Date: 10 January 2005

Systems Modeling Language (SysML) Specification

version 0.9
DRAFT

SysML Partners (www.sysml.org)

American Systems Corporation
ARTISAN Software Tools*
BAE SYSTEMS
The Boeing Company
Ceira Technologies
Deere & Company
EADS Astrium GmbH
EmbeddedPlus Engineering
Eurostep Group AB
Gentleware AG*
I-Logix*
International Business Machines*
International Council on Systems Engineering
Israel Aircraft Industries
Lockheed Martin Corporation
Mentor Graphics
Motorola*
National Aeronautics and Space Administration
National Insitute of Standards and Technology
Northrop Grumman
oose.de Dienstleistungen für innovative Informatik GmbH
PivotPoint Technology Corporation
Popkin Software
Raytheon Company
Structured Software Systems Limited
Telelogic AB*
THALES*
Vitech Corporation

* Submitter to OMG UML for Systems Engineering RFP

TRADEMARKS

Preface for OMG Submission

Editorial Comment: This is a draft of a work in progress that is being made available for public review. Please provide review feedback to the contacts listed in Section 0.2.

This Systems Modeling Language (SysML) Specification draft is being submitted to the OMG in response to the UML for SE RFP. Material in this submission that responds directly to the format required by the OMG submission process is localized entirely within this Preface. By separating the information unique to the OMG technical process and submission format, and referencing the applicable portions of the technical specification, we are able to organize the specification in a form that can facilitate further stages of the OMG technology adoption and ISO Publicly Available Specification (PAS) processes.

OMG RFP RESPONSE

The following SysML Partners are current OMG members who have submitted Letters of Intent to the OMG to respond to its UML for SE RFP: ARTISAN Software Tools, Gentleware, IBM, I-Logix, Motorola, Telelogic, and THALES.

The information required by Section 4.9.2 (“Required Outline”) of the UML for SE RFP is provided in the following parts.

PART I of RFP Response

0.1 Copyright Waiver and Trademark Usage

0.2 Submission contact points

The following persons may be contacted for information regarding this submission:

Cris Kobryn (Cris.Kobryn@sysml.org OR Cris.Kobryn@telelogic.com)

Sanford Friedenthal (Sanford.Friedenthal@sysml.org OR Sanford.Friedenthal@lmco.com)

In addition, the following public mailing list is available for providing feedback and requesting information about this specification: SysMLforum@googlegroups.com.

0.3 Guide to material in the submission

An overview of the background, goals and technical content of this proposal is described in Chapter 1 “Scope” of this document.

0.4 Overall design rationale

The design rationales for the language architecture and the specification approach used by this proposal are provided in Chapter 7 “Language Architecture” and Chapter 8 “Language Formalism” of this document.

0.5 Statement of proof of concept

This proposed specification is in the process of being prototyped or implemented by more than one of the submitting organizations.

0.6 Resolution of RFP requirements and requests

The proposed specification makes use of existing OMG specifications and follows OMG guidelines in conformance with Section 5 “General Requirements on Proposals” of the RFP.

Section 6.5 “Mandatory Requirements” of the RFP requires a specific form of matrix that indicates how the proposed solution satisfies each of numbered requirements in the “Specific Requirements on Proposals” section of the RFP. The requirements traceability matrix in Appendix E addresses this requirement, and is included in this section by reference to the Appendix.

0.7 Response to RFP issues to be discussed.

Section 6.7 of the RFP, “Issues to be discussed” contains a single issue, which requests a sample problem description as follows:

> Submissions shall include models of one or more sample problems to demonstrate how their customization of UML for SE addresses the requirements of this RFP. The submitter may select one or more sample problems of their choosing, or apply their proposed solution to the sample problem descriptions included on the RFP page at http://syseng.omg.org/UML_for_SE_RFP.htm. The compliance matrix referred to in Section 6.5, must include a reference to the portion of the sample problem, which demonstrates how each requirement is being addressed.

The response to this “Issue to be discussed” is provided in Appendix B “Sample Problem” of this document.

PART II of RFP Response

0.8 Proposed specification

The proposed specification is contained in the body of this document (including appendices). This specification includes both normative and explanatory material in a format that is largely self-contained and which could be adopted and published in conformance with the OMG process.

0.9 Proposed compliance points

Proposed compliance points are described in Chapter 2 “Compliance” of this specification.

PART III of RFP Response

0.10 Summary of requests versus requirements

See Section 0.6 in this Preface.

0.11 Changes or extensions required to adopted OMG specifications

See Section 6.1.

Table of Contents

Part I. Introduction

This specification defines a general-purpose modeling language for systems engineering applications, called the Systems Modeling Language (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 orgins 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.

Since SysML is being defined as a UML 2.0 Profile, SysML reuses the mature syntax and semantics of the second generation of UML, UML 2.0. As a consequence, systems engineers modeling with SysML and software engineers modeling with UML 2.0 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.

The next two chapters describe the SysML language architecture and the specification approach used to define SysML.

1 Scope

Editorial Comment: This is a draft of a work-in-progress that is being made available for public review. The scope statement below pertains to the final version of this specification, version 1.0, not the current draft. Please provide review feedback to the public mailing list described below.

The purpose of this document is to specify Systems Modeling Language (SysML), a new general-purpose modeling language for systems engineering. This specification documents the concrete syntax (notation), abstract syntax, semantics, and rationales of SysML, and provides examples of how it can be used to solve common systems engineering problems. Its intent is to specify the langauge 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. The following public mailing list is available for providing feedback and requesting information about this specification: SysMLforum@googlegroups.com.

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, system structure, functional behavior, and allocations during specification and design phases of systems engineering. The first version of the language does not support decision trees, testing, complete trade studies, comprehensive verification and validation, or fully executable functional behavior. These gaps may be addressed in future versions of SysML.

SysML is being aligned with two evolving interoperability standards: the ISO AP-233 data interchange standard for systems engineering tools and the OMG XMI 2.0 model interchange standard for UML 2.0 modeling tools. While the details of this alignment are beyond the scope of this specification, overviews of alignment issues and relevant references are furnished in Appendix F and Appendix G.

The following sections provide background information about this specification, including information about compliance and a glossary of terms. Instructions to either systems engineers and vendors who read this specification are provided in Section 6.2, '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 Compliance

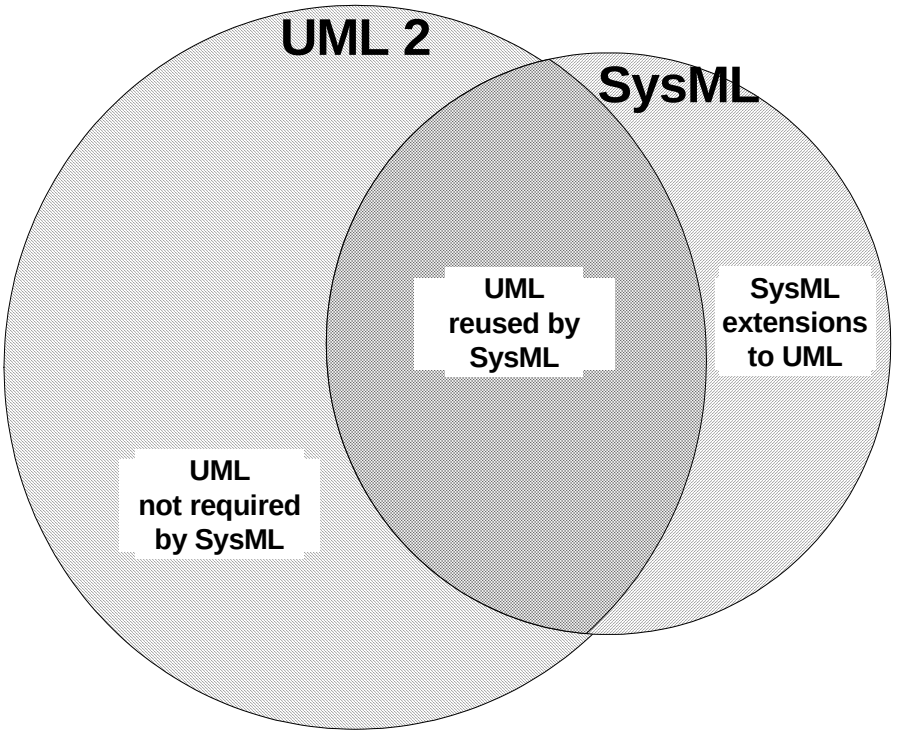
As with UML, the basic units of compliance for SysML are the packages which define the SysML metamodel. A summary of these packages is provided in Chapter 7, "Language Architecture." There are two kinds of SysML compliance. The first kind of compliance is concerned with defining the subset of UML 2 Superstructure (UML) packages required to implement SysML. The second kind of compliance is concerned with specifying the extent to which a SysML tool implements the SysML extensions to UML Superstructure.

In order to visulalize the relationship between the UML and SysML languages, consider the Venn diagram shown in Figure 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 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 1 indicates the new modeling constructs defined for SysML which have no counterparts in UML, or replace UML constructs. The compliance matrix in Table 2 below specifies two levels of SysML packages that a SysML tool must implement in order to provide these extensions.

Note that there is also a substantial part of UML 2 that is not required to implement SysML, which is shown by the region

marked “UML not required by SysML.” .



Editorial Comment: Details regarding the SysML package structure and compliance with UML 2 and SysML are still being sorted out.

The compliance matrix in Table 1 below specifies the UML 2 Superstructure packages that a SysML tool must reuse in order to implement SysML. Stated otherwise, these UML 2 Superstructure packages must be available for any SysML implementation. The valid options are shown below. The package structure in the individual chapters also shows which packages are required for SysML.

- **no**: SysML does not require this UML package. However, SysML is intended to be compatible with the package if it is used.
- **partial**: SysML only requires selected classes from the package.
- **complete:** SysML requires the complete UML package. Implementation fully complies with the concrete syntax, abstract syntax, well-formedness rules, and semantics of the package

Table 1 Summary of Which UML 2 Superstructure Packages are Required for SysML

Compliance Level	Compliance Point	UML Package Required for SysML
Basic (Level 1)	All packages	complete
Intermediate (Level 2)	Actions::IntermediateActions	partial
Intermediate (Level 2)	Activities:: IntermediateActivities	partial

Table 1 Summary of Which UML 2 Superstructure Packages are Required for SysML

Intermediate (Level 2)	Activities:: StructuredActivities	partial
Intermediate (Level 2)	CommonBehaviors:: Communications	partial
Intermediate (Level 2)	CommonBehaviors::Time	partial
Intermediate (Level 2)	Components::BasicComponents	no
Intermediate (Level 2)	CompositeStructures::Actions	partial
Intermediate (Level 2)	CompositeStructures::Ports	partial
Intermediate (Level 2)	CompositeStructures:: StructuredClasses	partial
Intermediate (Level 2)	Deployments::Artifacts	no
Intermediate (Level 2)	Deployments::Nodes	no
Intermediate (Level 2)	Interactions::Fragments	partial
Intermediate (Level 2)	Profiles	partial
Intermediate (Level 2)	StateMachines:: BehaviorStateMachines	partial
Intermediate (Level 2)	StateMachines:: MaximumOneRegion	no
Complete (Level 3)	Actions::CompleteActions	partial
Complete (Level 3)	Activities::CompleteActivities	partial
Complete (Level 3)	Activities:: CompleteStructuredActivities	partial
Complete (Level 3)	Activities:: ExtraStructuredActivities	partial
Complete (Level 3)	AuxiliaryConstructs:: InformationFlows	partial
Complete (Level 3)	AuxiliaryConstructs:: Models	partial
Complete (Level 3)	AuxiliaryConstructs::Templates	no
Complete (Level 3)	Classes:: AssociationClasses	no
Complete (Level 3)	Classes:: PowerTypes	no
Complete (Level 3)	CompositeStructures:: Collaborations	no

Table 1 Summary of Which UML 2 Superstructure Packages are Required for SysML

Complete (Level 3)	Components:: PackagingComponents	no
Complete (Level 3)	Deployments:: ComponentDeployments	no
Complete (Level 3)	StateMachines::ProtocolStateMachines	no

The compliance matrix in Table 2 below specifies the levels of compliance for a tool vendor to comply with the SysML Specification. The following compliance options are valid:

- **no** compliance: Implementation does not comply with the concrete syntax, abstract syntax, well-formedness rules, semantics, and XMI schema of the package.
- **basic** compliance: Implementation complies with the basic features of the package.
- **advanced** compliance: Implementation complies with the advanced features of the package.

For an implementation of SysML to comply with a particular SysML package requires complying with any packages on which the particular package depends. For SysML, this includes not only other SysML packages, but all UML packages on which the SysML package depends.

Table 2 Summary of SysML Compliance Points

Compliance Level	Compliance Point	Valid Options
Basic SysML	All packages, all Basic level constructs	full, interchange
Basic	Activities, Basic level constructs	no, partial, full, interchange
Basic	Allocations, Basic level constructs	no, partial, full, interchange
Basic	Assemblies, Basic level constructs	no, partial, full, interchange
Basic	Auxiliary Constructs, Basic level constructs	no, partial, full, interchange
Basic	Classes, Basic level constructs	no, partial, full, interchange
Basic	Interactions, Basic level constructs	no, partial, full, interchange
Basic	Parametrics, Basic level constructs	no, partial, full, interchange
Basic	Requirements, Basic level constructs	no, partial, full, interchange
Basic	State Machines, Basic level constructs	no, partial, full, interchange
Basic	Use Cases, Basic level constructs	no, partial, full, interchange
Advanced	All packages, all Advanced level constructs	full, interchange
Advanced	Activities, Advanced level constructs	no, partial, full, interchange
Advanced	Allocations, Advanced level constructs	no, partial, full, interchange
Advanced	Assemblies, Advanced level constructs	no, partial, full, interchange
Advanced	Axuiliary Constructs, Advanced level constructs	no, partial, full, interchange

Table 2 Summary of SysML Compliance Points

Advanced	Classes, Advanced level constructs	no, partial, full, interchange
Advanced	Interactions, Advanced level constructs	no, partial, full, interchange
Advanced	Parametrics, Advanced level constructs	no, partial, full, interchange
Advanced	Requirements, Advanced level constructs	no, partial, full, interchange
Advanced	State Machines, Advanced level constructs	no, partial, full, interchange
Advanced	Use Cases, Advanced level constructs	no, partial, full, interchange

3 Normative references

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

- UML 2.0 Superstructure Specification
- UML 2.0 Infrastructure Specification

4 Terms and definitions

Editorial Comment: The terms and definitions in this lexicon are still being unified from SysML, UML and INCOSE sources, and many inconsistencies exist in the current draft. Note that the latest draft of the OMG UML2 Superstructure (ptc/04-10-02) no longer supports a Glossary.

For the purposes of this specification, the terms and definitions given in the following apply. The lexicon contains a mix of terms and definitions that are being unified from various UML, SysML, and INCOSE sources, including previous OMG UML glossaries, UML for SE RFP (ad/03-03-41) and the INCOSE MDSD Concept Model Semantic Dictionary (work in progress).

(Note: The following conventions are used in the term definitions below:

- The entries usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.
- When one or more words in a multi-word term is enclosed in brackets, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.
- A phrase of the form “Contrast: <term>” refers to a term that has an opposed or substantively different meaning.
- A phrase of the form “See: <term>” refers to a related term that has a similar, but not synonymous meaning.
- A phrase of the form “Synonym: <term>” indicates that the term has the same meaning as another term, which is referenced.
- A phrase of the form “Acronym: <term>” indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.)
- A term followed by [**UML for SE RFP**] or a phrase preceded by [**UML for SE RFP**] has been added to the UML 2

glossary to integrate the definitions referenced in the UML for SE RFP (ad/03-03-41).

