System inter-connection*		*Internal block diagram shows connections using parts, ports, and connectors.*	*SysML::Block, SysML::Block Property, UML Association, UML::Connector: SysML::Nested ConnectorEnd*	*Blocks*	1.0
6.5.1.3.1	*Port*	Y	*A port defines an interaction point on a block or part that enables the user to specify what can flow in/out of the block/part (flow port) or what services the block/part requires or provides (Standard Port). Ports are connected using connectors.*	*SysML::Standard Port,* *UML::Interface,* *SysML::FlowPort, SysML::Flow Specification, SysML::Flow Property*	*Ports and Flows*	1.0
6.5.1.3.2	*System boundary*	Y	*The enclosing block for an internal block diagram and its ports.*	*SysML::Block* *SysML::Standard Port,* *SysML::FlowPort*	*Blocks, Ports and Flows*	1.0
6.5.1.3.3	*Connection*	Y	*A connector binds two ports to support interconnection. A connector can be typed by an association. A logical connector can be allocated to a more complex physical path depicting a set of parts, ports, and connectors (refer to allocation). Note: A connector has limited decomposition capability at this time.*	*UML::Association, UML::Connector, SysML::Nested ConnectorEnd*	*Blocks*	1.0
6.5.1.4	*Deployment of components to nodes*	Y	*A structural allocation relationship enables the allocation (deployment) of one structural element to another.*	*SysML::Allocation, SysML::Allocated, UML::Named Element*	*Allocations*	1.0
	a.	Y	*Software part, block or component deployed to a hardware part or block (processor or storage device).*	SysML::Allocation, SysML::Allocated, SysML::Block, SysML::Block Property, UML::Component	Allocations	1.0
	b.	Y	*Generalized deployment relationship between a deployed element and its host.*	SysML::Allocation, SysML::Allocated, SysML::Block, SysML::Block Property	Allocations	1.0

	c	Y	*Deployed element and host can be decomposed using blocks and parts.*	SysML::Block, SysML::Block Property	Allocations	1.0
6.5.2	*Behavior*	N/A	*Behavior diagrams include activity, sequence, and state machine diagrams. Communication diagrams, interaction overview diagrams, and timing diagrams are interaction diagrams that are not included in SysML. Use case diagrams are also viewed as a behavior diagram in that they represent the functionality in terms of the usages of the system, but do not depict temporal relationships and associated control flow or input/output flow.*		Behavioral Constructs	
6.5.2.1	*Functional Transformation of Inputs to Outputs*		*A behavior is the generalized form of a function with inputs and output parameters. Activity is a subclass of behavior.*	*UML::Behavior*	Activities	
6.5.2.1.1	*Input/Output*	Y	*Inputs and outputs can be represented as parameters of activities, object nodes flowing between action nodes, and as item flows between parts in an internal block diagram. Note: Object nodes are more precisely represented by pins on action nodes.*	*UML::Parameter, UML::ObjectNode, SysML::ItemFlow*	*Activities, Ports and Flows*	1.0
	a	Y	*Parameters, object nodes, and item properties are typed by classifiers (blocks or value types) that can have properties.*	*SysML::Block, UML::Parameter, UML::ObjectNode, SysML::ItemFlow*	Activities, Ports and Flows	1.0
	b	Y	*The classifiers that represent the things that flow (type of parameter, object node, and item property) can be decomposed and specialized.*	*SysML::Block, UML::Parameter, UML::ObjectNode, SysML::ItemFlow*	Activities, Ports and Flows, Blocks	1.0
	c	Y	*"ItemFlows" associate the things that flow with the connectors that bind the ports. The parameters and object nodes are bound to the corresponding activities and actions.*	*SysML::Block, UML::Parameter, UML::ObjectNode, SysML::ItemFlow*	*Activities, Ports and Flows*	1.0

6.5.2.1.2	*System store*	Partial	*Stored items can be represented as parts of a block, and also represented in an activity diagram as object nodes or central buffer nodes.*	*SysML::Block, SysML::Block Property,* *UML::ObjectNode* *UML::Central BufferNode*	*Blocks, Activities*	1.0
	a	Partial	*Object nodes in an activity diagram can represent depletable stores, and a data store node can represent non-depletable stores.*	*UML::ObjectNode, UML::DataStore Node*	Activities	1.0
	b	Y	*A stored item can be the same type of classifier as an input or output in both an internal block diagram and an activity diagram. The classifier supports different roles (store vs. flow).*	*SysML::Block, SysML::Block Property, UML::ObjectNode, UML::DataStore Node*	Blocks, Activities	1.0
6.5.2.1.3	*Function*	Y	*Activity specifies a generic subclass of behavior that is used to represent a function definition in activity diagrams, sequence diagrams, and state-machine diagrams. Activities contain CallBehaviorActions that call (invoke) other activities to support execution of the generic behaviors.*	*UML::Activity*	Activities, Interactions, State Machines	1.0
	a	Y	*Behaviors and the associated parameters are named (i.e., name of activity and activity parameter node).*	*UML::Behavior*	*Activities, Interactions* *State Machines*	1.0
	b	Y	*The action semantics define different types of actions that include CreateObject, DestroyObject, ReadStructuralFeature (monitor), and WriteStructurealFeature (update). A CallBehavior action is a generalized action that can call any behavior (activity, interaction, state).*	*UML::CreateObject Action, UML::DeleteObject Action, the various object modification actions in UML, monitoring with UML::AcceptEvent Action*	*Activities, Interactions* *State Machines*	1.0
	c	Y	*The object nodes (pins) bind input and output parameters to actions.*	*UML::ObjectNode, UML::Pin*	*Activities*	1.0

	d	Y	*The queuing semantics for object nodes are specified. The default queuing is FIFO, but other forms of queuing including LIFO, ordered, and unordered as defined by the enumeration for ObjectNodeKind.*	*UML::Behavior, SysML::InputPin, SysML::ObjectNode*	*Activities*	1.0
	e	Partial	*Resource constraints to support an execution can be specified by Preconditions and PostConditions. The constraints can apply to resources that are generated, consumed, produced, and released, such as inputs and outputs, or the availability of memory or CPU. The constraints imposed on the resources can be further modeled using parametric diagrams.*	*UML::Constraint,* *SysML::Constraint Block*	*Activities, Constraint Blocks*	1.0
	f	Y	*Refer to c*	*UML::ObjectFlow, UML::Pin*	*Activities*	1.0
	g.	Y	*An activity can be decomposed into lower level actions that invoke other activities.*	*UML::Activity, UML::CallBehavior Action, UML::Activity ParameterNode, UML::ObjectFlow, UML:: Pin*	*Activities*	1.0
	h.	Y	*An action has control inputs that can enable the execution of a function, and a control value input from a control operator that can both enable or disable an execution of a function. An execution of a function can also be terminated when it is enclosed in an interruptible region. Alternatively, state machine diagrams can be used to enable or disable execution upon transition events.*	*UML::Action, UML::Interruptible ActivityRegion, SysML::ControlValue UML::State*	*Activities, State Machines*	1.0

	i	Y	*A computational expression can be used to specify the behavior (i.e. activity) that is invoked by an action or an action that represents a primitive function such as an arithmetic expression. Specific math expressions may be included in a math model library. The expressions should be represented in a formal mathematical language and specify the language if they are to be interpreted by a computational engine.*	*UML::Activity, UML::Action*	*Activities, Interactions* *State Machines*	1.0
	j	Y	*A continuous or discrete rate stereotype can be applied to inputs and outputs. Inputs and outputs are discrete by default. A time continuous input or output is an input or output whose value can change in infinitely small increments of time. An activity can accept the continuous inputs and provide continuous outputs while executing if the inputs and outputs are also streaming. An alternative approach is to continuously invoke an activity that does not have streaming inputs or outputs, in which case each execution of an activity accepts the inputs at the start of execution and produces the output at the completion of execution.*	*SysML::Rate* *SysML::Continous, SysML::Discrete* *UML::Parameter (isStream=Value)*	*Activities, State Machines*	1.0
	k	Partial	Different actions can invoke concurrent executions of the same generalized behavior. Actions can have multiplicity.	*UML::Behavior, UML::Action*	Activities	1.0
6.5.2.2	*Function activation/ deactivation*	*N/A*	*Actions can be activated and deactivated using multiple mechanisms within SysML as described below including control flows, control operators, and interruptible regions.*		*Activities, Interactions* *State Machines*	1.0
6.5.2.2.1	*Control input*	Y	*Control flows in activity diagrams provide the control input. Control flow is represented in state machine diagrams by a transitions which activate states and in sequence diagrams by the passing of messages.*	*UML::ActivityEdge,* *UML::ControlFlow, UML::Transition, UML::Message, SysML::ControlValue*	*Activities, Interactions* *State Machines*	1.0

	a	Y	*Multiple control flows in an activity diagram that are input to a single activity node (i.e., action) are assumed to be "anded" together.*	*SysML::ControlValue, SysML::InputPin.is Control=true for control queuing*	*Activities*	1.0
	b	Y	*Control inputs are discrete valued inputs that can enable or disable an activity node.*	*SysML::ControlValue*	*Activities*	1.0
	c	Y	*In activity diagrams, the activity is invoked (enabled) when a token is received by the calling action. This includes tokens from all mandatory inputs and control inputs.*	*UML::Action,* *UML::ControlFlow, UML::ActivityEdge*	*Activities*	1.0
	d	Y	*In activity diagrams, a control operator can produce an output control value to disable the execution of an activity. An action enclosed within an interruptible region also can disable the execution of an activity. In state machine diagrams, transition events can disable the actions in a state.*	*UML::Action, UML::Interruptible ActivityRegion, SysML::ControlValue, UML::State*	*Activities, State Machines*	1.0
	e	Y	*An executing activity with non-streaming inputs and outputs terminates when it completes its transformation and produces an output value. An executing activity with continuous streaming inputs will terminate when it receives a disable from a control value and/or a signal that terminates the actions within an interruptible region. A TimeExpression can be specified in a control operator or can signal a termination in an interruptible region. An activity can also be terminated based on events, including timeout events, on a transition in a state machine diagram. In state machine diagrams, completion events occur upon completion of an activity.*	*UML::Activity, UML::Interruptible ActivityRegion, SysML ControlValue, UML::Time Expression, UML::State*	*Activities, State Machines*	1.0
	f	Y	*The enabling of actions without explicit control flows as inputs are enabled based on the control associated with its inputs.*	*UML::Action, UML::ObjectNode*	*Activities*	1.0

	g	Y	*A control flow connects the control inputs from one activity node to another. The control input can also be the output control value of a control operator.*	*SysML::ControlValue, UML::Parameter, UML::ControlFlow*	*Activities*	1.0
6.5.2.2.2	*Control operator*	Y	*A control operator provides the mechanism apply control logic to enable and disable activity nodes.*	*SysML::Control Operator, SysML::ControlValue*	Activities	1.0
	a	Y	*Control Nodes such as joins, forks, etc. provide capability to activate activity nodes based on "and" and "or" logic. A SysML Control Operator provides the additional capability to disable an activity node.*	*UML::ControlNode, SysML::Control Operator, SysML::ControlValue, UML::Parameter*	*Activities*	1.0
	b	Y	*A join specification can be used to specify arbitrarily complex logic for enabling an activity node. A control operator can also be used to specify complex logic for enabling and disabling an activity node.*	*UML::JoinNode with join specification, UML::Parameter, SysML::Control Operator, SysML::ControlValue*	*Activities*	1.0
	c	Y	*The control nodes identified below provide the basic control logic for enabling activity nodes. Note: multi exit functions are supported by parameter sets. Also, Interaction Operators provide similar logic in Sequence Diagrams.*	*UML::ControlNode, UML::Interaction Operator*	Activities, Interactions	1.0
	c1	Y	*Decision nodes in activity diagrams support selection. The "alt" Interaction Operator supports selection in sequence diagrams.*	*UML::DecisionNode, UML::Interaction Operator.Alt*	Activities, Interactions	1.0
	c2	Y	Forks in activity diagrams support a single input flow generating multiple concurrent output flows. The “par” Interaction Operator supports concurrent message flow in Sequence Diagrams.	*UML::Fork, UML::Interaction Operator.par*	Activities, Interactions	1.0
	c3	Y	A join “and's” multiple input flows together resulting in a single output flow.	*UML::Join*	Activities	1.0
	c4	Y	A merge results a single output flow upon arrival of the first of multiple input flows.	*UML::Merge*	Activities	1.0

	c5	Y	*Decision and loop nodes support iteration and looping. The "loop" Interaction Operator supports loops in sequence diagrams.*	UML::Decision-Node, UML::Loop Node, Interaction-Operator.loop	Activities, Interactions	1.0
	c6	N				
6.5.2.2.3	*Events and conditions*	Partial	*Triggers and constraints as guards provide the mechanism for modeling events and conditions.*		Activities, Interactions, State Machines	1.0
	a	Partial	*A trigger can be used to specify an event. Events can be associated with control flows in activity diagrams, transitions in state machine diagrams, and sending and receiving of messages in sequence diagrams.*	*UML:: Trigger, UML::AcceptEvent Action including UML::TimeTrigger, UML::Event Occurence in Interactions.* *Note: Failure event can be result in various types of actions that terminate an Interruptible Region in Activities, etc.*	*Activity, Interactions* *State Machines*	1.0
	b	Y	*Refer to a) above*	*UML::ActivityEdge, UML::Trigger*	*Activity, Interactions* *State Machines*	1.0
	c	Y	*Conditions can be specified as constraints that define guards to control execution of behaviors.*	*UML::Constraint (guard)*	*Activity, Interactions* *State Machines*	1.0
6.5.2.3	*Function-based behavior*	Y	*Activity diagrams provide the capability to model function based behavior.*	*UML:: Activity*	*Activities*	1.0
6.5.2.4	*State-based behavior*		*State machine diagrams provide the capability to model state based behavior with the specific modeling constructs indicated. Note 2 response: Activities are common to each type of behavior including both function based and state based. Note 3 response: A state is defined based on some invariant being true. The invariant can include reference to certain property values.*	*UML::StateMachine*	*State Machines*	1.0

	a	Y	*State*	*UML::State*	State Machines	1.0
	b	Y	*Simple state*	*UML::State,* *isSimple=True*	State Machines	1.0
	c	Y	*Composite states can contain one region or two or more orthogonal (concurrent) regions, each with one or more mutually exclusive disjoint states*	*UML::State* *isComposite=True*	State Machines	1.0
	d	Y	*Transitions between states which are triggered by events with guard conditions.*	*UML::Transition, UML::Trigger*	State Machines	1.0
	e	Y	*Transition within a composite state*	*UML::Transition (TransitionKind= Internal)*	State Machines	1.0
	f	Y	*Pseudo states include joins, forks and choice*	*UML::PseudoState*	State Machines	1.0
	g	Y	*Transitions between states which are triggered by events with guard conditions.*	*UML::Activity*	State Machines	1.0
	h	Y	*Entry, exit, doActivities are performed upon entry or exit from a state or while in a state.*	*UML::Activity*	State Machines	1.0
	i	Y	*State machine semantics define the ordering of actions that are completed when exiting a composite state (refer to UML transition semantics). When a composite state is exited, the exit actions are executed beginning with the most nested state.*	*UML::State (Note: refer to semantics)*	State Machines	1.0
	j	Y	*Entry and exit actions must be completed prior to exiting a state. A doActivity does not need to be completed to execute.*	*UML::State (Note: refer to semantics)*	State Machines	1.0
	k	Y	*Send and receive signals can be sent via actions to interact with other objects.*	*UML::SendSignal Action*	State Machines	1.0
	l	Partial	The failure and/or exception states are user defined and have no uniquely defined representation. The use of exit points on states can be used to exit the state when a failure event occurs.	*UML::State*	State Machines	1.0