- A term followed by [**SysML**] or a phrase preceded by [**SysML**] means that this term was not originally in the UML for SE Definitions List but it is an additional definition introduced by SysML. <*> means the term or definition has been modified from the original definition. The modification is noted in *italics.*
- A term followed by [**UML 2 UPDATE**] or a phrase preceded by [**UML 2 UPDATE**] means that it was added to the glossary based on the original supersturcture specification.

abstract class
A class that cannot be directly instantiated. Contrast: *concrete class*.
abstraction
The result of empasizing certain features of a thing while de-emphasizing other features that are not relative. An abstraction is defined relative to the perspective of the viewer.
action
A fundamental unit of behavior specification that represents some transformation or processing in the modeled system, be it a computer system or a real-world system. Actions are contained in activities, which provide their context. See: *activity.* [**UML for SE RFP:** A non-interruptible function. Note: An action represents an atomic unit of processing or work. Actions may be continuous or discrete. Discrete actions may or may not be assumed to execute in zero time. See *function.*]
action sequence
An expression that resolves to a sequence of actions. [**UML for SE RFP:** See *scenario*.]
action state
A state that represents the execution of an atomic action, typically the invocation of an operation.
activation
The initiation of an action execution. [**UML for SE RFP:** See *activation/deactivation event, control input.*]
activation/deactivation event [**UML for SE RFP**]
An event that occurs when a function is activated or deactivated. See *activation, control input*.
activation/deactivation requirement [**UML for SE RFP**]
The activation or deactivation that one or more functions must satsify when specified events and conditions occur. See *requirement.*
activation/deactivation rules [**UML for SE RFP**]
The logic which determines when one or more functions are activated and deactivated.
activation time [**UML for SE RFP**]
The interval of time that a function or state is active. See *time expression*.
active class
A class whose instances are active objects. See: *active object.*
active object
An object that may execute its own behavior without requiring method invocation. This is sometimes referred to as “the object having its own thread of control.” The points at which an active object responds to communications from other objects are determined solely by the behavior of the active object and not by the invoking object. This implies that an active object is both autonomous and interactive to some degree. See: *active class, thread.*
activity
A specification of parameterized behavior that is expressed as a flow of execution via a sequencing of subordinate units (whose primitive elements are individual actions). See *actions*. [**UML for SE RFP**: 1) One or more related actions. 2) [**SysML:** Usage of a function. See *action, function.*]
activity diagram
A diagram that depicts behavior using a control and data-flow model.
activity partition
NEW: See *partition* - definition 1.
actor
A construct that is employed in use cases that define a role that a user or any other system plays when interacting

with the system under consideration. It is a type of entity that interacts, but which is itself external to the subject. Actors may represent human users, external hardware, or other subjects. An actor does not necessarily represent a specific physical entity. For instance, a single physical entity may play the role of several different actors and, conversely, a given actor may be played by multiple physical entities. [**UML for SE RFP:** See *environment, user*.]

actual parameter

Synonym: *argument*.

aggregate

A class that represents the “whole” in an aggregation (whole-part) relationship. See: *aggregation*.

aggregation

A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. See: *composition*. [**UML for SE RFP:** See *decomposition*.]

allocated element [SysML]

A stereotype of an element that is the client or supplier of an allocation with properties allocatedFrom or allocatedTo. See *allocation*.

allocation [**SysML**]

A stereotype on a usage dependency that represents a mapping between one set of model elements (supplier) and another (client). The mapping is often performed as part of the design process to refine the design. Typical examples of allocation include allocation of functions to components, logical to physical components, flows to connectors, and software to hardware. The allocation of requirements to components is generally accomplished using the SysML satsify relationship. See *allocated element*.

analysis

The phase of the system development process whose primary purpose is to formulate a model of the problem domain that is independent of implementation considerations. Analysis focuses on what to do; design focuses on how to do it. Contrast: *design*. [**UML for SE RFP:** The process of evaluating elements, properties and associated relationships.]

analysis model [**UML for SE RFP**]

A model used to analyze the structure, behavior, and/or properties of systems and environments.

analysis time

Refers to something that occurs during an analysis phase of the software development process. See: *design time, modeling time.*

AP-233 [**UML for SE RFP**]

ISO STEP Application Protocol for Systems Engineering Data Interchange Standard

argument

A binding for a parameter that is resolved later. An independent variable.

artifact

A physical piece of information that is used or produced by a development process. Examples of Artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component. Synonym: *product*. Contrast: *component*.

assembly [**SysML**]

A class that describes a structure of interconnected parts.

association

A relationship that may occur between instances of classifiers.

association class

A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end

The endpoint of an association, which connects the association to a classifier.

attribute

A structural feature of a classifier that characterizes instances of the classifier. An attribute relates an instance of a classifier to a value or values through a named relationship.

auxiliary class

A stereotyped class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. Auxiliary classes are typically used together with focus classes, and are particularly useful for specifying the secondary business logic or control flow of components during design. See also: *focus*.

behavior

The observable effects of an operation or event, including its results. It specifies the computation that generates the effects of the behavioral feature. The description of a behavior can take a number of forms: interaction, statemachine, activity, or procedure (a set of actions). [**UML for SE RFP**:The activation/deactivation of one or more functions. Note: This describes how a system interacts with its environment. Reactive behavior includes the stimulus and response.]

behavior allocation [**UML for SE RFP**]

The allocation of functions and/or states to systems, and the allocation of inputs and outputs to system ports.

behavior diagram

A form of diagram that depict behavioral features.

behavioral feature

A dynamic feature of a model element, such as an operation or method.

behavioral model aspect

A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association

An association between two classes. A special case of an n-ary association.

binding

The creation of a model element from a template by supplying arguments for the parameters of the template.

boolean

An enumeration whose values are true and false.

boolean expression

An expression that evaluates to a boolean value.

call

An action state that invokes an operation on a classifier.

cardinality

The number of elements in a set. Contrast: *multiplicity*.

category [**UML for SE RFP**]

A partitioning of elements based on a classification.

central buffer node [From: Superstructure 12.3.9 CentralBufferNode**]**

An object flow for managing flows from multiple sources and destinations.

child

In a generalization relationship, the specialization of another element, the parent. See: *subclass*, *subtype*. Contrast: *parent*.

class

A classifier that desctibes of a set of objects that share the same specifications of features, constraints, and semantics.

classifier

A collection of instances that have something in common. A classifier can have features that characterize its instances. Classifiers include interfaces, classes, datatypes, and components.

classification

The assignment of an instance to a classifier. See *dynamic classification*, *multiple classification* and *static classification*.

class diagram

A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

client

A classifier that requests a service from another classifier. Contrast: *supplier.*

collaboration
The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction. See: *interaction*.
collaboration occurrence
A particular use of a collaboration to explain the relationships between the parts of a classifier or the properties of an operation. It may also be used to indicate how a collaboration represents a classifier or an operation. A collaboration occurrence indicates a set of roles and connectors that cooperate within the classifier or operation according to a given collaboration, indicated by the type of the collaboration occurrence. There may be multiple occurrences of a given collaboration within a classifier or operation, each involving a different set of roles and connectors. A given role or connector may be involved in multiple occurrences of the same or different collaborations. See: *collaboration*.
comment [**UML 2 Update**]
A textutal annotation that can be attached to a set of elements.
communication diagram
A diagram that focuses on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numberering scheme. Sequence diagrams and communication diagrams express similar information, but show it in different ways. See: *sequence diagram*.
compile time
Refers to something that occurs during the compilation of a software module. See: *modeling time, run time*.
complex number [**UML for SE RFP**]
A number which includes a real and imaginary part.
component
A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). [**UML for SE RFP:** A constituent part of an item or system that contributes to the properties and behaviors of the whole (emergent). Note: A leaf component does not have constituent parts. See *item, system, structured class*.]*
component diagram
A diagram that shows the organizations and dependencies among components.
composite
A class that is related to one or more classes by a composition relationship. See: *composition*.
composite aggregation
Synonym: *composition*.
composite function [**UML for SE RFP**]
A function which is decomposed into lower level functions. See *function.*
composite state
A state that consists of either concurrent (orthogonal) substates or sequential (disjoint) substates. See: *substate*. [**UML for SE RFP:** A state which includes nested states.]
composite structure diagram
A diagram that depicts the internal structure of a classifier, including the interaction points of the classifier to other parts of the system. It shows the configuration of parts that jointly perform the behavior of the containing classifier. The architecture diagram specifies a set of instances playing parts (roles), as well as their required relationships given in a particular context.
composition
A form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts. Composition may be recursive. Synonym: *composite aggregation*.
concrete class
A class that can be directly instantiated. Contrast: *abstract class.*

concurrency
The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. See: *thread.*
concurrent substate
A substate that can be held simultaneously with other substates contained in the same composite state. See: *composite state.* Contrast: *disjoint substate.*
condition [**UML for SE RFP**]
An expression with a discrete output, which is true as long as the expression evaluates true, and is false otherwise. See *guard condition*.
connectable element
An abstract metaclass representing model elements which may be linked via connector. See: *connector*.
connecting component [**UML for SE RFP**]
A specialized component or system, whose primary function is to connect the outputs from one system to the inputs of another system via its ports. Note: This may be a wire, network, or mechanical coupler that has properties and behaviors, which may transform the inputs and outputs. See *component*.
connection [**UML for SE RFP**]
Identification of which ports connect to one another. See *connector*.
connection path [**UML for SE RFP**]
Multiple connections that may represent a single logical connection.
connector
A link that enables communication between two or more instances. The link may be realized by something as simple as a pointer or by something as complex as a network connection. [**UML for SE RFP:** See *connection, connection path, connecting component*.]
constraint
A semantic condition or restriction. It can be expressed in natural language text, mathematically formal notation, or in a machine-readable language for the purpose of declaring some of the semantics of a model element.
container
1. An instance that exists to contain other instances, and that provides operations to access or iterate over its contents. (for example, arrays, lists, sets).
2. A component that exists to contain other components.
containment hierarchy
A namespace hierarchy consisting of model elements, and the containment relationships that exist between them. A containment hierarchy forms a graph.
context
A view of a set of related modeling elements for a particular purpose, such as specifying an operation. [**UML for SE RFP**: See *System context*.]
continuous rate [**SysML**]
A subclass of rate which enables a parameter value to be sampled at an infinite rate. See *rate, discrete rate*.
continuous time model [**UML for SE RFP**]
A model which is based on properities that vary continuously with time.
control input [**UML for SE RFP**]
An input that activates or deactivates a function. See *activation, activation/deactivation event*.
control node [From: Superstructure 12.3.13 ControlNode**]**
An activity node used to coordinate the flows between other nodes. It covers initial node, final node, and its children, fork node join node, decision node, and merge node.**[UML for SE RFP:** See *control operator*.**]**
control operator [**UML for SE RFP**]
A specialized function that provides logic to transform input events and conditions to discrete values that are supplied as control inputs to functions. [**SysML:** A stereotype of an activity that can control execution of other activities by outputting control values.See *control value*.] See *control node, decision, fork, join, merge.*
control value [**SysML**]
An enumerated value produced by a control operator to control the execution of an activity. See *control operator.*

data [**UML for SE RFP**]
A component of information.
data type
A type whose values have no identity (i.e., they are pure values). Data types include primitive built-in types (such as integer and string) as well as enumeration types.
decomposition [**UML for SE RFP**]
A description of a whole in terms of its component parts. See *aggregation*.
decision node [From: Superstructure 12.3.15 DecisionNode**]**
A control node that choosed between outgoing flows.
delegation
The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast: *inheritance*.
dependency
A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).
dependency set [**SysML**]
A grouping of dependencies that may have a common supplier or client.
deployment [**UML for SE RFP**]
A dependency relationship between components, where one component depends on the hosting component (i.e node) for resources in order to r perform its functions. See *deployment diagram*.*
deployment diagram
A diagram that depicts the execution architecture of systems. It represents system artifacts as nodes, which are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. See: *component diagrams*.
derive
A dependency relationship between two requirements in which a client requirement can be generated or inferred from the supplier requirements or additional design information. Derived requirements may refine or restate a requirement to improve stakeholder communications or to track design evolution.
design
The phase of the system development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system. [**UML for SE RFP:** The process of transforming requirements to an implementation. See *requirement.*]
design constraint [**UML for SE RFP**]
A requirement that one or more components of a system must satisfy. Note: This term is sometimes used to refer to a constraint on the design process versus the system.See *requirement.*
design time
Refers to something that occurs during a design phase of the system development process. See: *modeling time*. Contrast: *analysis time*.
development process
A set of partially ordered steps performed for a given purpose during system development, such as constructing models or implementing models.
diagram
A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the diagrams listed in Appendix A. See *diagram usage*. [**UML for SE RFP:** A graphical representation of a model view. UML for SE RFP supports the diagrams listed in Appendix A. **NEW]**
diagram description [**SysML**]
A comment that provides standardized information about a diagram.See *reference data*.
diagram interchange [**UML for SE RFP**]

The ability to exchange notational information on a diagram, including the layout of the diagram.

diagram usage [**SysML**]

A form of stereotype applied to a diagram to constrain its use. See *diagram.*

dimension[**SysML**]

A type of value expressed by a quantity. See *quantity, unit*.

discrete rate [**SysML**]

A subclass of rate which enables a parameter value to be sampled at a finite rate. See *rate, continuous rate*.

discrete time model [**UML for SE RFP**]

A model which is based on properities that vary discretely with time.

disjoint substate

A substate that cannot be held simultaneously with other substates contained in the same composite state. See: *composite state*. Contrast: *concurrent substate*.

distributed quantity [**SysML**]

A quantity with an associated probability distribution on its values. See *quantity, distribution definition, distribution result.*

distribution definition [**SysML**]

A parametric constraint that constrains the value of one of its properties to be selected from within a range of possible values. Synonym *probability distribution.* See *distributed quantity, distribution result.*

distribution result [**SysML**]

A stereotype of property that designates the distributed value defined by a disribution definition. See *distributed quantity, distribution definition.*

distribution unit

A set of objects or components that are allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

domain

An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area. [**UML for SE RFP**] A scope that encompasses a set of entities and relationships that may be addressed by the model.

dynamic classification

The assignment of an instance from one classifier to another. Contrast: *multiple classification, static classification*.

effectiveness measure [**UML for SE RFP**]

A criterion for system optimization that is critcial to the success of the mission. Note: The criterion are often used to support trade studies to select among alternatives,as well as to optimize a given design.

EIA 632 [**UML for SE RFP**]

A process standard for Engineering a System.

element

A constituent of a model.

enabling system [**UML for SE RFP**]

Any system which may support another system throughout its life cycle, and typically includes the development, production, deployment, support, and disposal systems.

entry action

An action that a method executes when an object enters a state in a state machine regardless of the transition taken to reach that state.

enumeration

A data type whose instances a list of named values. For example, RGBColor = {red, green, blue}. Boolean is a predefined enumeration with values from the set {false, true}.

environment [**UML for SE RFP**]

A collection of systems and items that interact either directly or indirectly with the system of interest.See *item, system*.*

event

The specification of a significant occurrence that has a location in time and space and can cause the execution of

an associated behavior. In the context of state diagrams, an event is an occurrence that can trigger a transition. [**UML for SE RFP:** A noteworthy occurrence that occurs at the instant of time when a specified expression evaluates true.]

exception

A special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. The receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified.

execution [**UML for SE RFP**]

The state of the system or model when it is running. For a model, this implies that model computation is occuring.

execution occurrence

A unit of behavior within the lifeline as represented on an interaction diagram.

exit action

An action that a method executes when an object exits a state in a state machine regardless of the transition taken to exit that state.

export

In the context of packages, to make an element visible outside its enclosing namespace. See: *visibility.* Contrast: *export* [OMA], *import*.

expression

A string that evaluates to a value of a particular type. For example, the expression “(7 + 5 * 3)” evaluates to a value of type number.

extend

A relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See *extension point*, *include*.

extension

An aggregation that is used to indicate that the properties of a metaclass are extended through a stereotype, and that gives the ability to flexibly add and remove stereotypes from classes.

extension point [From: Superstructure 16.3.4 ExtensionPoint**]**

Identifies a point in the behavior of a use case where that behavior can be extended by the behavior of some other extending use case as specified by an extend relationship .

facade

A stereotyped package containing only references to model elements owned by another package. It is used to provide a ‘public view’ of some of the contents of a package.

facility [**UML for SE RFP**]

A physical infrastructure that supports use of equipment and other resources. See *component*.

failure [**UML for SE RFP**]

An inability to satsify a requirement. See *requirement.*

feature

A property, such as an operation or attribute, that characterizes the instances of a classifier.

final state

A special kind of state signifying that the enclosing
composite state or the entire state machine is completed.

fire

To execute a state transition. See: *transition*.

focus class

A stereotyped class that defines the core logic or control flow for one or more auxiliary classes that support it. Focus classes are typically used together with one or more auxiliary classes, and are particularly useful for specifying the core business logic or control flow of components during design. See also: *auxiliary class*.

focus of control

A symbol on a sequence diagram that shows the period of time during which an object is performing an action,

either directly or through a subordinate procedure.

fork [**UML for SE RFP**]

A control operator which enables all of its outputs, when the input is evaluated true.

formal parameter

Synonym: *parameter*.

framework

A stereotyped package that contains model elements which specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks. See: *pattern*.

function [**UML for SE RFP**]

A transformation of inputs to outputs that may include the creation, monitoring, modification or destruction of elements, or a null transformation.

function port [**UML for SE RFP**]

A binding of an input/output to the arguments of a function. See *argument, input/output, pin*.

function time-line [**UML for SE RFP**]

A representation of the interval of time that one or more functions and/or states are active and inactive.

functional requirement [**UML for SE RFP**]

A function a system must perform. See *requirement.*

generalizable element

A model element that may participate in a generalization relationship. See: *generalization*.

generalization

A taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier indirectly has features of the more general classifier. See: *inheritance.* [**UML for SE RFP:** See *specialization*]

geometric model [UML for SE RFP]

A model of the geometric relationships associated with one or more elements. See *spatial representation*.

guard condition

A condition that must be satisfied in order to enable an associated transition to fire. [**UML for SE RFP:** See *condition.*]

hardware [**UML for SE RFP**]

A component of a system that has geometric contstraints. See *component*.