6.5.2.4.1	*Activation time*	Y	*The interval of time that an activity or state is active can be modeled by a UML Time Trigger or Time Interval and corresponding Time Expression (refer to UML trigger and interval notation). Note: A UML timing diagram is not included in SysML at this time, but could be used to model the time associated with the occurrence of events, such as state changes, or changes in property values.*	*UML::SimpleTime*	Activities, Interactions, State Machines	1.0
6.5.2.5	*Allocation of behavior to systems*	Y	An allocation relationship provides a generalized capability to allocate one model element to another.	*SysML::Allocation, SysML::Allocated, UML::NamedElement*	*Allocations*	1.0
	a	Y	In general, behaviors such as activities, interactions, and state machines are owned by a Behaviored Classifier which can correspond to an block. The SysML Allocation relationship can be used to explicitly allocate behaviors to blocks. Alternatively, activity partitions (swim lanes) can be used to allocate the action and/or activity to a part and/or block.	*UML::BehavioredClassifier and UML::Behavior (owned behavior) - Refer to UML Common Behaviors, SysML::Allocate, SysML::Allocate AcitivtyPartition*	*Allocations, Activities*	1.0
	b	Partial	*An object node in an activity diagram can be allocated to an item that flows in an internal block diagram using an allocation relationship. Note: the object node is typed by the same classifier as the item that flows. See req't 6.5.2.1.1.*	*SysML::Block (type of ObjectNode to type of ItemProperty), UML::ObjectNode, UML::Property*	*Allocations, Activities, Ports and Flows*	1.0
6.5.3	*Property*	N/A	Properties and their relationships are represented in SysML using properties of blocks in conjunction with constraint blocks to capture the relationships between them.		*Blocks, Constraint Blocks*	
6.5.3.1	*Property type*	Y	Primitive types, data types, and value types provide the capability to model the different types of quantitative properties.	*UML:: PrimitiveType, UML::DataType,* *SysML::Value Type*	*Blocks*	1.0
	a	Y	Primitive type.	UML::Integer		

	b	Y	Primitive type.	UML::Boolean		
	c	Y	Primitive type.	UML::Enumeration		
	d	Y	Primitive type.	UML::String		
	e	Y	Primitive type.	SysML::Real		
	f	Y	Data type.	SysML::Complex		
	g	Y	Composite data type made up of primitive types.	Refer to a-f		
	h	Y	Composite data type made up of primitive types.	Refer to a-f		
6.5.3.2	*Property value*	Y			*Auxiliary*	1.0
	a	Y	Value properties are typed by a value type or data type and have an associated value.	SysML::Block Property, SysML::ValueType, UML::DataType,	Blocks	1.0
	b	Y	A value type can include a dimension and units such as length and feet or meters.	SysML::ValueType (unit and dimension are defined as blocks in a model library)	Blocks	1.0
	c	Y	A value property is a block property that is typed by a value type that can have an associated probability distribution on its values.	*SysML::ValueType, SysML::Distribution Definition*	Blocks	1.0
	d	Y	Source data can be included in a comment attached to the property or a user defined stereotype could be applied.	UML::Comment	Model Elements	1.0
	e	Y	Reference data can be included in a comment attached to the property or a user defined stereotype could be applied.	UML::Comment	*Model Elements*	1.0
6.5.3.3	*Property association*		*A value property can be a feature of any classifier (.i.e., block)*	SysML::Block, SysML::Block Property	Blocks	1.0
	a	Y	Blocks, parts, or items that flow can have (or reference) properties.	*SysML::Block, SysML::Block Property*	Blocks	1.0
	b	Y	A function (activity) can have properties since it is a class	*UML::Activity*	Activities	1.0

	c	Partial	An event is specified by a trigger which is an element. The element does not have properties.A signal which is sent upon the occurrence of the event can have properties.	UML::Signal		1.0
	d	Y	A property can be related to other properties through a constraint property	SysML::Constraint-Block, SysML::Constraint-Property	Constraint Blocks	1.0
6.5.3.4	*Time property*	Y	Time can be treated as a property, typed by a Real that can represent either continuous or discrete time. Time ultimately derives from clocks which can be continuous or discrete. Clocks can be modeled as blocks which have a time property that can be bound to a parameter of a constraint property (e.g., equation). Time durations, start and stop times, etc. can be modeled using the UML time model for time triggers, time expressions, intervals, and durations. Note: More elaborate models of time and clocks can be found in the UML schedulability, performance, and time profile.	*SysML::Block,* *SysML::Block Property,* *SysML::ValueType,* *SysML::Constraint Property, SysML::Constraint Parameter, UML::SimpleTime Package*	*Blocks,* *Constraint Blocks, Interactions*	1.0
6.5.3.5	*Parametric model*	*Y*	*The parametric diagram supports modeling of constraints which bind parameters of the constraints to value properties.*	*SysML::Constraint Block,* *SysML::Constraint Property* *SysML::Constraint Parameter,* *SysML::Block Property,* *UML::Connector, SysML::Nested ConnectorEnd*	Constraint Blocks	1.0
	a	Y	Constraints blocks and their usages (constraint properties) specify the mathematical relationships/constraints between constraint parameters.	SysML::Constraint-Block, SysML::Constraint-Parameter	Constraint Blocks	1.0

	b	Partial	Mathematical and logical expressions can be defined in SysML in a reference language, but there is no interpreter built into SysML. The range of values can be specified via value properties and probability distributions per 6.5.3.2a-c.	SysML::Block Property, SysML::Distribution-Definition	Blocks	1.0
	c	Y	The reference language for interpreting the constraint can be included as part of the ConstraintBlock along with the compartment for the expression.	SysML::Constraint-Block	Constraint Blocks	1.0
6.5.3.6	*Probe*	N	*No specific mechanization has been provided. In the testing profile, there is a mechanism to capture data and create actions in response to the data. This will be investigated in a future version of SysML.*			N
6.5.4	*Requirement*	N/A	The requirements diagram provides the basic capability for relating text based requirements to other SysML models.		Requirements	1.0
6.5.4.1	*Requirement specification*	Y	A requirement is a stereotype of a class in SysML. The various subtypes of requirement are specified as subclasses of the the requirement stereotype and can include specific properties and constraints on what model elements can satisfy the subclass of requirement. A sample set of subclasses of requirements are included in the NonNormative Extensions Annex C.	*SysML::Requirement*	*Requirements,* *Non-Normative Extensions, Profiles & Model Libraries*	1.0
	Note 1	Y	Values and tolerances can be specified as part of the text property or via property values and distributions per 6.5.3.2a-c.	Requirement.text, SysML::Value Property	Requirements, Blocks	1.0
	Note 2	Y	There is no explicit subclass of requirement as a stakeholder need, but a requirement can be named or subclassed as “stakeholderNeed.”	SysML::Requirement	*Requirements,* *Non-Normative Extensions*	1.0

	Note 3	Y	User defined requirements can be added via subclasses to specify any type of life cycle requirement of interest to the modeler.	SysML:: Requirement	*Requirements,* *Non-Normative Extensions, Profiles & Model Libraries*	1.0
	a	Y	Operational requirement	SysML:: Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	b	Y	Functional requirement	SysML::functional-Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	c	Y	Interface requirement	SysML::interface Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	d	Y	Performance requirement	SysML::perfor-manceRequirement	*Requirements,* *Non-Normative Extensions*	1.0
	e	Y	Activation/Deactivation (Con-trol) requirement	SysML:: Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	f	Y	Storage requirement	SysML:: Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	g	Y	Physical requirement	SysML::physical Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	h	Y	Design constraint	SysML:: Requirement	*Requirements,* *Non-Normative Extensions*	1.0

	i	Y	Specialized requirement	SysML:: Requirement	*Requirements,* *Non-Normative Extensions*	1.0
	j	Y	Measure of effectiveness	SysML::moe	*Requirements,* *NonNormative Extensions*	1.0
6.5.4.2	*Requirement properties*	Y	A requirement includes default properties for id and text. Other properties can be added via stereotype properties.	*SysML::Requirement*	*Requirements,* *Non-Normative Extensions, Profiles & Model Libraries*	1.0
6.5.4.3	*Requirement relationships*	Y	The requirement relationships include the relationships containment, trace, deriv-eReqt, satisfy, verify and refine relationships.		Require-ments	1.0
	a	Y	A derive relationship relates a derived (target) requirement to a source requirement.	*SysML::deriveReqt*	*Requirements*	1.0
	b	Y	A satisfy relationship relates the model elements (i.e. the design) to the requirements that are satisfied.	*SysML::satisfy*	*Requirements*	1.0
	c	Y	Goals, capabilities, or usages of systems are often expressed using use cases. Subgoals can be represented using the include and extend relationships between use cases. Requirements can be related to use cases using the refine relationship. Requirements use the containment relation-ship to breakdown an existing requirement into its containing requirements.	*UML:UseCase, UML::Include,* *SysML::Requirement, UML:refine*	Require-ments, Use Case	1.0
6.5.4.4	*Problem*	Y	A problem is an extension of a comment that can be attached to any model element. Note: This could also be used to rep-resent issues.	*SysML::Problem*	*Model Elements*	2.0

6.5.4.5	*Problem association*	Y	Refer to 6.5.4.4	*SysML::Problem*	*Model Elements*	2.0
6.5.4.6	*Problem cause*	N				2.0
6.5.5	*Verification*	N/A	*The following responses to the Verification requirements will include references to the Testing Profile [OMG Adopted Specification* *ptc/03-08-03] which is not currently part of SysML but is intended to be evaluated for integration with version 1.1 of SysML [refer to white paper on integrating SysML with Testing Profile]*		Require-ments	
6.5.5.1	*Verification Process*					
	a	Y	*The SysML verify relationship between one or more system requirements and one or more test cases represents the method for verifying that a system design satisfies its requirements. A verified system design implies that the system will satisfy its requirements if the component parts satisfy their allocated requirements. An alternative approach to capture the verify relationship is to associate a test case with a satisfy relationship using the rationale.*	*SysML::Verify, SysML::Rationale*	*Requirements, Model Elements*	1.0
	b	Y	*The SysML verify relationship between one or more requirement(s) and one or more test case(s) is used to verify that the implemented system design instances satisfy their requirements. Alternatively, a reference to a TestCase using SysML:Rationale may be attached to a satisfy relationship.*	*SysML::Verify SysML::Rationale*	Require-ments, Model Elements	1.0
	c	Y	*A derive relationship between the requirement being validated and the higher level requirement or need may have a Rationale attached that references the validation method(s).*	*SysML:deriveReqt* *SysML::Rationale*	Require-ments, Model Elements	1.0

	Note 1	Y	*Verification methods of analysis and similarity may be modeled as a Rationale with reference to the specific analysis report or other reference data. Verification methods including Test, Inspection, and Demonstration may be modeled as a TestCase.*	*SysML::Rationale, SysML::TestCase*	Require-ments	1.0
	Note 2	Partial	*Validation methods are user defined. A rationale can reference the user defined methods.*	*SysML::Rationale*	Model Elements	1.0
6.5.5.2	*Test case*	Partial	A test case refers to the method for verifying a require-ment. Note: The testing profile associates a test case with a behavior that can include the specific method and associ-ated input stimulus and response.	SysML::TestCase	*Requirements*	1.0
	Note 1	Partial	Refer to above note on the testing profile.			1.x
	Note 2	Partial	The test criteria can be established via the require-ment			1.x
	Note 3	Partial	Test cases can contain other test cases, like any other named element.	SysML::TestCase	Require-ments	1.0
6.5.5.3	*Verification result*	Partial	*The result of a SysML:TestCase may be expressed through its verdict attribute (Testing Profile)*	*SysML::TestCase, SysML::Verdict*	*Requirements*	1.0
6.5.5.4	*Requirement verification*	Partial	*A constraint may be used to relate the required value to the verification result.*	*SysML::Constraint Property;* *SysML::TestCase,* *SysML::Rationale*	*Requirements, Constraint Blocks*	1.0
6.5.5.5	*Verification procedure*	Partial	A rationale can be associated with the test case or the satisfy relationship between a require-ment and a design, and reference a verification procedure. Note: The testing profile will associate a behav-ior with a test case which can be implemented by a specific procedure.	*SysML::TestCase, SysML::Rationale*	*Requirements, Model Elements*	1.x
	Note					

6.5.5.6	*Verification system*	Partial	A verification system can be modeled as any other system (block) or it can be modeled as the system environment. However, the future integration with the testing profile is intended to provide explicit modeling of the verification system.	SysML::Block	Blocks	2.0
6.5.6	*Other*	N/A				
6.5.6.1	*General relationships*	Y	*SysML includes several standard UML relationships as described below.*			
	a	Y	An association relationship.	UML::Association	Blocks	1.0
	b	Y	A package contains packageable elements and can represent collections of elements.	UML::Package, UML::Packageable Element; UML::owned Member	Class	1.0
	c	Partial	Blocks can be decomposed into parts that are typed by other blocks using composition (refer to Reqt 6.5.1.1). The completeness of the decomposition is not explicitly represented.	SysML::Block, SysML::Block Property, UML::Association (composition)	Blocks	1.0
	d	Y	A dependency relationship.	UML::Dependency	Model Elements	1.0
	e	Y	Generalization/specialization relationship. Generalization sets provide the means to partition specializations to support further categorization.	*UML::Generalization, UML::Generalization Set*	Blocks	1.0
	f	Y	Instantiation is modeled using Instance Specifications to uniquely identify a classifier. Instances are represented as a property specific value with a unique set of values.	UML::Instance Specification, UML::InstanceValue	Blocks	1.0

6.5.6.2	*Model views*	Partial	A view represents the model from a particular viewpoint. Both the view and the viewpoint are represented in SysML. The view is a stereotype of package that identifies the set of model elements that conform to the viewpoint, and the viewpoint specifies the stakeholders, their purpose, concerns and the construction rules (language and methods) to specify the view. Note: The model elements that depict the view are visually represented in diagrams, tables, and other notation. Integrity between model views is accomplished by creating a well formed model. This in part results from the constraints imposed by the language, and in part is defined by the specific methodology and tools that are employed. Navigation among views results from a tool vendor implementation.	SysML::View, SysML::Viewpoint SysML::Conform	*Model Elements*	1.0
6.5.6.3	*Diagram types*				Diagram Appendix	1.0
	a		The standard UML diagram types that are needed to support the requirements have been included in SysML. Some additional diagram types provide some redundant capabilities, but have been preserved to allow flexibility in representations and methodologies. For example, the sequence diagrams along with activity and state diagrams provide overlapping capability for representing behavior. A few diagram types have not been included explicitly in SysML, although they are not precluded from use along with SysML.	*N/A*	Diagram Appendix	1.0

	b		The requirements diagram and parametric diagram have been added to address the requirements of this RFP. In addition, an informal mechanism has been added to represent diagram usages. This enables renaming and constraining the usage of a particular diagram type for a particular usage.	SysML::Diagram Usage	Diagram Appendix	1.0
6.5.6.4	*System role*	Partial	A part in a block represents the role for a classifier in the context of the enclosing block. It defines the relationship between an instance of the classifier that types the part and an instance of the block that encloses the part. This is a primary mechanism for providing a unique context for a part of a whole (enclosing block). The part may use only a subset of the behavior and properties of the class that types the part. However, the specific mechanism for containing the subset has not been explicitly defined.	SysM::Block, SysML:Block Property	Blocks	1.0
6.6	*Optional Requirements*	N/A				
6.6.1	*Topology*	N				
	a	N				2.0
	b	N				2.0
6.6.2	*Documentation*	Y	A document (stereotype of artifact).	UML::Document	*Diagram Appendix*	1.0
	a	Y	The document stereotype can include stereotype properties to represent information about the document.	UML::Document	*Profiles & Model Libraries*	1.0
	b	Y	The trace relationship relates a document to other model elements.	UML::Trace	*Diagram Appendix*	1.0
	c	N	The ability to represent the text of the document in terms of the descriptions provided by the related (traced) model elements is accomplished by a tool implementation.			