IDEF0 [**UML for SE RFP**]

Air Force Standard for process modeling.

implementation

A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation.

implementation class

A stereotyped class that specifies the implementation of a class in some programming language (e.g., C++, Smalltalk, Java) in which an instance may not have more than one class. An Implementation class is said to realize a type if it provides all of the operations defined for the type with the same behavior as specified for the type's operations. See also: *type*.

implementation inheritance

The inheritance of the implementation of a more general element. Includes inheritance of the interface. Contrast: *interface inheritance*.

import

In the context of packages, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: *export*.

include

A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location which is defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its

structure (i.e., attributes or operations). See *extend*.

inheritance

The mechanism by which more specific elements incorporate structure and behavior of more general elements. See *generalization*.

initial state

A special kind of state signifying the source for a single transition to the default state of the composite state.

input/output [**UML for SE RFP**]

An item that is subject to a transformation by a function. See argument, *function port, parameter, signature*.*

instance

An entity that has unique identity, a set of operations that can be applied to it, and state that stores the effects of the operations. See: *object*. [**UML for SE RFP**: A unique model element in a set that its defined by the general features of its classifier]

integer [**UML for SE RFP**]

A whole number

interaction

A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See *collaboration* .**[UML for SE RFP:** Emergent behavior that results from two or more dependent behaviors Note: A system or component interacts with other components its environment, to yield an emergent system behavior from the individual component behaviors.]

interaction diagram

A generic term that applies to several types of diagrams that emphasize object interactions. These include communication diagrams, sequence diagrams, and the interaction overview diagram.

interaction overview diagram

A disgram that depicts interactions through a variant of
activity diagrams in a way that promotes overview of
the control flow. It focuses on the overview of the flow of control where each node can be an interaction diagram.

interface

A named set of operations that characterize the behavior of an element. [**UML for SE RFP:** The inputs, outputs, ports, connections, connecting components (i.e. harness), and associated information that support one or more interactions between systems. Note: The UML definition of interface also includes the operations that must be performed in response to the inputs or invocations.]

interface requirement [**UML for SE RFP**]

An interface a system must support. See *requirement.*

interface inheritance

The inheritance of the interface of a more general element. Does not include inheritance of the implementation. Contrast: *implementation inheritance*.

internal transition

A transition signifying a response to an event without changing the state of an object.

ISO 15288 [**UML for SE RFP**]

A process standard for system life cycle processes.

issue (technical) [**UML for SE RFP**]

A potential problem, that requires resolution.

item [**UML for SE RFP]**

Anything of interest to the modeler, which is uniquely identifiable and can be characterized by a set of properties. (previously called *element *)* [**SysML:** A classifier that represents the type of thing that flows or the type of thing that is stored (i.e. stored item). See *item flow, item property.*]

item flow [**SysML**]

A subclass of a directed relationship (Note: more precisely a subclass of information flow) that is realized by a relationship that conveys the item(s). See *item, item property.*

item property [**SysML**]

A property that relates the instances of the item to the instances of its enclosing class. See *item, item flow,*

property.

iteration loop [**UML for SE RFP**]

A specialized loop where the loop repeats a specified number of times.

join [**UML for SE RFP**]

A control operator which enables its control output, when all of its inputs are evaluated true

layer

The organization of classifiers or packages at the same level of abstraction. A layer may represent a horizontal slice through an architecture, whereas a partition represents a vertical slice. Contrast: *partition*.

leaf function [**UML for SE RFP**]

A function which is not further decomposed. See *function, action.*

lifeline

A modeling element that represents an individual participant in an interaction. A lifeline represents only one interacting entity.

link

A semantic connection among a tuple of objects. An instance of an association. See: *association*.

link end

An instance of an association end. See: *association end*.

logical component [**SysML**]

a component which is specified in terms `of functionality, state, and logical characteristics and are implementation independent`. See *component, subsystem.* Contrast *physical component.*

manual procedure [**UML for SE RFP**]

A set of operations that provide instructions for a user to perform. See *procedure*.

mean [**UML for SE RFP**]

The expected value associated with a probability distirbution.

merge [**UML for SE RFP**]

A control operator which enables its output, when any of its inputs are evaluated true

message

A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation. [**UML for SE RFP:** See *control input, triggering input*.]

metaclass

A class whose instances are classes. Metaclasses are typically used to construct metamodels.

meta-metamodel

A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metamodel

A model that defines the language for expressing a model.

metaobject

A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method

The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

mission [**UML for SE RFP**]

The operational context and purpose that the system is intended to support.

model (graphical, visual)

A model represents a view of a physical system. It is an abstraction of the physical system, with a certain purpose.

model aspect

A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

model elaboration

The process of generating a repository type from a published model. Includes the generation of interfaces and

implementations which allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model element

An element that is an abstraction drawn from the system being modeled. Contrast: *view element*. [**UML for SE RFP:** A construct that is used to build a model. See *element*.]

model interchange [**UML for SE RFP**]

The ability to exchange model information.

model library

A stereotyped package that contains model elements that are intended to be reused by other packages. A model library differs from a profile in that a model library does not extend the metamodel using stereotypes and tagged definitions. A model library is analogous to a class library in some programming languages.

model view [**UML for SE RFP**]

A *specification of the* subset of model elements and associated relationships, that are of use to the modeler for a particular purpose and context. See *model*. *

modeling time

Refers to something that occurs during a modeling phase of the system development process. It includes analysis time and design time. Usage note: When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns. See: *analysis time, design time*. Contrast: *run time*.

multiple classification

The assignment of an instance directly to more than one classifier at the same time. See: *static classification*, *dynamic classification*.

multiple inheritance

A semantic variation of generalization in which a type may have more than one supertype. Contrast: *single inheritance*.

multiplicity

A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for association ends, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast: *cardinality*.

n-ary association

An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes. Contrast: *binary association*.

name

A string used to identify a model element.

namespace

A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: *name*.

natural object [**UML for SE RFP**]

An *item* that is not engineered, and may be part of a system or environment.See *item.**

need [**UML for SE RFP**]

A desired requirement of a stakeholder. See *requirement.*

nested connector end [**SysML**]

A stereotype of a connector end in which the connected property must be owned directly by an enclosing class or part, so that the connected property may be identified by a multi-level path of accessible properties from the classifier that owns the connector.

nested state [**UML for SE RFP**]

A state which is enabled by its composite state.

no buffer [**SysML**]

A stereotype of an object node that specifies that arriving tokens are discarded if they are not immediately consumed.

node

A classifier that represents a run-time computational resource, which generally has at least memory and often

processing capability. Run-time objects and components may reside on nodes. [**UML for SE RFP:** A component of a system that provides resources to support execution.]

note
An annotation attached to an element or a collection of elements. A note has no semantics. Contrast: *constraint*.

notation [**UML for SE RFP**]
The graphical depiction of a model construct.

object
An instance of a class. See: *class, instance*.

object diagram
A diagram that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a communication diagram. See: *class diagram, communication diagram.*

object flow state
A state in an activity diagram that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object lifeline
A line in a sequence diagram that represents the existence of an object over a period of time. See: *sequence diagram*.

operation
A feature which declares a service that can be performed by instances of the classifier of which they are instances. [**UML for SE RFP:** See *activity, function*.]

operational requirement [**UML for SE RFP**]
A requirement which is associated with the operation of a system, and typically includes a combination of functional, interfrace, and performance requirements. See *requirement.*

optional [**SysML**]
A stereotype of a parameter with lower multiplicity equal to zero indicating that the parameter is not required for the activity to begin execution. Otherwise, the lower multiplicity must be greater than zero, which is call “required”. Synonym *non-tiggering input*.

overwrite [**SysML**]
A stereotype of an object node that specifies that the current value replaces the previous value resulting in a queue of one.

package
A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages.

package diagram
A diagram that depicts how model elements are organized into packages and the dependencies among them, including package imports and package extensions.

parameter
An argument of a behavioral feature. A parameter specifies arguments that are passed into or out of an invocation of a behavioral element like an operation. A parameter’s type restricts what values can be passed. Synonyms: *formal parameter.* Contrast: *argument*.

parametric constraint [**SysML**]
A stereotype of an assembly used only to constrain the values of properties within a containing assembly. The parametric constraint can be viewed as a generic constraint, such as F=m*a, that is used in a particular context. See *paramteric relationship*.

parametric model [**UML for SE RFP**]
An analysis model which defines a set of dependent or logically grouped parametric relationships.

parametric relationship [**UML for SE RFP**]
A dependency between properties, such that a change to the value of one property impacts the value of the other property. See *constraint*. See *parametric constraint*.

parameterized element

The descriptor for a class with one or more unbound parameters. Synonym: *template*.

parent

In a generalization relationship, the generalization of another element, the child. See: *subclass*, *subtype*. Contrast: *child*.

part

An element representing a set of instances that are owned by a containing classifier instance or role of a classifier. (See *role.)* Parts may be joined by attached connectors and specify configurations of linked instances to be created within an instance of the containing classifier. (See *property*)

participate

The connection of a model element to a relationship or to a reified relationship. For example, a class participates in an association, an actor participates in a use case.

partition

A grouping of any set of model elements based on a set of criteria.

1. activity diagram: A grouping of activity nodes and edges. Partitions divide the nodes and edges to constrain and show a view of the contained nodes. Partitions can share contents. They often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the nodes of an activity.
2. architecture: A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice. Contrast: *layer*.

pattern

A template collaboration that describes the structure of
a design pattern. UML patterns are more limited than those used by the design pattern community. In general, design patterns involve many non-structural aspects, such as heuristics for their use and usage trade-offs.

performance property [**UML for SE RFP**]

A measure of the transformation or response of a function or behavior (i.e response time, etc.).

performance requirement [**UML for SE RFP**]

A performance property a system must satsify. See *requirement.*

persistent object

An object that exists after the process or thread that created it has ceased to exist.

physical component [**SysML**]

a component which includes implementaton constraints. See *component, subsystem.* Contrast *logical component.*

physical property [**UML for SE RFP**]

A physical characteristic of a system or element (i.e. weight, color).

physical requirement [**UML for SE RFP**]

A physical property a system must satsify. See *requirement.*

physical system

1. The subject of a model.
2. A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast: *system*.

pin

A model element that represents the data values passed into a behavior upon its invocation as well as the data values returned from a behavior upon completion of its execution. [**UML for SE RFP:** See *function port*., input/ output.]

port

A feature of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to other ports through connectors through which requests can be made to invoke the behavioral features of a classifier. [**UML for SE RFP:** The part of a system or component that provides access between a system's behaviors and properties, and its environment. Note: this is sometimes referred to as an interaction point.]

postcondition

A constraint expresses a condition that must be true at the completion of an operation.

powertype

A classifier whose instances are also subclasses of another classifier. Power types, then, are metaclasses with an extra twist: the instances are also subclasses.

precondition

A constraint expresses a condition that must be true when an operation is invoked.

primitive type

A pre-defined data type without any relevant substructure (i.e., is not decomposable) such as an integer or a string. It may have an algebra and operations defined outside of UML, for example, mathematically.

probability distribution [**UML for SE RFP**]

A mathematical function which defines the likelihood of a paritcular set of outcomes. See *expression*.

probe [**UML for SE RFP**]

A component that monitors the values associated with one or more parameters (i.e. properties).

problem [**UML for SE RFP**]

A deficiency, limitation, or failure to satisfy a requirement or need, or other undesired outcome. Note: A problem may be associated with the behavior, structure, and/or properties of a system or element at any level of the hierarchy (i.e. system of system level, down to a component/part level). See *need, requirement.*

problem cause [**UML for SE RFP**]

The relationship between a problem and its source problems (i.e. cause). Note: This cause ***e***ffect relationship is often represented in fishbone diagrams, fault trees, etc..*****

procedure

A set of actions that may be attached as a unit to other parts of a model, for example, as the body of a method. Conceptually a procedure, when executed, takes a set of values as arguments and produces a set of values as results, as specified by the parameters of the procedure.

process

1. A heavyweight unit of concurrency and execution in an operating system. Contrast: *thread*, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.
2. A software development process—the steps and guidelines by which to develop a system.
3. To execute an algorithm or otherwise handle something dynamically. [**UML for SE RFP:** A set of inter-related functions and their corresponding inputs and outputs, which are activated and deactivated by their control inputs.]

profile

A stereotyped package that contains model elements that have been customized for a specific domain or purpose using extension mechanisms, such as stereotypes, tagged definitions and constraints. A profile may also specify model libraries on which it depends and the metamodel subset that it extends.

projection

A mapping from a set to a subset of it.

property

A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML; others may be user defined. See: *tagged value*. [**SysML**] A usage of a class that relates the instances of the enclosing class to the instances of the class that types the property.[**UML for SE RFP:** A quantifiable characteristic. See *value*.]

property association [**UML for SE RFP**]

The assignment of a property to a model element or set of model elements.

property attribute [**UML for SE RFP**]

Unique state of a property.

pseudo-state

A vertex in a state machine that has the form of a state, but doesn't behave as a state. Pseudo-states include initial and history vertices.

qualifier

An association attribute or tuple of attributes whose values partition the set of objects related to an object across an

association.

quantity[**SysML**]
A stereotype of a data that specifies a value with a dimensions and/or unit of measure. See *dimension, unit, distributed quantity*.

rate [SysML]
A stereotype of parameter that specifies how often objects and values can traverse an activity edge. See *discrete rate, continuous rate.*

rationale [**SysML**]
An element that documents the principles or reasons for a modeling decision, such as an analysis choice or a design selection. It provides or references the basis for the modeling decision, and can be attached to any model element.

real number [**UML for SE RFP**]
A number which can have any value from negative infinity to infinity.

realization
A specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other representing an implementation of the latter (the client). Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.

receive [a message]
The handling of a stimulus passed from a sender instance. See: *sender, receiver*.

receiver
The object handling a stimulus passed from a sender object. Contrast: *sender*.

reception
A declaration that a classifier is prepared to react to the receipt of a signal.

reference
1. A denotation of a model element.
2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym: *pointer*.

refinement
A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relationship
An abstract concept that specifies some kind of connection between elements. Examples of relationships include associations and generalizations.

replicate function [**UML for SE RFP**]
A function which represents the same transformation, but is implemented by separate resources. See *function*.

repository
A facility for storing object models, interfaces, and implementations.

requirement
A statement of a capability or condition that a system must satisfy. A requirement may specify a function that a system must perform or a performance condition that a system must satisfy.

requirement allocation [**UML for SE RFP**]
The assignment of a requirement to an element, component, or system. See *allocation, requirement, requirement satisfaction, trace.*

requirement attribute [**UML for SE RFP**]
An attribute fo a requirement, which may include its criticality or weighting, level of uncertainty, verification status, etc. See *requirement.*

requirement traceability [**UML for SE RFP**]
The relationship between a source requirement and the derived requirements needed to satisfy the source requirement. See *requirement, derive, trace.*

requirement type [**UML for SE RFP**]
A category of requirement. Note: This includes functional, interface, performance, etc.

requirement verification [**UML for SE RFP**]

A comparison between a requirement and the verification results that is intended to satisfy the requirement. See verify, *verdict, verification result.*

resource [**UML for SE RFP**]
Any element that is needed for the execution of a function. See *resource constraint.*

resource constraint [SysML]
A stereotype of a constraint that specifies the types of resources used by the model element it constrains. See *resource.*

responsibility
A contract or obligation of a classifier.

reuse
The use of a pre-existing artifact.

role
The named set of features defined over a collection of entities participating in a particlar context.
Collaboration: The named set of behaviors possessed by a class or part participating in a particular context.
Part: a subset of a particular class which exhibits a subset of features possessed by the class
Associations: A synonym for association end often referring to a subset of classifier instances that are participating in the association. [**UML for SE RFP:** See *part, system role*.]

run time
The period of time during which a computer program or a systemexecutes. Contrast: *modeling time*.

satisfy [**SysML**]
A dependency relationship between a requirement and a model element that fulfills the requirement. See: *derive*, *verify*.

scalable [**UML for SE RFP**]
A measure of the extent to which the modeling langauge (or methodology, etc), can be adapted to an increase in scope and/or complexity.

scenario
A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction or the execution of a use case instance. See: *interaction.*

selection [**UML for SE RFP**]
A control operator which represents a test that enables an output based on the values/conditions of the input.

semantic variation point
A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

semantics [**UML for SE RFP**]
The meaning of a model element. Note: a precise meaning should be able to be expressed mathematically.

send [a message]
The passing of a stimulus from a sender instance to a receiver instance. See: *sender, receiver*.

sender
The object passing a stimulus to a receiver instance. Contrast: *receiver*.