6.6.3	*Trade-off studies and analysis*	Partial	*Parametric diagrams can depict the relationship between measures of effectiveness and various system properties (including probability distributions on their values) to evaluate the effectiveness of a particular system model. Specific constructs for criteria, weighting, and alternatives are planned for a future version of SysML to support modeling of trade studies.*	*SysML::moe, SysML::objective Function, SysML::Constraint Property*	Constraint Blocks, Non-Normative Extensions	1.0
	a	Y	Alternative models can be specified via organization of models/packages. Model libraries can be used to establish reusable portions of the model.	UML::Model, UML::Package	Model Elements, Profiles & Model Libraries	1.0
	b	Partial	Criteria can be modeled as properties typed by value types or as Requirements	SysML::Block Property, SysML::ValueType, SysML::Requirement	Blocks, Requirements	1.0, 2.0
	c	Y	Measures of effectiveness are modeled as a subclass of block property that represents a value property. A constraint can represent the objective function.	SysML::moe, SysML::Constraint-Property	Non-Normative Extensions, Constraint Blocks	1.0
6.6.4	*Spatial representation*	N				
6.6.4.1	*Spatial reference*	N				
6.6.4.2	*Geometric relationships*	N				
6.6.5	*Dynamic structure*	Partial				
	a	Y	The action semantics provide the capability for creating and destroying objects.	*UML::CreateObject Action, UML::DestroyObject Action*	Action (UML Spec)	1.0
	b	Partial	The capability is partially provided by 6.6.5a.			2.0
	c	N				2.0
	d	N				2.0

6.6.6	*Executable semantics*	*Partial*	*The action semantics are intended to provide execution semantics. There is no formal graphical syntax for this.*	*UML::Action*	Action in UML Spec	1.0
6.6.7	*Other behavior modeling paradigms*	Y	A UML behavior is a generalized behavior that can accommodate a wide range of behavior modeling paradigms. This include function based, state based, and message based behavior (sequence diagrams).	*UML::Behavior*	Activities, Interactions, State Machines	1.0
6.6.8	*Integration with domain-specific models*	Partial	SysML is a general purpose language that will integrate with other types of domain specific models. This is accomplished in part by mapping SysML via XMI to the AP233 data interchange standard. In addition, the parametric diagram is intended to provide a capability to integrate with domain specific engineering analysis models.		Model Interchange	1.x, 2.0
6.6.9	*Testing Model*	Partial	SysML is intended to be integrated with the UML Testing Profile. Refer to Response to Reqt 6.5.5 above.	SysML::TestCase	Requirement	2.0
6.6.10	*Management Model*	N				

Annex F: Terms and Definitions

(informative)

The SysML glossary is included as a support document ad/2006-03-04 to this specification**.** The terms and definitions are referred to in the SysML specification and are derived from multiple sources including the UML Superstructure (formal/05-07-04) and the UML for Systems Engineering RFP (ad/03-03-41).

Annex G: BNF Diagram Syntax Definitions

(informative)

Editorial Comment: *This appendix has not been updated since an earlier draft version of the SysML specification. BNF productions below include notations that are no longer included by the current SysML specification and leave out other notations that are included. Only selected diagram types of SysML are currently included. Detailed productions for specific diagram types will be updated during the finalization phase of the SysML specification. In the meantime, these productions show the approach to specification of language syntax that a hybrid graphical-textual form of BNF can support. This appendix is for illustration only. The diagram elements tables in each SysML chapter, along with descriptions of diagram extensions, provide the current normative specification of SysML notations.*

G.1 Overview

This annex provides detailed definition of the graphical and textual elements that may appear in SysML diagrams. It specifies the structure of these diagrams using an extension of Backus-Naur Form (BNF) to define both textual and graphical syntax. BNF is a widely used notation for defining a language grammar.

The BNF definition of diagram elements in this appendix is more complete, both in identifying the diagram elements that may be present and the combinations in which they may appear, than the diagram summary tables contained in the preceding chapters of the SysML specification. Tool implementors may use the BNF definition to make sure the diagram elements they support are well-formed according to the grammar of the SysML language. Users of the SysML language may use the BNF definition as a comprehensive guide to the permitted forms of diagrams and diagram elements, including optional elements and notational variants, even though the BNF formalism may be less familiar or convenient to a user than examples of syntax produced according to the grammar. For both tool implementors and users, the grammar of diagram elements is only an partial definition of valid diagram structures. Diagrams also impose additional constraints on how diagram elements may be used, for example to interpret the diagrams in consistent ways according to the meaning of a model, above the level of the context-free language grammar defined by BNF.

G.2 Summary of BNF Syntax Definition Conventions

Backus-Naur Form (BNF) is a notation to specify the grammar of a context-free language. A context-free language is defined by a set of production rules, which specify the permitted expansions of symbols that group lower-level elements of the language. A terminal symbol is a symbol that has no further expansion. BNF defines a notation for the specifying production rules containing terminal and non-terminal symbols of a language grammar.

The BNF notation used in this appendix is based on the variant of the BNF notation used by UML to specify selected elements of textual syntax. It has been extended to support the semi-formal definition of the hybrid graphical and textual syntax of SysML diagrams. Following is a summary of the conventions of the BNF used in this Annex:

- Symbols not defined by literal graphics or text are enclosed in angle brackets (e.g., <symbol>).
- Production rules for non-terminal symbols are expressed by the ‘::=’ operator followed by an expression specifying the permitted symbols of the expansion.
- An expression may be either a sequence of subexpressions and operators, or a two-dimensional rendering of a grammatical element containing subexpressions.

- Terminal symbols consisting of literal text strings are enclosed in single quotes (e.g., ‘=’).
- Symbols represented by graphical elements are represented by an informal rendering of the graphical element within the expansion, possibly augmented by informal comments enclosed in parentheses.
- An optional element of an expression is specified by enclosing it in square brackets (e.g., [<symbol>])
- Zero or more repetitions of an expression is specified by an asterisk following the expression: (e.g., <symbol>*).
- One or more repetitions of an expression is specified by a plus-sign following the expression: (e.g., <symbol>+).
- Alternative choices within an expression are separated by the ‘|’ symbol (e.g., <alternative-A> | <alternative-B>).
- A sequence of subexpressions and operators to be grouped as a single expression is enclosed in parentheses (e.g., ([<symbol-1>] <symbol-2>*).
- A comment is specified by a text string enclosed in parentheses, either within a graphical expansion of a symbol, or following a production and starting on a new line.
- Alternative productions for a symbol may be specified by multiple productions in which the symbol appears to the left of the ‘::=’ operator. This extension to standard BNF permits the grammar of multiple diagrams to be distributed across multiple subsections of the total definition.
- A connection to a symbol defined in another production is specified by preceding the referenced symbol with an ampersand (e.g., &<symbol>). This extension to standard BNF assists in the definition of two-dimensional connection topologies, in which the ends of graphical path elements are connected to graphical node elements already defined in other productions.

Symbols in this annex which have graphical productions follow a naming convention of having capitalized names without internal hyphens, while those with textual productions have lower-case names with internal hyphens as needed.

G.3 BNF Definition of SysML Diagrams

G.3.1 Top-level Productions

<SysMLDiagram> ::=
<PackageDiagram> |<BlockDefinitionDiagram> | <InternalBlockDiagram> | <InstanceDiagram> |
<ParametricDiagram>
(additional productions for this symbol can appear in subsections below)

G.3.2 General-purpose Symbols

<name-elements> ::= [<name>] <stereotype-icon>*
(icons located in upper-right-hand corner of containing node)

<name> ::= [<keywords>] [<namespace-visibility>] <name-string>

<keywords> :: <stereotype-icon>* [(‘«’ <name-string> ‘»’)+ | ‘«’ <name-string> (‘,’ <name-string>)+ ‘»’]
(no whitespace may appear immediately inside the ‘«’ ‘»’ characters)

<namespace-visibility> ::= ‘+’ | ‘-’

<name-string> ::=
(terminal symbol consisting of string of characters in some character set encoding)

<body-text> ::=
(terminal symbol consisting of string of characters in some character set encoding,
which may include various forms of text formatting)

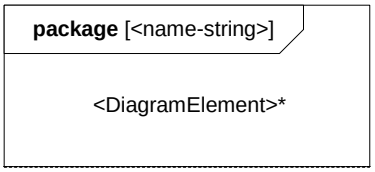
<digit-string> ::=
(terminal symbol consisting of string of digit characters in some character set encoding)

<stereotype-icon> ::=
(terminal symbol consisting of a graphical icon which may be shown instead of a stereotype keyword)

<custom-notation> ::=
(terminal symbol consisting of any graphical notation for an element to which a stereotype has been applied)

G.4 Diagram Elements Defined in Model Elements Chapter

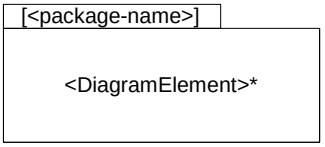
<PackageDiagram> ::=



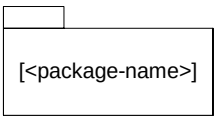
<DiagramElement> ::=
<Package> | <PublicPackageImport> | <PrivatePackageImport> | <PackageContainment> |
<CrossCuttingElement> |
<custom-notation>
(additional productions for this symbol appear in other subsections below)

<Package> ::= <PackageWithNameInTab> | <PackageWithNameInside>

<PackageWithNameInTab> ::=



<PackageWithNameInside> ::=

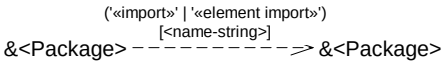


<package-name> ::= <name-elements> <element-import>*

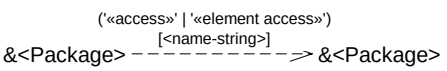
<element-import> ::= ‘{‘ ‘element’ (‘import’ | ‘access’) [‘as’ <name-string>‘ <qualified-name>}’

<qualified-name> ::= <name-string> [‘::’ <name-string>]*

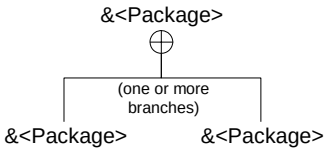
<PublicPackageImport> ::=



<PrivatePackageImport> ::=



<PackageContainment> ::=

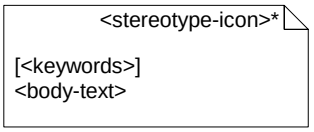


<CrossCuttingElement> ::= <Comment> | <Constraint> | <Dependency>

<Comment> ::=
[&<ModelElement> <dashed-line-connection>] * <CommentNoteBox>

<dashed-line-connection> ::=

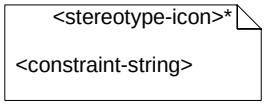
<CommentNoteBox> ::=



<Constraint> ::= <ConstraintNote> | <ConstraintTextNote> |
<ConstraintAsDashedLine> | <ConstraintAsDashedLineCrossingPaths>

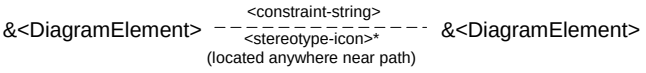
<ConstraintNote> ::=
[&<ModelElement> <dashed-line-connection>]+ <ConstraintNoteBox>

<ConstraintNoteBox> ::=

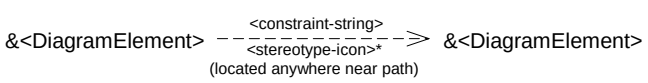


<ConstraintTextNote> ::= &<ModelElement> <constraint-string>
(constraint string located near element)

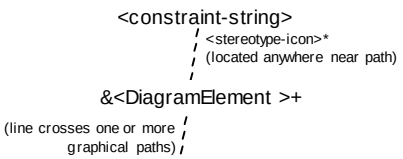
<ConstraintAsDashedLine> ::=



<ConstraintAsDashedLine> ::=

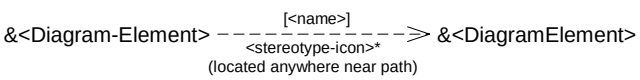


<ConstraintAsDashedLineCrossingPaths> ::=

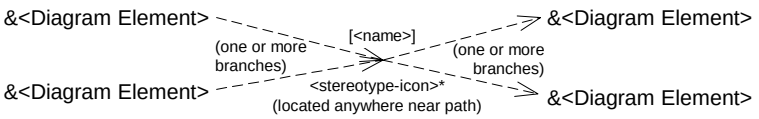


<constraint-string> ::= ‘{‘ [<name> ‘:’] <value-specification> ‘}’’

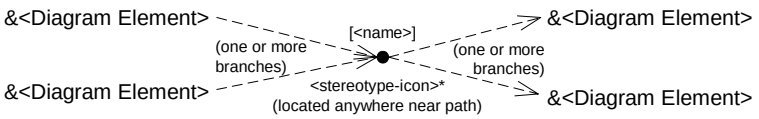
<Dependency> ::=



<Dependency> ::=



<Dependency> ::=



<value-specification> ::=
<expression> | <opaque-expression> |
<literal-boolean> | <literal-integer> | <literal-unlimited> | <literal-null> | <literal-string>

<expression> ::=
<name-string> ['(' [<value-specification>] (',' [<value-specification>]] ')')* |
<body-text>

<opaque-expression> ::=
['{' <language-identifier> '}'] <body-text>

<language-identifier> ::= <name-string>

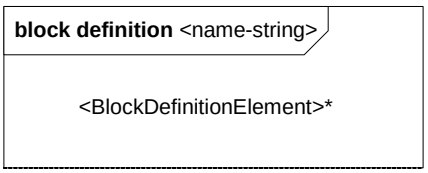
<literal-boolean> ::= 'true' | 'false'
<literal-integer> ::= <digit-string>
<literal-unlimited> :: <literal-integer> | '*'
<literal-null> ::= 'null'
<literal-string> ::= '"' <body-text> '"'

G.5 Diagram Elements Defined in Blocks Chapter

G.5.1 Diagram Elements Defined in Block Definition Diagrams

<DiagramElement> ::= <BlockDefinitionElement>
(additional production for previously defined symbol)

<BlockDefinitionDiagram> ::=

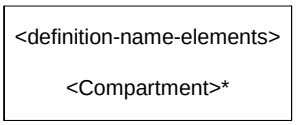


<BlockDefinitionElement> ::= <Package> | <CrossCuttingElement> | <BlockDefinitionNode> | <BlockDefinitionPath>