sequence diagram
A diagram that depicts an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding event occurrences on the lifelines.
Unlike a communication diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and communication diagrams express similar information, but show it in different ways. See: *communication diagram*.

sequential state [**UML for SE RFP**]
A state which can only be active when the other sequential states are not active. See *state, concurrent substate.*

signal
The specification of an asynchronous stimulus that triggers a reaction in the receiver in an asynchronous way and without a reply. The receiving object handles the signal as specified by its receptions. The data carried by a send

request and passed to it by the occurrence of the send invocation event that caused the request is represented as attributes of the signal instance. A signal is defined independently of the classifiers handling the signal.

signature

The name and parameters of a behavioral feature. A signature may include an optional returned parameter. [**UML for SE RFP**: See *input/output*.

simple state [**UML for SE RFP**]

A state that does not have nested states.See *state, composite state.*

single inheritance

A semantic variation of generalization in which a type may have only one supertype. Synonym: *multiple inheritance* [OMA]. Contrast: *multiple inheritance*.

slot

A specification that an entity modeled by an instance specification has a value or values for a specific structural feature.

software [**UML for SE RFP**]

A component of a system that specifies instructions which are executed by a computer. See *component*.

software module

A unit of software storage and manipulation. Software modules include source code modules, binary code modules, and executable code modules.

source requirement [**UML for SE RFP**]

The requirement which is the basis for deriving one or more other requirements.

spatial representation [**UML for SE RFP**]

A geometrical relationship among elements. See *geometric model*.

specialization [**UML for SE RFP**]

A classification of an entity (e.g., element, system, function, requirement, ...), which specifies the common features of the more general element, and unique features of the specific element. See *generalization*.

specialized requirement [**UML for SE RFP**]

A requirement that is not explicitly addressed by the default requirement types. Note: This may include safety, reliabillity, maittianability, producibility, usability, security, etc.

specialty engineering [**UML for SE RFP**]

A general term for engineering disciplines associated with some specific aspects of a system, suchas reliability or safety engineering.

specification

A set of requirements for a system or other classifier. Contrast: *implementation*. [**UML for SE RFP:** One or more requirements for a system, component or item.]

stakeholder [**UML for SE RFP**]

Individuals, groups, and/or institutions which may be impacted by the system throughout its life cycle, including acquisition, development, production, deployment, operations, support, and disposal.

state

A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast: *state* [OMA]. [**UML for SE RFP:** A condition of a system or element, as defined by some of its properties, which can enable system behaviors and/or structure to occur. Note: The enabled behavior may include no actions, such as associated with a wait state. Also, the condition that defines the state may be dependent on one or more previous states.]

state-based behavior [**UML for SE RFP**]

Behavior which is described by states and transitions between states.

state machine diagram

A diagram that depicts discrete behavior modeled through finite state-transition systems. In particular, it specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions. See: *state machine*.

state machine

A behavior that specifies the sequences of states that an object or an interaction goes through during its life in

response to events, together with its responses and actions.

static classification

The assignment of an instance to a classifier where the assignment may not change to any other classifier. Contrast: *dynamic classification.*

stereotype

A class that defines how an existing metaclass (or stereotype) may be extended, and enables the use of platform or domain specific terminology or notation in addition to the ones used for the extended metaclass.
Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of the extensibility mechanisms in UML. See: *constraint, tagged value.*

stimulus

The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See: *message*.

storage device [**UML for SE RFP**]

A component of a system that is used to store a stored item. Note: this may include memory device, a battery, or a tank. See *component*, *stored item*. *

store requirement [**UML for SE RFP**]

A stored item a system must store. See *requirement, stored item*. *

stored item [**UML for SE RFP**]

An item that persists over time, which may be depletable or non-depletable. Note: Non-depletable stores may include data store in computer memory, and depletable stores may include energy in a battery, or fluid in a tank.. Physical stores obey the conservation laws (only take out what is put in). A non-depletable store, such as a data store, is not constrained by the conservation laws. The stored item should be differentiated from the storage device, which stores the item. See *central buffer node, storage device*. Note: This was previously a system store. *

streaming [**UML 2 UPDATE**]

A property of a parameter that specifies whether its values can be accepted or produced by an action while executing. A non-streaming parameter specifies that the parameter value can only be accepted at the beginning of execution and produced at the end of execution.

string

A sequence of text characters. The details of string representation depend on implementation, and may include character sets that support international characters and graphics.

structure [**UML for SE RFP**]

The relationships between the components that contribute to the properties of the whole, and enable them to interact (inter-relate).

structural feature

A static feature of a model element, such as an attribute.

structural model aspect

A model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

structure diagram

A form of diagram that depicts the elements in a specification that are irrespective of time. Class diagrams and component diagrams are examples of structure diagrams.

structured class [**UML 2 UPDATE**]

A class that can be decomposed into its parts.

subactivity state

A state in an activity diagram that represents the execution of a non-atomic sequence of steps that has some duration.

subclass

In a generalization relationship, the specialization of another class, the superclass. See: *generalization*. Contrast: *superclass*.

submachine state

A state in a state machine that is equivalent to a

composite state but whose contents are described by another state machine.

substate

A state that is part of a composite state. See: *concurrent state, disjoint state*.

subpackage

A package that is contained in another package.

subsystem

A unit of hierarchical decomposition for large systems. A subsystem is commonly instantiated indirectly. Definitions of subsystems vary widely among domains and methods, and it is expected that domain and method profiles will specialize this construct. A subsystem may be defined to have specification and realization elements. [**UML for SE RFP**:

A logical or physical partitioning of a system. See *system, logical component, physical component*.]

subtype

In a generalization relationship, the specialization of another type, the supertype. See: *generalization*. Contrast: *supertype*.

superclass

In a generalization relationship, the generalization of another class, the subclass. See: *generalization.* Contrast: *subclass*.

supertype

In a generalization relationship, the generalization of another type, the subtype. See: *generalization*. Contrast: *subtype*.

supplier

A classifier that provides services that can be invoked by others. Contrast: *client*.

synch state

A vertex in a state machine used for synchronizing the
concurrent regions of a state machine.

system

An organized array of elements functioning as a unit Also, a top-level subsystem in a model. [**UML for SE RFP:** An *item*, with structure, that exhibits observable properties and behaviors. See *item, component, structured class*. *]

system (component) boundary [**UML for SE RFP**]

The set of all ports, which connect the system (component) to its environment.

system context [**UML for SE RFP**]

A depiction of the inputs and outputs between a system and its environment. See *context*.

system hierarchy [**UML for SE RFP**]

A decomposition of a system and its components.

system interconnection [**UML for SE RFP**]

The connection between systems and between components. See *connector*.

system role [**UML for SE RFP**]

A subset of its behaviors, properties, and structure. Note: The subset may be associated with specific interactions. ation laws. See *role*.

tagged value

The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML; others may be user defined. Tagged values are one of three extensibility mechanisms in UML. See: *constraint, stereotype.*

template

Synonym: *parameterized element*.

test case

A process or activity that is used to determine whether a system has fulfilled its requirements. See *requirement, satisfy*.

test scenario [**UML for SE RFP**]

A scenario which replicates the behavior of the environment that interacts with the system under test.

text-based requirement [**UML for SE RFP**]
One or more requirements specified in text. See *requirement*.
thread [of control]
A single path of execution through a program, a dynamic model, or some other representation of control flow. Also, a stereotype for the implementation of an active object as lightweight process. See *process*. [**UML for SE RFP:** A process with no concurrent functions, and represents a single path of execution.]
time event
An event that denotes the time elapsed since the current state was entered. See: *event*.
time expression
An expression that resolves to an absolute or relative value of time.
time property [**UML for SE RFP**]
A property of the model that represents a local or global time, which other properties may depend on. . Note: The property can support continuous or discrete-time models. This variable should not be confused with the measured or computed time that an actual system uses, which depends on a number of implementation specific factors related to clocks, synchronization, etc. See *property*.
time reference [**SysML**]
The time property from which other time properties are derived. See *time property*.
timing diagram
An interaction diagram that shows the change in state or condition of a lifeline (representing a Classifier Instance or Classifier Role) over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli.
top level
A stereotype denoting the top-most package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces “see” outwards. For example, opLevel subsystem represents the top of the subsystem containment hierarchy.
topology [**UML for SE RFP**]
A graph of nodes and arcs.
trace
A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other. Trace dependencies are used to track requirements and changes across models.
trade-off study [**UML for SE RFP**]
An evaluation of alternatives based on a set of evaluation criteria.
transient object
An object that exists only during the execution of the process or thread that created it.
transition
A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. [**UML for SE RFP:** Response to events/conditions, which triggers a behavior.]
triggering input [**UML for SE RFP**]
An input which is required for a function to be activated.
type
A stereotyped class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type may not contain any methods, maintain its own thread of control, or be nested. However, it may have attributes and associations. Although an object may have at most one implementation class, it may conform to multiple different types. See also: *implementation class*
Contrast: *interface*.
type expression
An expression that evaluates to a reference to one or more types.
uninterpreted
A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has

a corresponding string representation. See: *any* [CORBA].

unit [**SysML**]

A standard for expressing a quantity. See *dimension, quantity*.

usage

A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

user [**UML for SE RFP**]

An individual or group of individuals that use a system. See *actor.*

use case

The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. See: *use case instances.*

use case diagram

A diagram that shows the relationships among actors and the subject (system), and use cases.

use case instance

The performance of a sequence of actions being specified in a use case. An instance of a use case. See: *use case class*.

use case model

A model that describes a system's functional requirements in terms of use cases.

utility

A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience.

validation [**UML for SE RFP**]

The process for demonstrating that a system or its requirements satisfy the stakeholder needs.

value

An element of a type domain.[**SysML:** See *quantity*.]

value binding constraint [**SysML**]

A sterotype of a connector that declares that two connected properties are constrained to have the same value.

variance [**UML for SE RFP**]

A measure of the distribution about the mean of a probability distribution. Refer to the mathematical definition associated with a probability distribution.

vector [**UML for SE RFP**]

A data type, which specifies a magnitude and direction.

verdict [**SysML**]

The outcome of executing one or more test cases or verification procedures. See *test case, verification procedure, verification result, verify.* Note: This term is borrowed from the testing profile.

verification [**UML for SE RFP**]

The process for demonstrating a system satisfies its requirements.

verification procedure [**UML for SE RFP**]

The functions needed to support execution of a test case. Note. This may include generating an input stimulus and monitoring an output response. See *procedure, manual procedure*.

verification result [**UML for SE RFP**]

The outcome of executing one or more test cases.* See *verdict.*

verification system [**UML for SE RFP**]

The system that implements the verification procedures.

verify

A relationship between a requirement and a test case that can determine whether a system fulfills the requirement. See *requirement, test case, verdict.*

vertex

A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See: *state, pseudo-state.*

view
See *model*.
visibility
An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.
well-formedness rule [**UML for SE RFP**]
A rule which specifies the allowable relationships and constraints among model elements.

5 Symbols

Editorial Comment: This section, which is currently not used, will likely be removed in a future revison.

6 Additional information

6.1 Relationships to Other Standards

SysML is defined as an extension of the OMG *UML 2.0 Superstructure Specification* (OMG document number ptc/2004-10-02). If SysML requires any changes to this UML specification, they will be described in a future version of this document.

SysML is also being aligned with two evolving interoperability standards: the ISO AP-233 data interchange standard for systems engineering tools and the OMG XMI 2.0 model interchange standard for UML 2.0 modeling tools. While the details of this alignment are beyond the scope of this specification, overviews of alignment issues and relevant references are furnished in Appendix F and Appendix G.

SysML supports the OMG's Model Driven Architecture initiative by its reuse of the UML standard, and its architectural alignment with the OMG XMI 2.0 and ISO AP-233 interoperability standards.

6.2 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 - "Structure", Part III - "Behavior", and Part IV - "Cross Cutting". Systems engineers should read the Overview, Diagram Elements and Usage Examples sections in each chapter, and explore the Package Structure, UML Extension and Compliance level sections as they see fit. Modeling tool vendors should read all sections. In addition, Systems engineers who want to understand how to apply the language and assess its coverage should read the Appendix B - "Sample Problem" and Appendix E - "Requirements Traceability" respectively.

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

6.3 Acknowledgements

The following companies submitted or supported parts of this specification:

- Industry

    - BAE SYSTEMS
    - Boeing
    - Deere & Company
    - EADS Astrium
    - EmbeddedPlus Engineering
    - Eurostep
    - Israel Aircraft Industries
    - Lockheed Martin Corporation
    - Motorola
    - Northrop Grumman
    - oose.de Dienstleistungen für innovative Informatik GmbH
    - PivotPoint Technology
    - Raytheon
    - THALES
- 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
    - Gentleware
    - IBM
    - I-Logix
    - Mentor Graphics
    - Popkin Software
    - Telelogic
    - Structured Software Systems Limited
    - Vitech
- Liaisons
    - Consultative Committee for Space Data Systems (CCSDS)
    - Embedded Architecture and Software Technologies (EAST)
    - International Council on Systems Engineering (INCOSE)
    - ISO STEP AP-233
    - Systems Level Design Language (SLDL) and Rosetta

The following persons were members of the core team that designed and wrote this specification: Vincent Arnould, Laurent Balmelli, Ian Bailey, James Baker, Conrad Bock, Carolyn Boettcher, Roger Burkhart, Murray Cantor, Bruce Douglass, Harald Eisenmann, Marilyn Escue, Sanford Friedenthal, Eran Gery, Drora Goshen, Hal Hamilton, Dwayne Hardy, James Hummel, Cris Kobryn, Michael Latta, Robert Long, Alan Moore, Veronique Normand, Salah Obeid, David Price, Joseph Skipper, Rick Steiner, Robert Thompson, Jim U'Ren, Tim Weilkiens, 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, Mike Dickerson, Orazio Gurrieri, Julian Johnson, Jim Long, Henrik Lönn, Dave Oliver, Jim Schier, Matthias Weber, Bran Selic, Peter Shames and Thomas Weigert.

7 Language Architecture

The SysML specification is defined using a profiling and metamodeling approach that adapts formal specification techniques. While this approach lacks some of the rigor of a formal specification method, it offers the advantages of being more intutitive and pragmatic for most implementors and practitioners.[1] This chapter explains design principles and how they are applied to define the SysML language architecture.

7.1 Design Principles

The fundamental design principles for SysML are to provide:

- Parsimony. SysML is based on a subset of UML that economically satisfies the basic requirements of the systems engineering community as defined in the UML for SE RFP. Additional constructs and diagram types are added to this UML subset as necessary to address other SE requirements. This surgical reduction and augmentation of UML constructs is intended to make SysML easier to learn, implement and apply.

- Reuse. SysML strictly reuses UML wherever practical, 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 or later versions.

- Modularity. This principle of strong cohesion and loose coupling is applied to group constructs into packages and organize features into metaclasses, stereotypes and model library classes.

- Layering. Layering is applied in two ways to the SysML metamodel. First, SysML packages are specified as an extension layer to the UML metamodel. Second, a 4-layer metamodel architectural pattern is consistently applied to separate concerns (especially regarding instantiation) across layers of abstraction.

- Partitioning. Partitioning is used to organize conceptual areas within the same layer. In general, SysML partitioning strives to be consistent with the UML package partitioning to facilitate reuse and implementation.

- Extensibility. SysML furnishes the same extension mechanisms furnished by UML (metaclasses, stereotypes, model libraries), so that the language can be further extended for specific systems engineering domains, such as automotive, aerospace, manufacturing and communications.

- Interoperability. SysML is aligned with the semantics of the ISO AP-233 data interchange standard to support interoperability among engineering tools, and inherits the XMI interchange from UML.

7.2 Architecture

The SysML langauge reuses and extends many of the packages from UML, as is shown in Figure 7-1. Several extensions mechanisms are used including stereotypes, metaclasses and model libraries. The SysML user model is created by instantiating the stereotypes and metaclasses specified in the SysML metamodel and subclassing the classes in the SysML model library.

In order to understand the SysML package structure, it is helpful to understand the UML Superstructure package structure which it extends. UML 2 Superstructure package structure is shown in Figure 7-2. Each package contains metaclasses that define the basic language constructs. The dependencies between the packages are shown as dashed arrows. The metaclasses and their interrelationships in a package are specified in the UML Specification as one or more class diagrams to specify the

---

1. The specification of SysML as a metamodel does not preclude it from being later specified via a mathematically formal language (e.g., VDM, Object-Z).

abstract syntax along with the class descriptions, constraints, and concrete syntax (notations). Collectively all the packages along with their class diagrams, class descriptions, and constraints are referred to as the UML metamodel.

The UML metamodel defines packages for structural, behavioral and auxiliary constructs, as well as profile constructs for customizing the language. The packages for structure include Classes, Composite Structures, Components, and Deployments. The packages for behavior includes Actions, Activities, Interactions, State Machines and Use Cases, as well as a Common Behavior package. The UML package structure corresponds closely with the UML major diagram types.