<BlockDefinitionNode> ::= <BlockOrValueType>

<BlockDefinitionPath> ::= <Association> | <Generalization> | <BlockNamespaceContainment>

<BlockOrValueType> ::=



<definition-name-elements> ::= [<definition-name>] <stereotype-icon>*
(icons located in upper-right-hand corner of containing node)

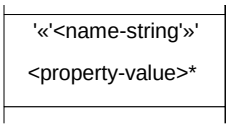
<definition-name> ::= [<definition-keywords>] [<namespace-visibility>] <name-string>

<definition-keywords> :: <stereotype-icon>* [(‘«’ <definition-keyword> ‘»’)+ |
‘«’ <definition-keyword> (‘,’ <definition-keyword>)+ ‘»’]
(no whitespace may appear immediately inside the ‘«’ ‘»’ characters)

<definition-keyword> ::= ‘block’ | ‘value’ | <name-string>
(additional productions for this symbol appear in subsections below)

<Compartment> ::= <StereotypePropertyCompartment> | <NamespaceCompartment> | <StructureCompartment> |
<FeatureCompartment>

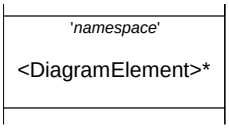
<StereotypePropertyCompartment> ::=



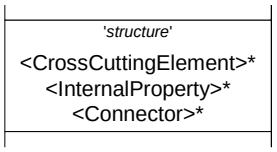
(no whitespace may appear immediately inside the ‘«’ ‘»’ characters)

<property-value> ::= <name-string> ‘=’ <value-specification> (, <value-specification>)*

<NamespaceCompartment> ::=



<StructureCompartment> ::=



<FeatureCompartment> ::=

<CompartmentLabel> ::= ‘*operations*’ | ‘*properties*’ | ‘*parts*’ | ‘*references*’ | ‘*values*’ | ‘*constraints*’ | ‘<name-string-in-italics>
(additional productions for this symbol appear in subsections below)

<name-string-in-italics> ::=
(terminal symbol consisting of string of characters in some character set encoding, shown in italic font)

<compartment-element-list> ::=
<block-constraint>* |
(<static-operation> | <operation>)* |
(<static-property> | <property>)*

<block-constraint> ::= ‘{‘ <constraint-text> ‘}’ | <property>

<static-operation> ::= <u><operation></u>

<operation> ::= [<block-visibility>] [<keywords>] <name-string> ‘(‘ [<parameter-list>] ‘)’ [‘:’ [<return-type>]]
‘{‘ <oper-modifier> [‘,’ <oper-modifier>]* ‘}’]

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

<parameter> ::= [<direction>] <parameter-name> ‘:’ <type-expression> [‘[‘<multiplicity>’]’] [‘=’ <value-specification>]
[‘{‘ <param-modifier> [‘,’ <param-modifier>]* ‘}’]

<direction> ::= ‘in’ | ‘out’ | ‘inout’

<param-modifier> ::= ‘ordered’ | ‘unique’ | <constraint-string>

<oper-modifier> ::= ‘query’ | ‘ordered’ | ‘unique’ | ‘redefines’ <oper-name> | <constraint-string>

<static-property> ::= <property>

<property> ::= [<block-visibility>] [‘/’] [<property-type-keywords>] <name-string> <property-declaration>

<property-keywords> ::= <stereotype-icon>* [(‘«’ <property-keyword> ‘»’)+ |
‘«’ <property-keyword> (‘,’ <property-keyword>)+ ‘»’]
(no whitespace may appear immediately inside the ‘«’ ‘»’ characters)

<property-keyword> ::= ‘part’ | ‘reference’ | ‘value’ | <name-string>
(additional productions for this symbol can appear in subsections below)

<property-declaration> ::= [‘:’ <property-type>] [‘[‘ <multiplicity> ‘]’] [‘=’ <value-specification>]
[‘{‘ <prop-modifier > [‘,’ <prop-modifier>]* ’}’]

<prop-modifier> ::= ‘constant’ | ‘readOnly’ | ‘ordered’ | ‘unique’ | ‘bag’ | ‘sequence’ | ‘seq’ | ‘union’ |
‘subsets‘ <property-name> | ‘redefines’ <property-name> |
‘unit’ = <unit-name> | ‘dimension’ = <dimension-name> | ‘distribution’ ‘=’ <distribution-spec> |
‘<prop-constraint>

<distribution-spec> ::= <distribution-name> ‘(‘ <distribution-param-spec>* ‘)’

<distribution-param-spec> ::= param-name ‘=’ <value-specification> ‘{‘ ‘unit’ ‘=’ <unit-name> ‘}’]

<block-visibility> ::=<namespace-visibility> | ‘#’ | ‘~’

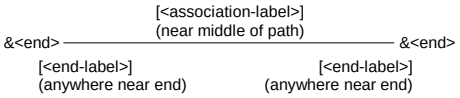
<multiplicity> ::= <lower-bound> ‘..’ <upper-bound>

<Association> ::= <ReferenceAssociation> | <PartAssocation>

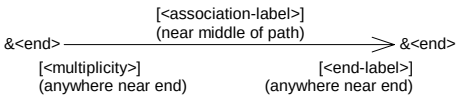
<ReferenceAssociation> ::= <BidirectionalReferenceAssociation> | <UnidirectionalReferenceAssocation>

<PartAssociation> ::= <BidirectionalPartAssociation> | <UnidirectionalPartAssocation>

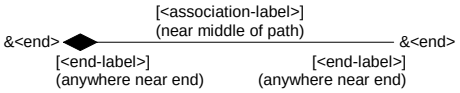
<BidirectionalReferenceAssociation> ::=



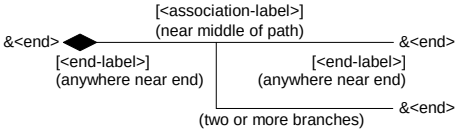
<UnidirectionalReferenceAssociation> ::=



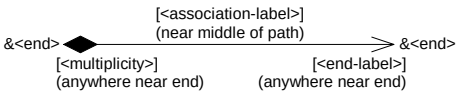
<BidirectionalPartAssociation> ::=



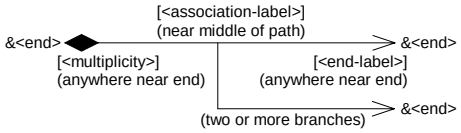
<BidirectionalPartAssociation> ::=



<UnidirectionalPartAssociation> ::=



<UnidirectionalPartAssociation> ::=



<end> ::= <BlockOrValueType>

<association-label> ::= <association-name-and-modifiers> |
<reading-direction> <association-name-and-modifiers> |
<association-name-and-modifiers> <reading-direction>

<association-name-and-modifiers> ::= <name-string> [‘{‘ <assoc-modifier> [‘,’ <assoc-modifier>]* ‘}’]

<reading-direction> ::= ▶

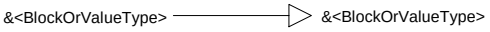
<reading-direction> ::= ◀

<end-label> ::= <end-label-element>*
(end label elements may appear in any relative position as long as all elements are near the end)

<end-label-element> ::= <end-name> | <prop-modifier> | <multiplicity>

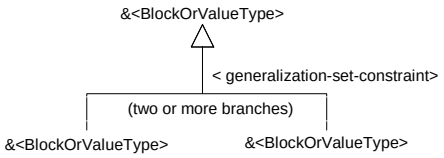
<end-name> ::= [<block-visibility>] [‘/’] [<keywords>] <name-string>