As previously stated, the design approach for SysML is to reuse a subset of UML and create extensions to support the specific requirements needed to satisfy the requirements in the UML for SE RFP. As shown in Figure 7-3, the SysML package structure is largely aligned with the UML package structure. Some UML packages are not being reused, since they are not considered essential for systems engineering applications to meet the requirements of the RFP. State machines, interactions, and use cases are included in SysML without modification. Some new extensions have been added to SysML packages for activities, classes, and auxiliary. The assemblies package reuses structured classes from composite structures and adds some minor extensions. New SysML packages have been added to support new constructs for Requirements, Parametrics, and Allocation.

It should be noted that the chapters in this specification align closely with the SysML package structure. The common behavior package and the profiles package are imported into SysML, but there is no corresponding chapter for these packages. The details of which UML packages are imported into SysML can be found in the package structure section of each chapter and is summarized in the compliance section.

Editorial Comment: Details regarding the SysML package structure and compliance are still being sorted out.

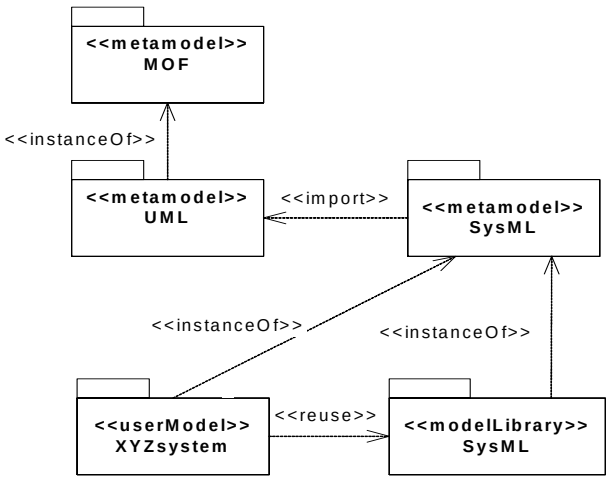


Figure 7-1. SysML Extension of UML

.

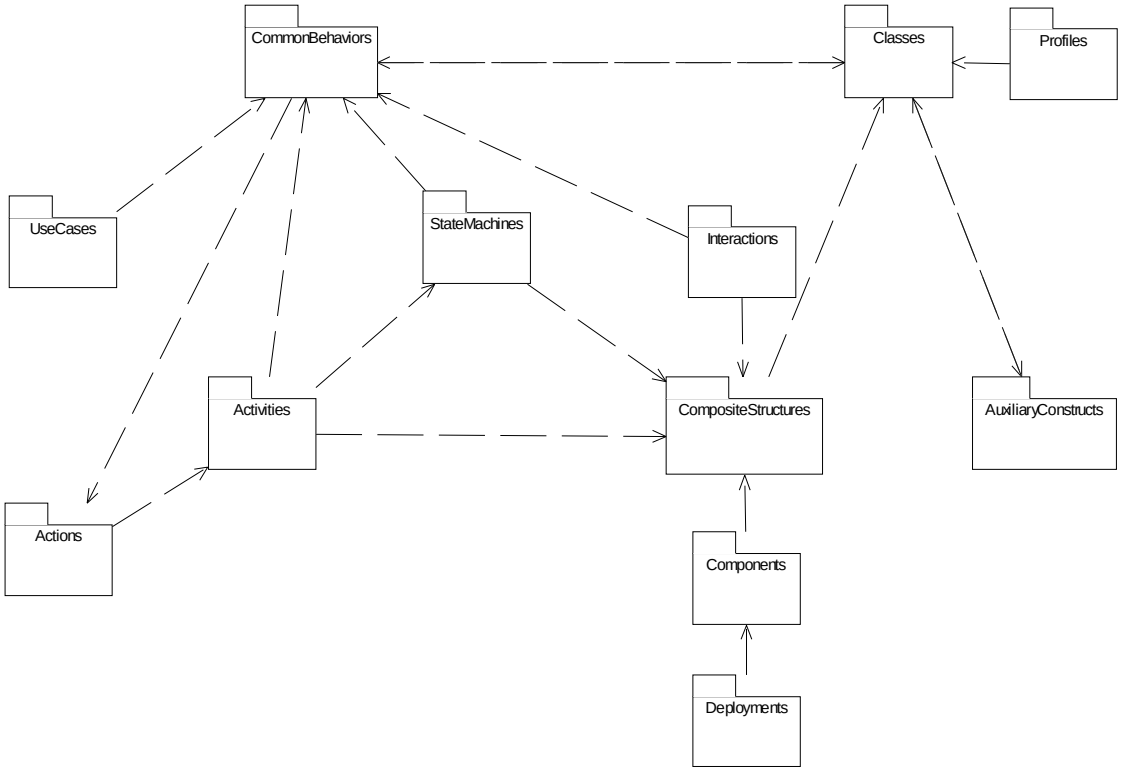


Figure 7-2. UML Superstructure Package Structure

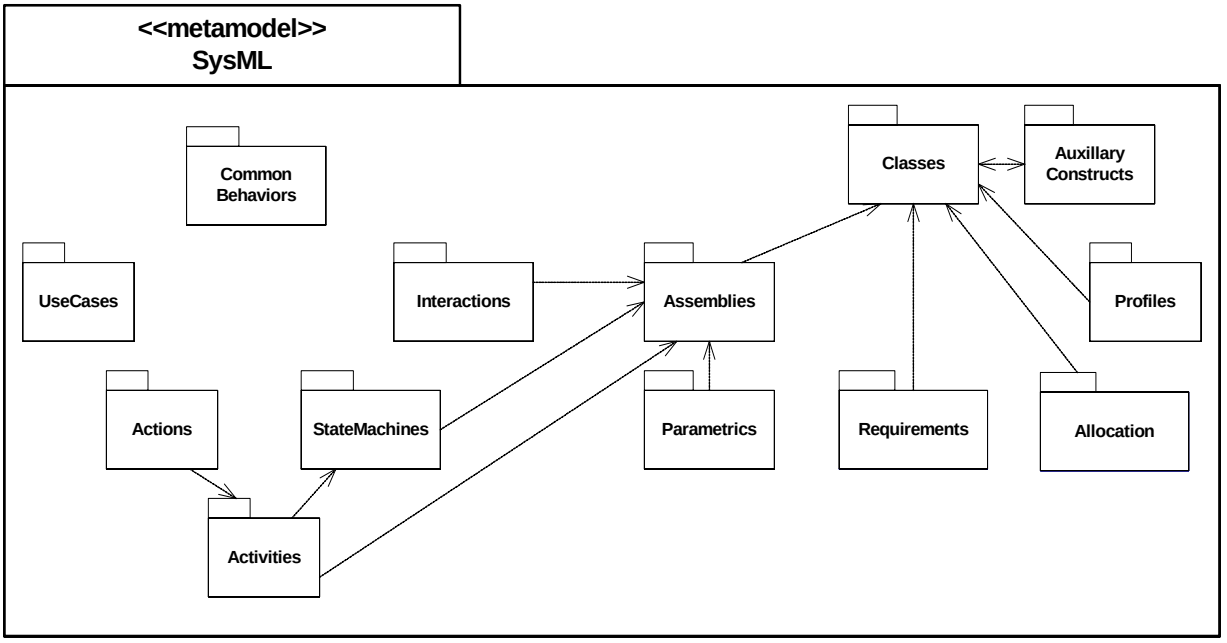


Figure 7-3. SysML Package Structure

7.3 Extension Mechanisms

Editorial Comment: Further information about how SysML extends UML, and how systems engineers can extend/ customize SysML will be provided in a future revision.

This specification uses two primary extension mechanisms to define SysML:

- UML stereotypes
- UML diagram extensions

SysML stereotypes define new modeling constructs by customizing existing UML 2.0 constructs with new properties and constraints. SysML diagram extensions define new diagram notations that supplement diagram notations reused from UML 2.0.

In addition, Chapter 19: Profiles and Appendix D: Model Libraries show examples how systems engineers can further customize SysML using stereotypes and model libraries, respectively.

7.4 4-Layer Metamodel Architecture

Like UML 2.0 on which it is based, the SysML language architecture conforms to a 4-layer metmodel architecture pattern, where the SysML metamodel reuses much of the UML metamodel, and both the SysML and the UML metamodel may be considered instances of the Meta Object Facility meta-metamodel.

7.5 AP-233 Alignment

One of the design principles is to align SysML with the ISO AP-233 standard to facilitate the interoperability between SysML tools and other engineering tools. The alignment approach is described in Appendix F.

8 Language Formalism

The SysML specification is defined by using a variation of the approach that adapts formal specification techniques. The fomal specification techniques are used to increase the precision and correctness of the specification. This chapter explains the specification techniques used to define SysML.

The following are the goals of the specifications techniques used to define SysML:

- Correctness.
- Precision.
- Conciseness.
- Consistency.
- Understandability.

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

8.1 Levels of Formalism

SysML is defined using a combination of profiling and metamodeling techniques that use precise natural language (English) to specify contraints and semantics. In general, the syntax of the language is specified precisely so that SysML will support tool interoperability via ISO AP-233 and XMI model interchange formats.

SysML's detailed semantics are described using natural language, striking a difficult balance between formal rigor and understandability. As executable SysML modeling tools become more mainstream, it is also likely that more formal techniques will be applied to improve the precsion of both SysML and UML semantics.

8.2 Chapter Specification Structure

This section provides information about how the top-level SysML packages are defined in each chapter. Each chapter has one or more of the following sections:

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.

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.

Package structure

This section specifies the package dependencies for the SysML packages that are defined in the chapter.

UML extensions

This section specifies how the SysML modeling constructs are defined using both UML stererotypes and diagram extensions.

Compliance levels

This section elaborates compliance requirements for the SysML modeling constructs as needed.

Usage examples

This section shows how the SysML modeling constructs can be applied to solve pragmatic systems engineering problems.

8.3 Constraints

SysML constraints are expressed using precise natural language (English).

8.4 Use of Natural Language

SysML uses natural language (English) for much of the specification, including the specification of constraints, and providing general descriptive text for classes, attributes, and associations.

8.5 Conventions and Typography

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

- While referring to stereoptypes, 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 mandatory section does not apply for a stereotype or metaclass, use the text: ‘No additional XXX’, where ‘XXX’ is the name of the heading. If an optional 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, structural constructs used in SysML structure diagrams, including the Class diagram, Assembly diagram, and Parametric diagram. The function and contents of these packages are specified in the following chapters, one for each of these three SysML diagram types.

9 Classes

9.1 Overview

Class diagrams define classes and relationships between them. Classes describe items of interest using features that include attributes and operations. Class relationships include associations, generalization, and dependencies. Associations have association ends that are used to specify multiplicity, navigability, ordering, and other features. Classes can be instantiated by uniquely identifying an object and creating the features and relationships of the class which describe it.

The class diagram can be used to represent many different aspects of a system. One example is to depict the conceptual elements that capture an operational concept. Another example is to represent an abstraction of the system and its components. Yet another is to specify an entity relationship diagram that describes relationships among the data in a system.

The following additions to UML are included in this chapter:

- A dependency set has been added to group dependency relationships.
- A root notation has been added to depict multiple levels of specialization.

9.2 Diagram elements

Table 3. *Graphical nodes included in class diagrams.*

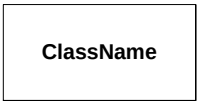
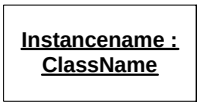
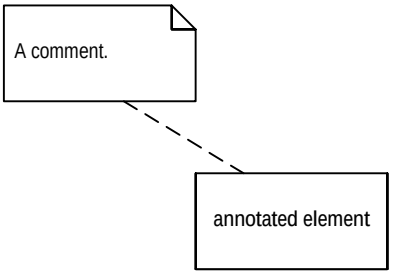
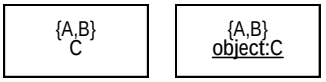
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Class		**UML::Kernel::Class**	**Basic**
InstanceSpecification		**UML::Kernel::InstanceSpecification**	**Basic**
Package		**UML::Kernel::Package**	**Basic**
Comment		**UML::Kernel::Comment**	**Basic**
Root notation		**Diagram extension**	**Advanced**

Table 4. *Graphical paths included in class diagrams.*

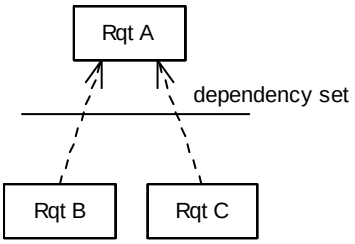
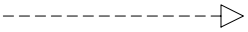
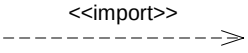
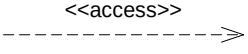
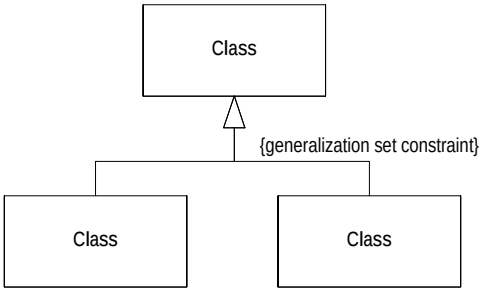
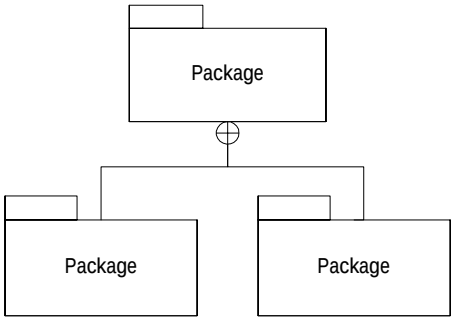
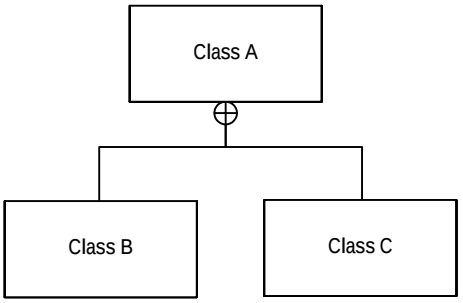
PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
DependencySet		**SysML::Classes::Dependency-Set**	**Advanced**
Association		**UML::Kernel::Association**	**Basic**
Composition		**UML::Kernel::Property with aggregation equal composite**	**Basic**
Aggregation		**UML::Kernel::Property with aggregation equal shared**	**Basic**
Dependency		**UML::Kernel::Dependency**	**Basic**
Generalization		**UML::Kernel::Generalization**	**Basic**
Realization		**UML::Kernel::Realization**	**Basic**
Public Package Import		**UML::Kernel::PackageImport**	**Basic**
Private Package Import		**UML::Kernel::PackageImport**	**Basic**

Table 4. *Graphical paths included in class diagrams.*

PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
GeneralizationSet		**UML::Kernel::Generalization-Set**	**Advanced**
Containment		**UML::Kernel::Package**	**Basic**
Containment		**UML::Kernel::Class**	**Basic**

9.3 Package structure

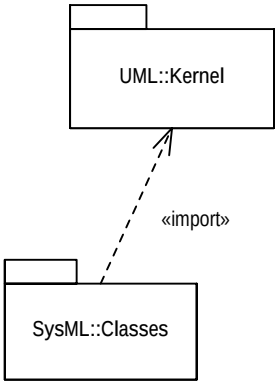


Figure 9-1. Package structure for SysML classes.

9.4 UML extensions

9.4.1 Stereotypes

Abstract Syntax

Package DependencySets

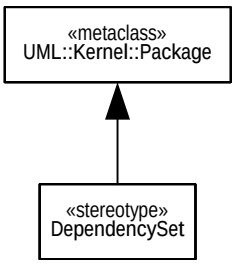


Figure 9-2. Stereotypes defined in package DependencySets.

9.4.1.1 DependencySet

Description

A grouping of dependencies that may have a common supplier or client, which can be used for many different purposes.

Constraints

[1] The elements within the stereotyped package must be of kind UML::Dependency.

9.4.2 Diagram extensions

9.4.2.1 Root notation

SysML extends the notation for classifiers and objects. It is possible to notate one or all of the more general classifiers. The comma-separated list of general classifiers is shown in curly brackets above the classifier or object name. The list is ordered and begins at the root of the generalization hierarchy. If there are more than one superclasses on the same level, they are separated by a semicolon.

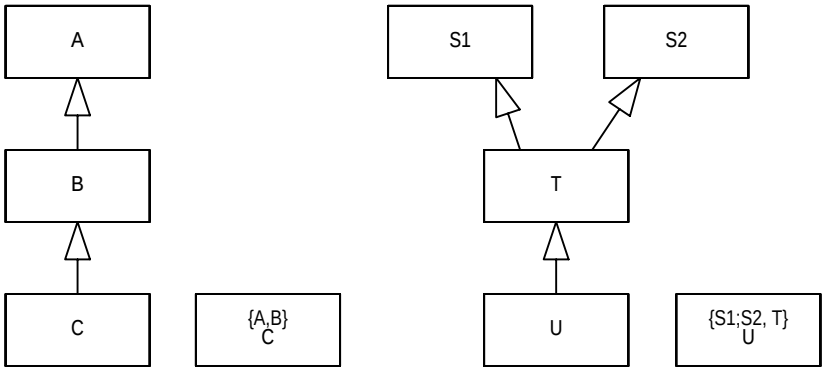


Figure 9-3. Root notation.

9.5 Compliance levels

The compliance levels are defined by the tables in section 9.2. SysML provides UML generalization sets, but exclude the use of power types themselves. SysML also excludes the package merge relationship.

9.6 Usage examples

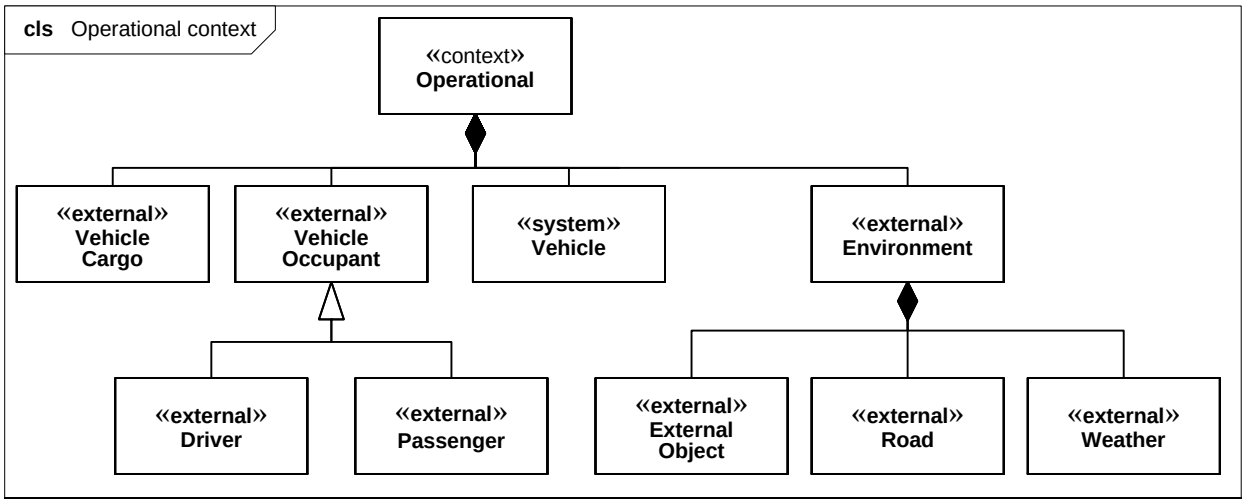


Figure 9-4. Class diagram

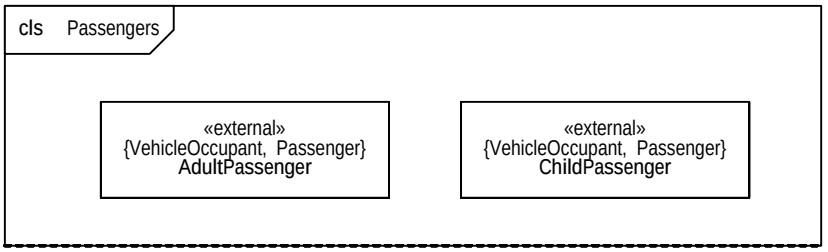


Figure 9-5. Root notation

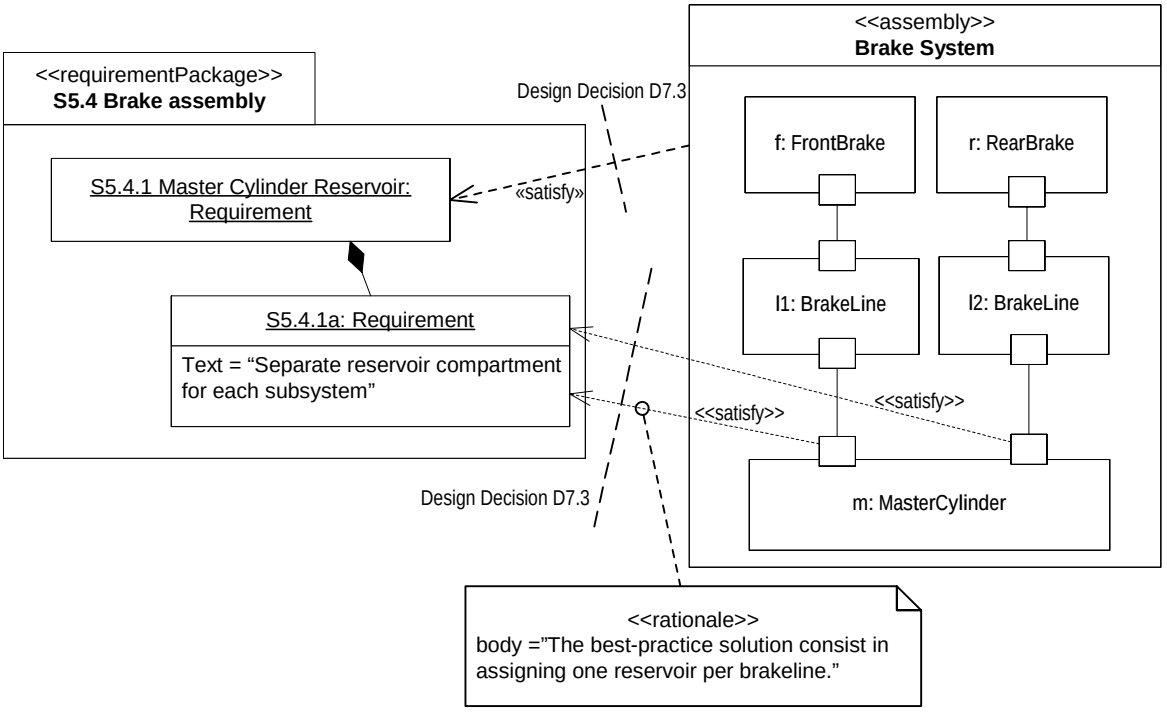


Figure 9-6. Example of a dependency set

10 Assemblies

10.1 Overview

A SysML Assembly describes a system as a collection of parts which fill specific roles within a larger whole. The ownership of parts by an assembly defines the boundary of a system. The assembly also shows the connections between parts that enable their interoperation as part of a larger whole. Some of the parts of an assembly may be shown as ports, to indicate that they can be connected externally in a larger context in which they are used. Each part can be defined by a class with its own parts, ports, and internal structure, so a uniform set of elements can be applied across multiple levels of a system hierarchy.

The SysML assembly model provides 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. The SysML assembly model facilitates reusing a library of component definitions across many different contexts and roles where they might apply, either in different systems or different roles within the same system.

The SysML assembly model 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, the specification of software, hardware, or human systems, and the use of parts that interact by many different means, such as software operations, discrete state transitions, flows of inputs and outputs, or continuous interactions.

The SysML model of assemblies and its associated diagrams are based on the UML specification for Composite Structures. UML Composite Structures already provide the essential mechanisms to define a component in terms of structural features belonging to a class. These include its internal parts, ports that can be used to connect it externally, and connections between parts and ports that enable their interaction as part of a containing whole. UML Composite Structure diagrams can be used to show either a “black box view,” in which only the externally visible elements are shown, or a “white box view,” which shows the internal details of its parts and connections. They go further than the block diagram models common to many engineering disciplines by specifying patterns of occurrences of their internal parts and connections, using structural features supported by UML classes.

To distinguish UML structured classes that adopt the SysML conventions for modeling system architecture, SysML defines a stereotype of UML classes called «assembly». This stereotype must be applied to any class to enable the assembly modeling extensions that SysML defines. It may also be used as a common root of user-defined stereotypes to classify specific kinds of systems and the kinds of elements they contain. Because a SysML «assembly» is also a class, it may be used as the type of any part, port, or other property of another assembly, thus providing a foundation for structural reuse at any level of a system design.

Because SysML assemblies may be applied to a wide variety of system types, SysML assemblies include only a subset of the modeling elements that UML defines for Composite Structures. Besides the general-purpose elements of parts, ports, and connectors, UML Composite Structures provide additional support for systems in which requests are relayed to system components responsible for performing them. SysML assemblies currently support only a domain-neutral approach to modeling of systems in which services or responsibilities may not be so clearly defined, and which can be defined using only a subset of UML facilities.

SysML encourages various forms of hybrid approaches in which UML modeling elements are used in combination with the more domain-neutral elements of SysML. For example, the required and provided interfaces of UML can be used with SysML assemblies in environments that support both. Hybrids of UML and SysML can be especially important for modeling software-intensive systems, which may need additional specialized capabilities such as code generation. Because system and software engineers often work together on the same projects and need to communicate closely as part of the same teams, it is important that they share a common set of core concepts. To support both these groups, many tools may support both SysML and UML capabilities on different parts of a larger system of systems.

UML provides several related modeling facilities for reusable structures, and which apply them further to specific modeling needs. These include: 1) collaborations, which bind roles into a surrounding context but do not otherwise commit to a specific structure to be built in a target system; 2) structured classes, which define an encapsulation of a class by means of communication across ports, and 3) components, which define modular units of a software system that may be deployed and replaced in a target system. SysML assemblies define a more basic set of modeling elements than any of these, to enable the modeling of system structures early in a development cycle before any commitment has been made as to which structures may end up being realized in what specific form.

In particular, any part, port, or property of a SysML assembly may be defined with any UML type that the native environment supports. This includes other SysML assemblies, but may also include any other form UML classifier that defines its own form of system structure. The ability to imbed one kind of system model within another can provide natural points of transition from the levels and views of a system in which software is not specifically addressed to those in which software is all-important. Additionally, SysML facilities for views and allocation are specifically designed to support multiple representations of the same system as it is refined and detailed throughout a development process. The ability to establish and maintain explicit relationships between different representations of the same system is an essential foundation for the interdisciplinary practice of systems engineering.

10.2 Diagram elements

Table 5. *Graphical nodes included in Assembly diagrams, at Basic compliance level.*

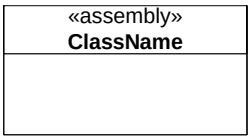
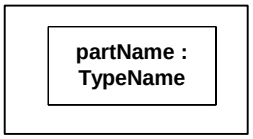
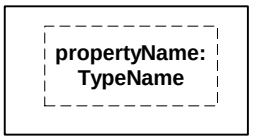
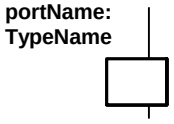
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
«assembly»		**SysML::Assemblies::Assembly**	**Basic**
Part		**UML::Property with isComposite equal True**	**Basic**
Non-composite Property		**UML::Property with isComposite equal False**	**Basic**
Port		**UML::Port**	**Basic**

Table 6. *Graphical paths included in Assembly diagrams, at Basic compliance level.*

Path Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Connector	connectorName: associationName <Association Annotations>	**UML::Connector**	**Basic**

Table 7. *Graphical nodes included in Assembly diagrams*

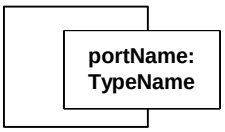
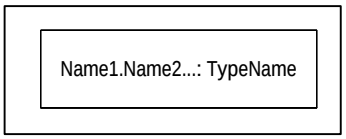
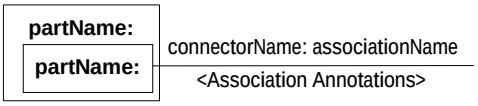
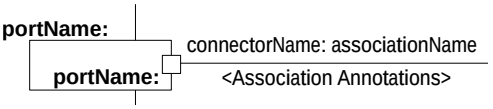
NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Port		**UML::Port**	**Advanced**
Property Path Reference		**UML::Property**	**Advanced**

Table 8. *Graphical paths included in Assembly diagrams, at Advanced compliance level.*

PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Nested Connector End		**SysML:Assemblies: NestedConnectorEnd**	**Basic**
Nested Connector End		**SysML:Assemblies: NestedConnectorEnd**	**Basic**

10.3 Package structure

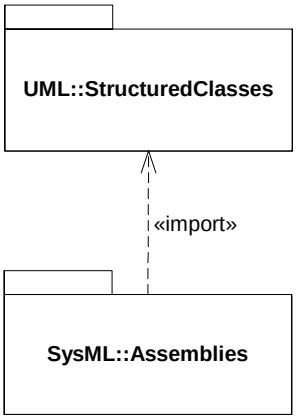


Figure 10-1. Package structure for SysML assemblies.

10.4 UML extensions

10.4.1 Stereotypes

Abstract syntax

Package Assemblies

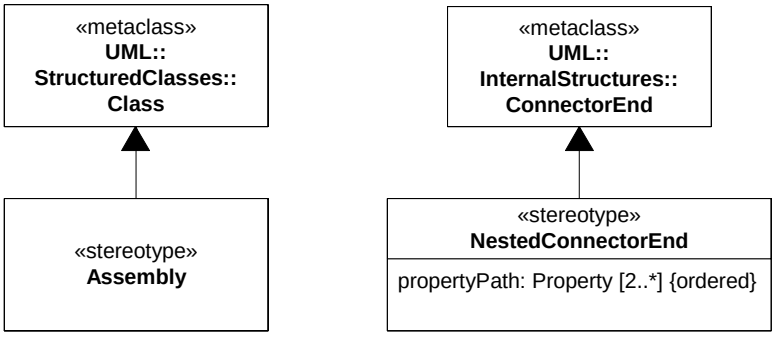


Figure 10-2. Stereotypes defined in the Assemblies package.

10.4.1.1 Assembly

Description

An «assembly» is a class that describes a structure of interconnected parts.

The SysML «assembly» stereotype indicates that a UML structured class is being utilized to express domain-neutral semantics and to enable associated modeling facilities. A SysML «assembly» defines an abstract structural syntax that may be used to model the structure of any kind of system, regardless of whether the components of the system consist of logical, physical, software, hardware, human, or other kinds of entities. This semantic assumption helps establish a simplicity and uniformity to the system structure model supported by «assembly» classes.

The elements of a SysML «assembly» class may be used to build domain-neutral views of system architecture, which avoid using additional features of structured classes defined by UML. Some UML capabilities which SysML assemblies are not required to support include the declaration of required or provided interfaces on the class or any of its ports, and specific features of UML ports such as their links to actions or internal behavior. The capabilities of UML structured classes are important for many specific kinds of systems, not only those implemented by software, but any system architecture in which clear responsibilities and communication paths are established for services and requests. While SysML does not require all the UML capabilities which apply to service-based architectures, it encourages support of these capabilities in combination with SysML, just as it encourages combination of SysML with other extended capabilities for specific target domains.

The subset of features that SysML requires for an «assembly» class does not result in any loss of inherent modeling power for systems in general. As much problem-specific detail as required can still be supplied by type declarations applied to parts, ports, properties, and connectors. The subset of features required by the «assembly» stereotype enables support of a simpler form of system structure model than a tool may support for specific domains such as software components or service-based architectures. Compared to UML classes, SysML «assembly» classes reduce the number of elements and options that a tool may choose to support, and they also simplify and standardize the use of classes to express system structure models. Their relative simplicity and uniformity can reduce learning and aid communication across stakeholders having many different backgrounds and areas of concern.

Besides the «assembly» stereotype, additional user-defined stereotypes may be defined and imported as part of a user profile to categorize different kinds of systems and the roles they fill in a particular context. Such distinctions may indicate specific stages of refinement, such as logical vs. physical, or be relative to a particular context in which a system appears, such as internal or external. Each such view can adopt or enforce its own conventions or rules for the modeling elements it imports or allows to be included within a model. The «assembly» stereotype can be used as an initial common root for such user-defined system types. See also the SysML facilities for allocation (in Chapter 16) and views (in Chapter 18) for additional ways to establish system views and relate elements between them.

Because of their definition as a stereotype of UML structured classes, any conventions of a SysML «assembly» are imposed only on classes that explictly apply it. This means that a tool may still support the full unrestricted capabilities of UML for other classes or types to which the stereotype has not been applied. These unrestricted classes may be used to type the parts or properties of any class which applies the «assembly» stereotype. In particular, UML Components, ports typed by UML Interfaces or linked to behavior, or classes without any other restrictions of a SysML «assembly» may be contained in larger system models expressed using only «assembly» classes.

10.4.1.2 NestedConnectorEnd

The NestedConnectorEnd stereotype of a UML ConnectorEnd extends a UML ConnectorEnd, in which the connected property must be owned directly by an enclosing class or part, so that the connected property may be identified by a multi-level path of accessible properties from the classifier that owns the connector.

The propertyPath list of the NestedConnectorEnd stereotype must identify a path of containing properties that identify the connected property in the context of the classifier that owns the connector. The ordering of properties is from the outermost property of the assembly that owns the connector, through the properties of each intermediate class 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 class that owns the connector.