<Generalization> ::=

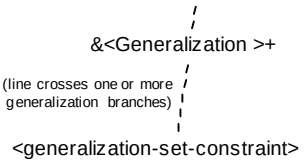


<generalization-node> ::= <BlockOrValueType>

<Generalization> ::=



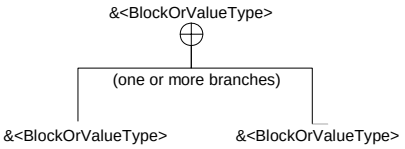
<GeneralizationSet> ::=



<generalization-set-constraint> ::= ‘{‘ <set-constraint> [‘,’ <set-constraint>] ‘}’

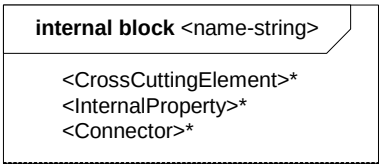
<set-constraint> ::= ‘complete’ | ‘incomplete’ | ‘disjoint’ | ‘overlapping’

<BlockNamespaceContainment> ::=

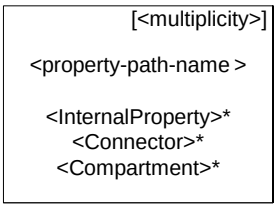


G.5.2 Diagram Elements Defined in Internal Block Diagrams

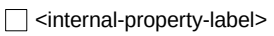
<InternalBlockDiagram> ::=



<InternalProperty> ::=



<InternalProperty> ::=



(additional productions for this symbol appear in subsections below)

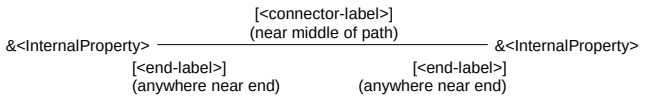
<property-path-name> ::= [<property-keywords>] [<block-visibility>] <name-string> [‘.’ <name-string>]* [‘:’ [<type-name>]

<type-name> ::= <name-string> | ‘[‘ [<name-string>] ‘]’

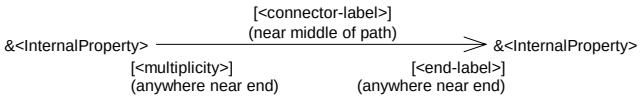
<internal-property-label> ::= <property-path-name> [‘[‘ <multiplicity> ‘]’]

<Connector> ::= <BidirectionalConnector> | <UnidirectionalConnector>

<BidirectionalConnector> ::=



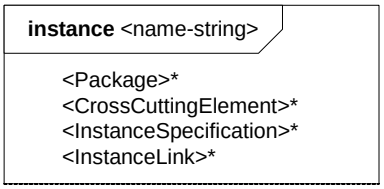
<UnidirectionalConnector> ::=



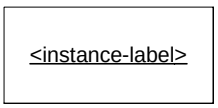
<connector-label> ::= [<name-string> ‘:’] [<name-string>]

G.5.3 Diagram Elements Defined in Instance Diagrams

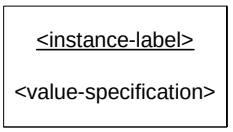
<InstanceDiagram> ::=



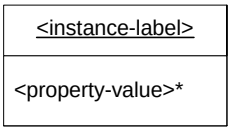
<InstanceSpecification> ::=



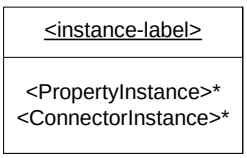
<InstanceSpecification> ::=



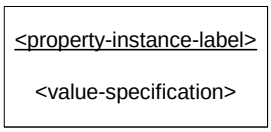
<InstanceSpecification> ::=



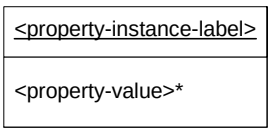
<InstanceSpecification> ::=



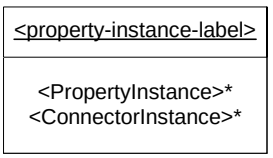
<PropertyInstance> ::=



<PropertyInstance> ::=



<PropertyInstance> ::=

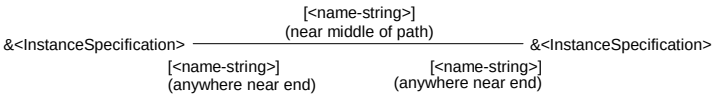


<instance-label> ::= [<name-string>] ':' [[<type-name> [',' <type-name>]*]

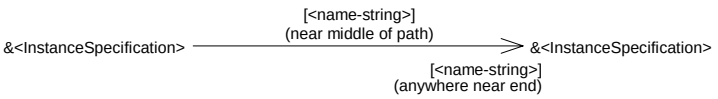
<property-instance-label> ::= [<name-string>] '/' [<name-string>] ':' [[<type-name> [',' <type-name>]*]

<InstanceLink> ::= <BidirectionalLink> | <UnidirectionalLink> | <BidirectionalPartLink> | <UnidirectionalPartLink>

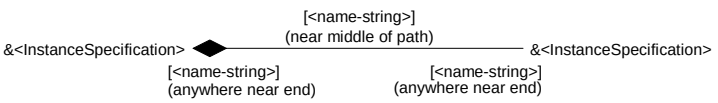
<BidirectionalReferenceLink> ::=



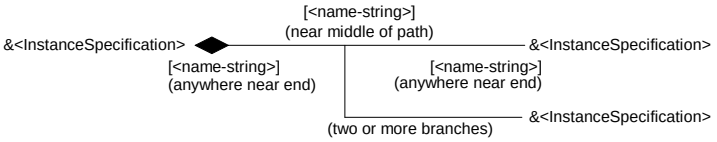
<UnidirectionalReferenceLink> ::=



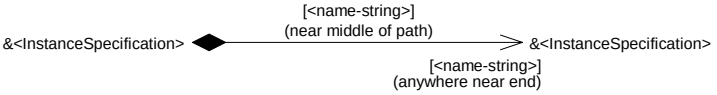
<BidirectionalPartLink> ::=



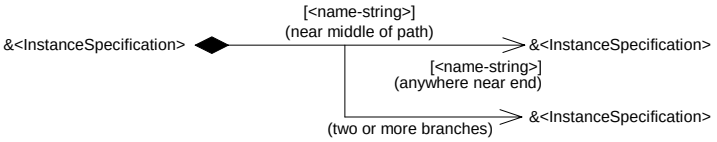
<BidirectionalPartLink> ::=



<UnidirectionalPartLink> ::=

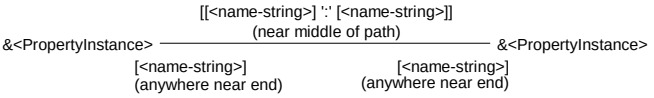


<UnidirectionalPartLink> ::=

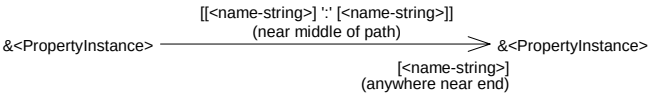


<ConnectorInstance> ::= <BidirectionaConnectorInstance> | <UnidirectionalConnectorInstance>

<BidirectionalConnectorInstance> ::=



<UnidirectionalConnectorInstance> ::=



G.6 Diagram elements Defined in Constraint Blocks Chapter

G.6.1 Diagram Elements Defined in Block Definition Diagrams

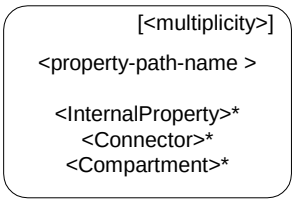
<definition-keyword> ::= ‘constraint’
(additional production for previously defined symbol)

<CompartmentLabel> ::= ‘*parameters*’

G.6.2 Diagram Elements Defined in Internal Block Diagrams

<InternalProperty> ::= <InternalConstraintProperty>
(additional production for existing symbol)

<InternalConstraintProperty> ::=



<property-keyword> ::= ‘constraint’
(additional production for existing symbol)

G.6.3 Diagram Elements Defined in Parametric Diagrams

<InternalBlockDiagram> ::=