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

10.4.2 Diagram extensions

10.4.2.1 Nested connector end

Connectors may be drawn that cross the boundaries of nested properties (including parts and ports) to connect to properties within them. The connector is owned by the most immediate class 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 violates encapsulation of the parts, ports, or other properties which a connector line crosses. The need for nested connector ends can be avoided if additional delegation ports are added to the class at each containing level. Nested connector ends are available for use in case the introduction of delegation ports is not feasible or appropriate, which must be judged according to principles guiding a particular design.

The ability to connect to nested properties within a containing class requires that multiple levels of decomposition be shown on the same diagram, either within a part or other property, or within a port on the boundary of another part. Showing the parts or ports on another port requires that the port be expanded from a small box on the part boundary to a larger box on the part boundary which shows nested ports or parts.

10.4.2.2 Property path reference

A property box may contain a name which references a 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 notation at the left in Figure 10-3 below is equivalent to the innermost nested box at the right:

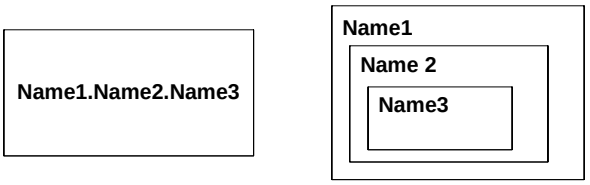


Figure 10-3. Nested property reference.

A nested property reference is shown as a dashed-outline box if any of the properties in its intervening levels would be shown with a dashed-outline box; otherwise, a solid-outline box is shown.

10.4.2.3 Property with value type always shown as part

A property that is typed by a DataType is always shown as a solid-outline box, regardless of the setting of the isComposite property. Composition is implicit for properties that have values without identity, since their values can only exist within the property. SysML assemblies may show them as parts regardless of whether the isComposite property on the Property metaclass is set.

10.5 Compliance levels

Elements have the compliance levels as indicated in the diagram elements table above.

10.6 Usage examples

10.6.1 System hierarchy

The Examples appendix (Appendix B) shows various examples of stereotypes of classes used to indicate various views of systems and levels within an overall “system of systems,” including a top-level context that includes external systems as well as a specific system of interest. All these stereotypes would typically be defined as specializations of «assembly» so that additional details of internal structure and connections could be provided. See, for example, B1.1, Concept Diagram for the “Vehicle System Operational Context”, and B1.2, Class Diagram for the “Vehicle System Operational Context,” which apply role classifications of «context», «external», and «system» at the top level of a system hierarchy. B1.9, Class Diagram for the “Vehicle System Hierarchy,” shows the use of assembly classes at progressively lower levels of the systems hierarchy. Finally, B1.10, Assembly Diagram for the “Power Subsystem,” shows the use of an assembly diagram to detail the internal structure at a particular level of the hierarchy. It also shows the use of item flows on assembly connectors, which are specified in Chapter 18, Auxiliary Constructs.

10.6.2 Engineering block diagram example

Many different engineering disciplines support visual diagramming tools in which components of a system may be selected from a library and “wired together” using lines to connect them at defined attachment points. Some forms of these diagrams can generate a complete system of equations or other representations that allow them to be simulated or analyzed as part of a larger system. This example is a simple form of a mass-spring-damper system that might be included, for example, as part of a vehicle suspension system. Figure 10-4 shows a diagram of the system as it might be represented in a typical engineering block diagram tool. Figure 10-5 shows the same system represented as a SysML Assembly diagram. The classes which type the parts in the assembly diagram would typically be defined already in a library of standard components, and imported for use in this assembly.

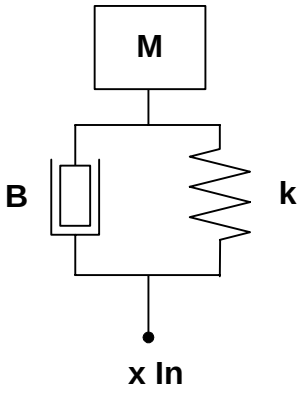


Figure 10-4. Mass-spring-damper example as engineering block diagram.

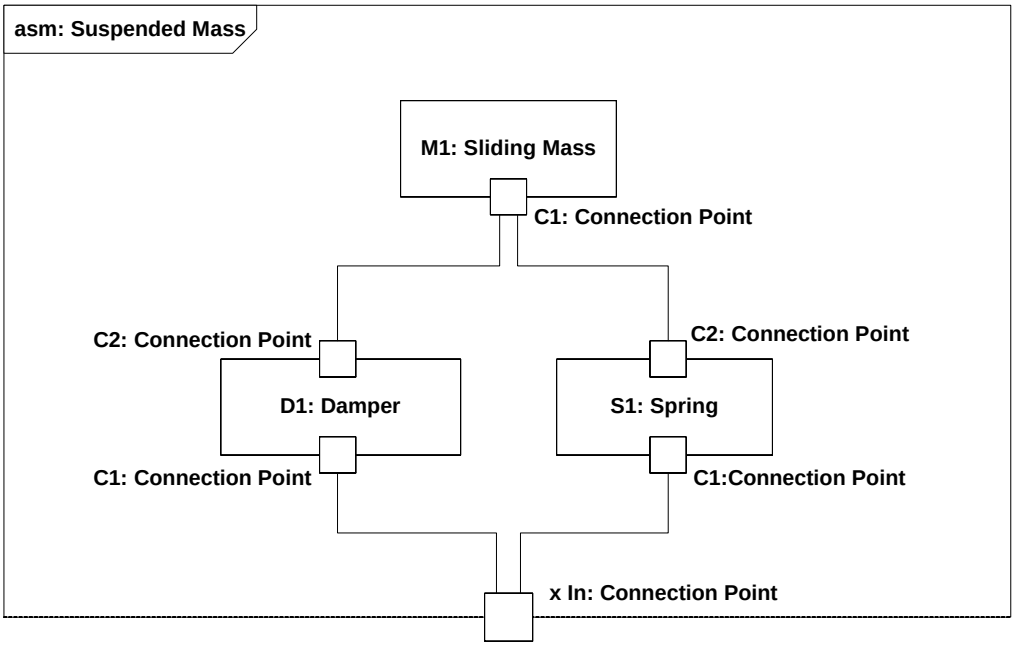


Figure 10-5. Mass-spring-damper example as SysML Assembly diagram.

10.6.3 Laptop power adapter

This example shows two levels of an assembly in which one component, a standard form of power adapter for a laptop computer made of the adapter itself plus a separate line cord, establishes an indirect connection between two parts of a larger system, the laptop computer and a power outlet. In the top-level setup, the connectors from the power adapter to both the laptop and power outlet are typed by associations, which are assumed to have been defined on a class diagram elsewhere and imported for use in this assembly..

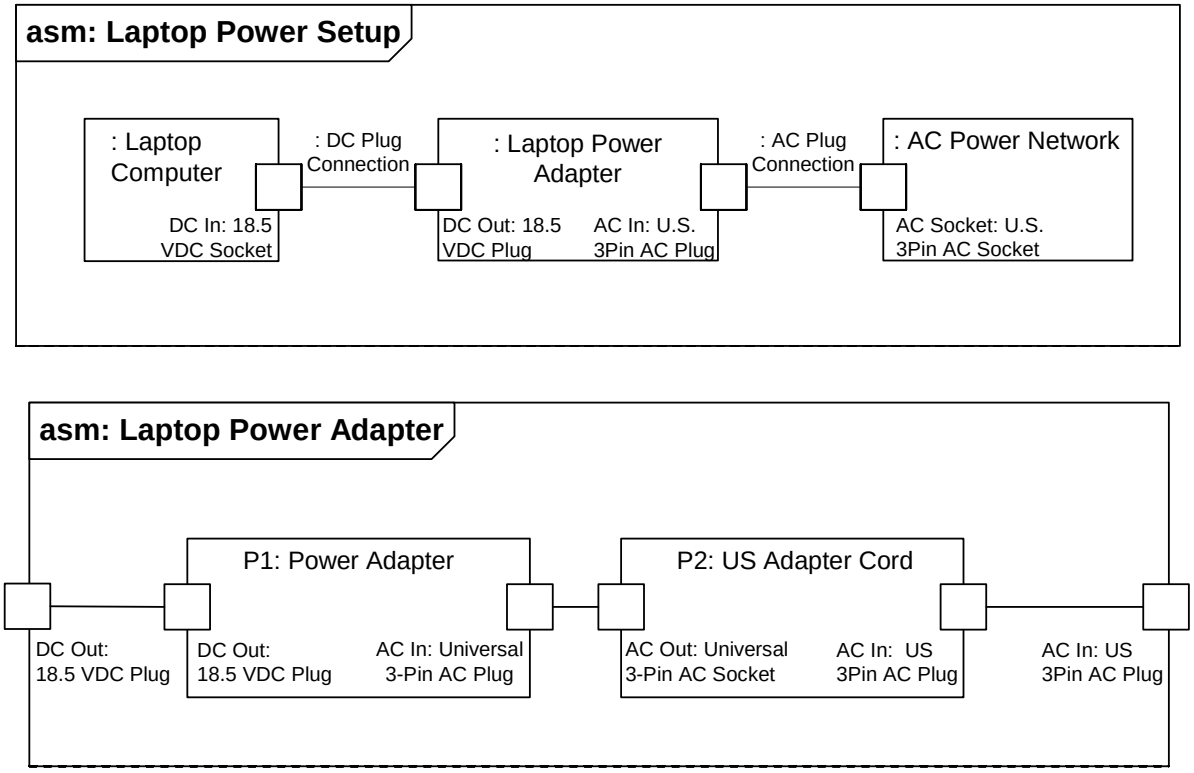


Figure 10-6. Laptop power adapter.

10.6.4 Automobile fuel system

This example shows an example usage of nested connector ends.

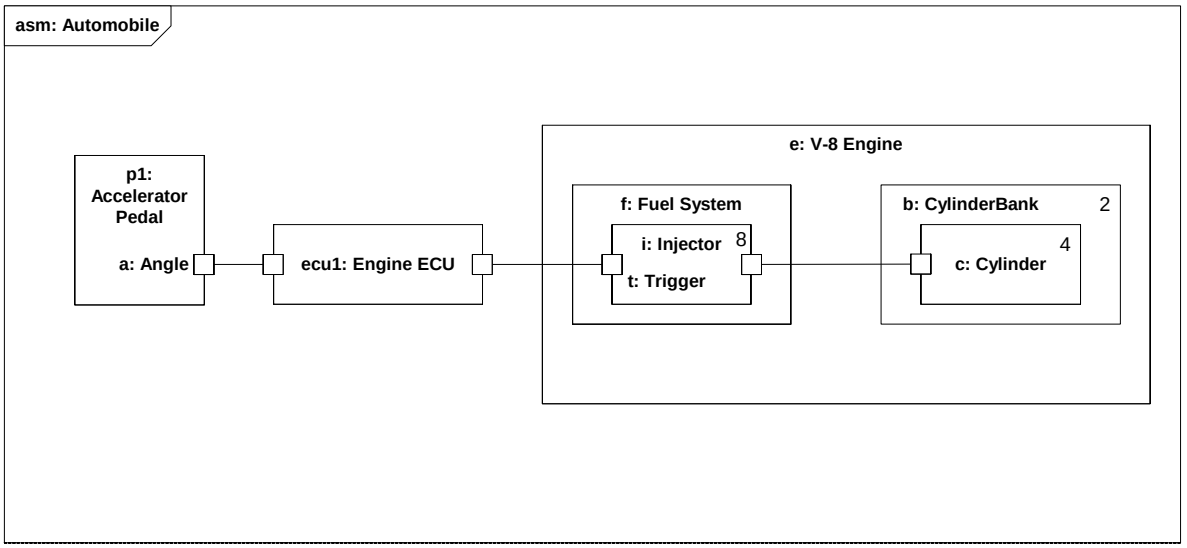


Figure 10-7. Automobile fuel system.

11 Parametrics

11.1 Overview

Parametric constraints provides mechanisms for integrating engineering analysis such as performance and reliability models with SysML assemblies. Parametric constraints depict a network of constraints among properties of a system. These constraints may be used to express mathematical expressions such as F=m*a and a = dv/dt that relate the properties of physical systems such as the aerodynamic forces on an airplane. 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.

Parametric models are analysis models that define a set of system properties and parametric relationships between them. A parametric relationship states how a change to the value of one property impacts the value of other properties. Typically, these properties express quantitative characteristics of a system, but parametric models may be used on non-quantitive properties as well. Parametric relationships are non-directional and so have no notion of causality. A parametric constraint can be used to express relationships between properties that are identified in the structural model of the system. These relationships can be built by reusing more primitive parametric relations such as basic mathematical operators.

Time can be modeled as an additional property that other properties can be dependent on. The time reference can be established by a local or global clock which produces a continuous or discrete time value, and which is defined by a property at some level of the system. Other values of time can be derived from this clock, by introducing other clocks that introduce delays and/or skew into the value of time. The Time property of a clock can typed by a Real type or Quantity class (see chapter 18, Auxiliary Constructs) and can be connected, via a property binding, to a parameter of a parametric constraint. Discrete values of time as well as calendar time can be derived from this global time property. UML offers more sophisicated descriptions of time, via the Time package in the UML 2 superstructure (ptc/03-08-02), which offers a simple view of synchronised time, and the Time Modeling subprofile of the UML Profile for Schedulability, Performance and Time (ptc/04-02-01), which offers a distributed model of time.

Parametric constraints can be dependent on the state of the object. To accomplish this, the state can be defined as a property of the object which is bound to parameters of the applicable parametric relationships.

Parametric models can be used to support tradeoff analysis. A parametric relation can be defined that represents an evaluation function to evaluate alternative solutions. The evaluation function produces one or more outputs that typically represent a general measure of effectiveness or merit. This evaluation function may include a weighting of utility functions associated with various criteria used to evaluate the alternatives. These criteria may be associated with selected system performance, cost, and physical properties. The corresponding properties from each alternative is put into the evaluation function to determine the overall measure of effectiveness. These properties may have probability distributions associated with them that are also fed into the evaluation function to compute a probabilistic or expected measure of goodness.

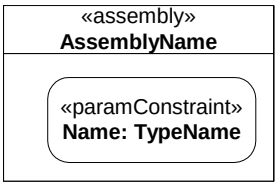
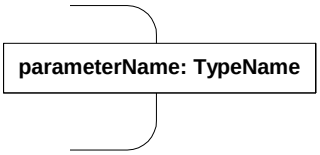
SysML identifies and names parametric constraints, but does not specify a computer interpretable language for them. SysML relies on other mathematical description languages such as MathML and associated tools to provide the execution engine for these relationships. SysML model libraries can be used to specify standard parametric relationships of general use and more customized relations for specific domains (e.g., Ohm’s law for the electrical domain).

Parametric constraints are typically used in combination with SysML assembly diagrams. A parametric constraint is defined by a stereotype of «paramConstraint» applied to a class definition. The properties of this class define the parameters of the constraint. The usage of a parametric constraint is shown by a part within an assembly, where its parameters are bound to other properties in the assembly to describe relationships between them. The usage of a parametric constraint is distinguished from other parts by a box having rounded corners rather than the square corners of an ordinary part.

The stereotype «paramConstraint» specifies that the parametric structure is used only to constrain the values of other properties in a containing context. The only valid usage of a parametric constraint is to bind the values of its parameters to other properties in a containing assembly. The semantics for any given parametric constraint (e.g., a mathematical relation between its parameter values) must be specified by whoever provides the relevant parametric constraint, either by informal specification or by a UML constraint provided within the parametric constraint definition.

11.2 Diagram elements

Table 9. *Graphical nodes included in parametric diagrams.*

Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Parametric Constraint Definition		**SysML::Parametrics:: ParamConstraint**	**Advanced**
Parametric Constraint Usage		**SysML::Property with isComposite equal True**	**Advanced**
Parametric Constraint Parameter		**UML::Port**	**Advanced**
Parametric Constraint Parameter		**UML::Port**	**Advanced**
Value Binding Constraint		**SysML::ParametricCon-straints::Binding**	**Advanced**

11.3 Package structure

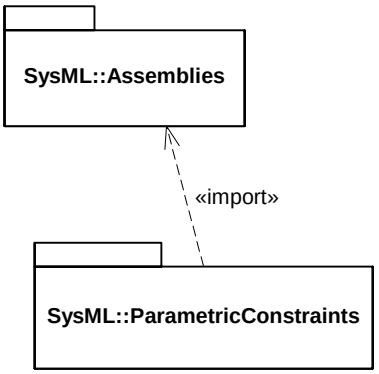


Figure 11-1. Package structure for SysML parametric constraints.

11.4 UML extensions

11.4.1 Stereotypes

Abstract syntax

Package ParametricConstraints

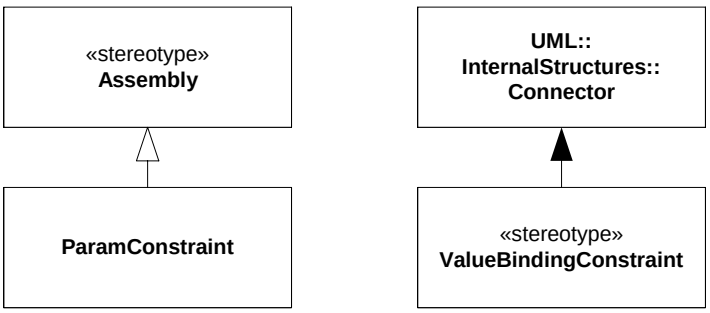


Figure 11-2. Stereotypes defined in ParametricConstraints package.

11.4.1.1 ParamConstraint

Description

A parametric constraint is an assembly used only to constrain the values of properties within a containing assembly.

The SysML «paramConstraint» stereotype declares that an assembly is defined for use as a parametric constraint. Its ports define the parameters of the parametric constraint. The only valid usage of a «paramConstraint» assembly is to bind its param-

eters to other properties of a containing assembly using a value binding constraint. A parametric constraint may contain other parametric constraints internally, to define the constraint as a composition of other constraints, but it may not contain any other parts or properties that would define any internal state of the constraint separate from the binding of its parameters. A parametric constraint does not specify any direction of causality by which the relation of its parameter values is established, but only specifies that a given relation is required to hold across its bound values. The specific relation that a parametric constraint defines may be specified either by a UML constraint on the constraint class, or by informal specification in documentation for the constraint.

11.4.1.2 ValueBindingConstraint

Description

The ValueBindingConstraint stereotype of a Connector declares that two connected properties are constrained to have equal values.

This constraint is defined as a stereotype of connector rather than a specialized form of UML constraint because only a connector has the built-in capability to connect to properties in the nested context of a containing assembly, and so that a multiplicity may also be specified on either end.

See Section 11.5.2, “Value Binding Constraint shown as a dashed line,” for the diagram notation with which a ValueBindingConstraint is shown on an assembly or parametric diagram.

Constraints

[1] The “type” association from a «binding» connector to an association that types the connector must be empty.

[2] The multiplicity of a «binding» connector must be exactly one on one end, and zero or one on the other end.

[3] The types of two properties connected by a value binding constraint must be compatible to the extent that equal values are possible.

11.5 Diagram extensions

11.5.1 Parametric diagram

Description

A special diagram type, indicated on the diagram frame by a diagram kind of “parametric” or the abbreviation “par” (see Appendix A, Diagrams), is available to show parametric constraints with special graphical conventions. Parametric constraints can also be shown on ordinary assembly diagrams, where the special graphical conventions for a parametric constraint must also be used.

On a parametric diagram, the usage of a parametric constraint is distinguished graphically from other parts of an assembly diagram by a box having round rather than square corners. The round-cornered notation is automatically applied when a part is typed by a class to which the «paramConstraint» stereotype has been applied. The keyword «paramConstraint» must precede the name of the property within the round-cornered box.

Additionally, on a parametric diagram any connector to a parametric constraint without any connector name, role names, or association name is automatically interpreted as having the «binding» stereotype of connector applied. Other than this special interpretation of unnamed connectors, a parametric diagram supports all the diagram capability of an assembly diagram.

11.5.2 Value Binding Constraint shown as a dashed line

Description

The ValueBindingConstraint stereotype of a Connector is shown on an assembly or parametric diagram as a simple dashed line with no arrowheads, names, or other notations except for the optional presence of multiplicities.

11.6 Compliance levels

Because parametrics frequently need to connect to properties nested more than one level deep in an assembly, which is an advanced feature of SysML assemblies, all elements of the parametrics chapter are assigned to the advanced compliance level.

11.7 Usage examples

11.7.1 Definition of parametric constraints on a class diagram

Parametric constraints can only be defined on a class diagram, where they must have the «paramConstraint» stereotype applied. This diagram shows two parametric constraints that are used in the subsequent firing range examples. The strings in braces below the class names are ordinary UML constraints, using a special compartment to hold the constraint, which is one of the options a tool may provide. These particular constraints are specified only in an informal language, but a more formal language such as MathML could also be used. The types of the properties shown in the attributes compartment of parametric constraint classes use Quantity subclasses defined in Chapter 18, Auxiliary Constructs.

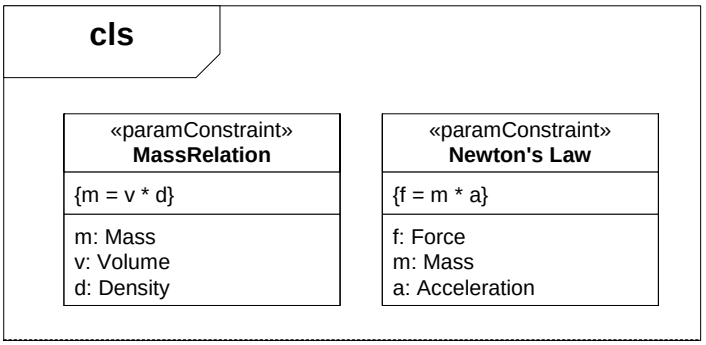


Figure 11-3. Parametric constraint definitions.

11.7.2 Usage of parametric constraints on an assembly diagram

This example shows the usage of parametric constraints on an assembly diagram. Showing parametric constraints on an assembly diagram allows them to be mixed with other part, property, and connector definitions, but requires that the parametric constraint usages and their binding connectors be shown without any special notation. As with the previous example, the types of the properties shown in the attributes compartment of the nested Cannon and Shot classes use Quantity subclasses defined in Chapter 18, Auxiliary Constructs. This example uses a separate class diagram to define the classes Cannon .

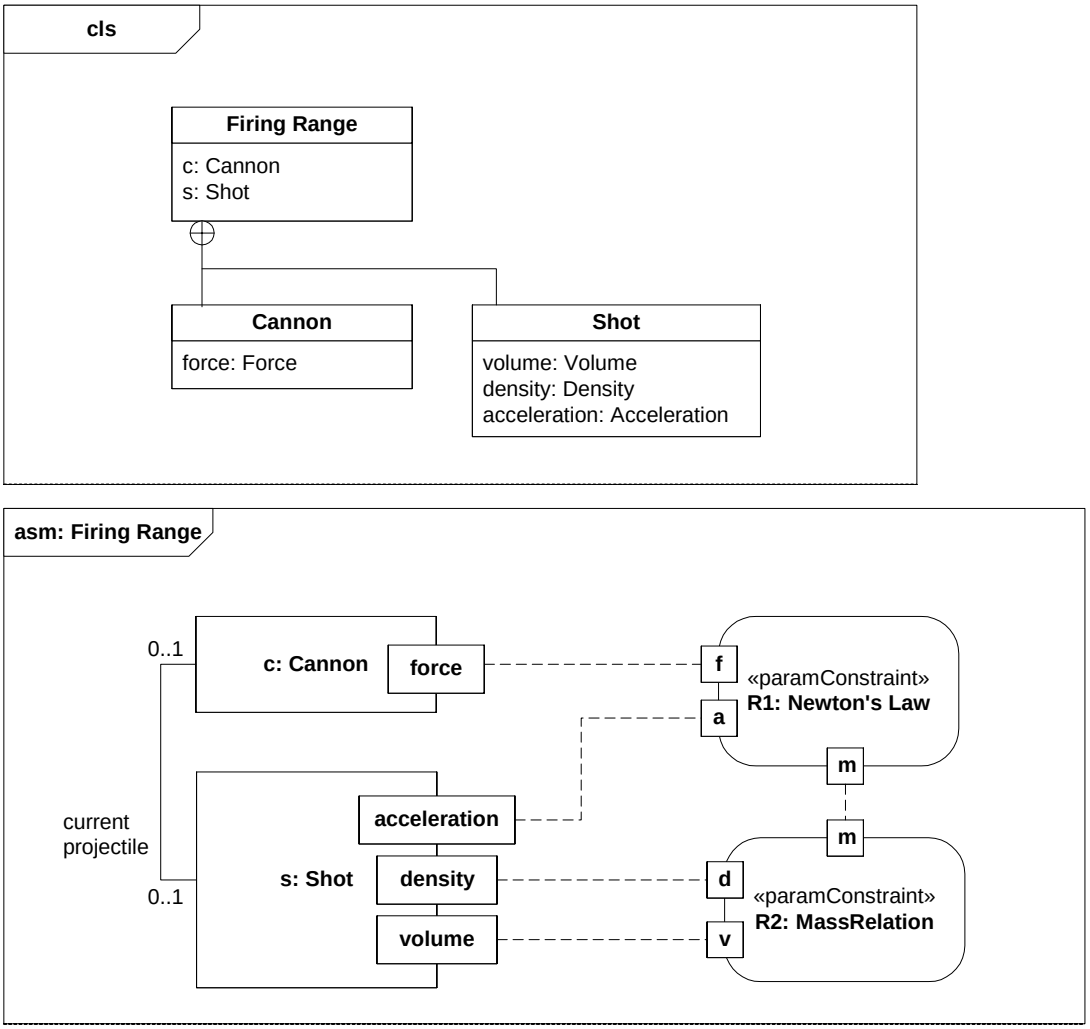


Figure 11-4. Parametric constraints on an assembly diagram.

11.7.3 Usage of parametric constraints on a parametric diagram

This example shows the use of parametric constraints on a parametric diagram. This diagram shows the use of nested property references to the properties of the parts C and S, which are assumed in this example to have already been defined as part of a Firing Range assembly. Parametric diagrams can make use of the nested property name notation to refer to multiple levels of nested property containment, not just one level deep as in this example.

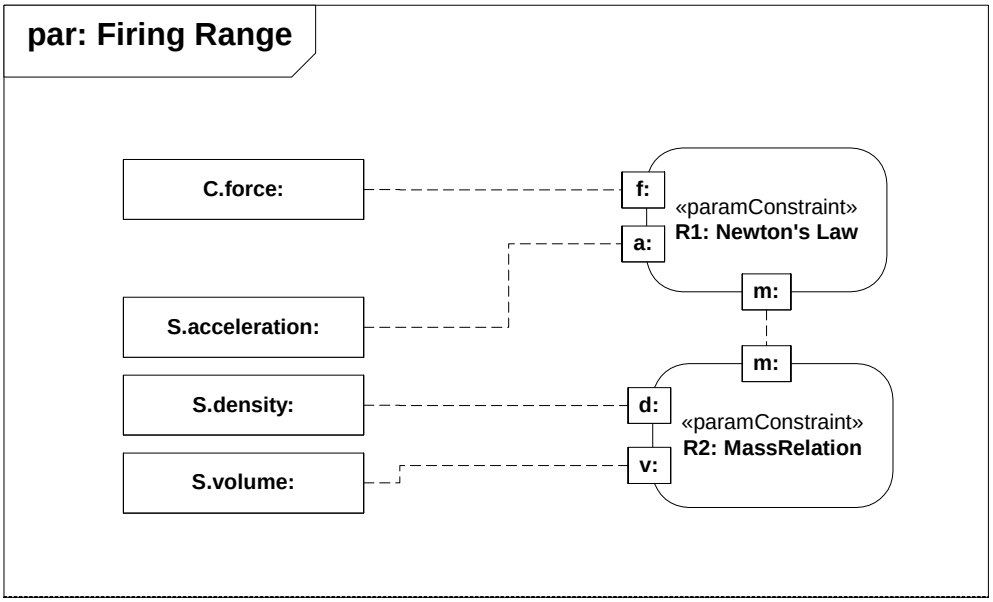


Figure 11-5. Parametric constraints on a parametric diagram.

11.7.4 System of equations

See Sample Problem B.4.12 for an example of a complete system of equations expressing performance parameters of a vehicle using a parametric diagram.

Editorial Comment: A self-contained example is still needed to show full details of a system structure with its properties, with a system of equations on this structure expressed in both textual and graphical forms.

Part III - Behavioral Constructs

This Part specifies the dynamic, behavioral constructs used in SysML behavioral diagrams, such as Activity diagrams, Sequence diagrams, and State Machine diagrams. The Activity represents the basic unit of behavior that is used in all behavioral diagrams. An activity is a behavior that is composed of actions, some of which may invoke other activities. The State Machine diagram includes activities that are invoked during transition between states, upon entry or exit from a state, or while in a state. The Sequence diagram includes activities as methods of operations that are invoked by messages. The following chapters are organized by major diagram type.

12 Activities

12.1 Overview

Activity modeling emphasizes the inputs and outputs, sequence, and conditions for coordinating other behaviors. It provides a flexible link to classifiers owning those behaviors. Activity diagrams are similar to Enhanced Functional Flow Block Diagrams (EFFBD), although the terminology and notation are different (Appendix C). The following is a summary of the SysML extensions to UML 2 activity diagrams.

Control as data

SysML extends control in activity diagrams to give the functionality of data:

1. In UML 2 Activities, control can only enable actions to start. SysML extends this with control for disabling actions that are already executing, by providing a model library with a type for control values (see "ControlValue" in Figure 12-8).
2. The above is also used to support control processed by behaviors called control operators, rather than determining whether the behavior starts or not (see "ControlOperator" in Figure 12-2). Control operators can represent a complex logical operator that transform its inputs to produce an output that controls other actions.

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:

1. SysML supports 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 12-2). 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.
2. SysML extends 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 12-2). It also extends object nodes with the option to discard values if they do not immediately flow downstream (see "NoBuffer" in Figure 12-2). 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 12-2):

1. SysML extends edges with expressions evaluating to probabilities for the likelihood that a value leaving the decision node or object node will traverse an edge.
2. SysML extends output parameter sets with probabilities for the likelihood that values will be output on a parameter set.

EFFBD extensions

EFFBD's are supported as optional extensions and translations to Activity Diagrams. The extension includes:

1. Constraints on usage of activity diagrams (section 12.4.4.1).
2. Additional notation (section 12.4.4.2).

Activities as classes

In UML 2, all behaviors are classes, including activities, and their instances are executions. Behaviors can appear on class diagrams, and participate in generalization and associations. SysML extends the class diagram notation for activities, and clarifies the semantics of composition association between activities, and between activities and classes that type object nodes in the activities, and defines consistency rules between these class diagrams and activity diagrams. See section 12.4.2.

12.2 Diagram elements

This section covers the concrete syntax added by SysML and inherited from UML. It identifies the compliance level for each element. Compliance may vary for subelements of each one, see section 12.5 for details.

12.2.1 Diagram elements

SysML activity concrete syntax is the same as the UML 2 activity and action concrete syntax, with keywords and properties per the SysML abstract syntax. No new graphical nodes or paths are defined. Table 10, Table 11, and Table 12 list the SysML extensions to UML 2 notation. The left column is the class or property, with classes in uppercase, properties in lower case. The second column from the right column references the class or class of the property. The corresponding table for UML 2 activities is in the Diagram section of Chapter 12 of the the UML 2 Superstructure specification, http://www.omg.org/cgi-bin/doc?ptc/03-08-02.

Table 10. *Graphical nodes included in activity diagrams.*

Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
ControlOperator	«controlOperator» Function	**SysML::«controlOperator»**	**Advanced**
NullTransformation	NullTransformation	**SysML::NullTransformation (in model library)**	**Advanced**
ResourceConstraint	«resourceConstraint»	**SysML::«resourceConstraint»**	**Advanced**
Action		**UML::Action**	**Basic**

Table 10. *Graphical nodes included in activity diagrams.*

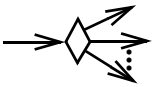
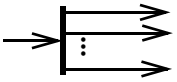
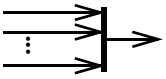
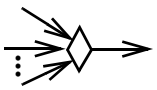
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
ActivityFinal		**UML::ActivityFinalNode**	**Basic**
ActivityNode	**See ExecutableNode, ControlNode, and ObjectNode.**	**UML::ActivityNode**	**Basic**
ControlNode	**See DecisionNode, FinalNode, ForkNode, InitialNode, JoinNode, and MergeNode.**	**UML::ControlNode**	**Basic**
DecisionNode		**UML::DecisionNode**	**Basic**
FinalNode	**See ActivityFinal and FlowFinal.**	**UML::FinalNode**	**Basic**
FlowFinal		**UML::FlowFinalNode**	**Advanced**
ForkNode		**UML::ForkNode**	**Basic**
InitialNode		**UML::InitialNode**	**Basic**
JoinNode		**UML::JoinNode**	**Basic**
MergeNode		**UML::MergeNode**	**Basic**

Table 10. *Graphical nodes included in activity diagrams.*

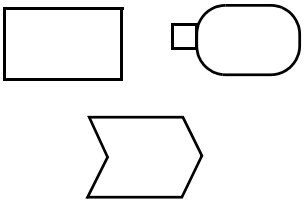
NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
ObjectNode		**UML::OjectNode and its children.**	**Basic**

Table 11. *Graphical paths included in activity diagrams.*

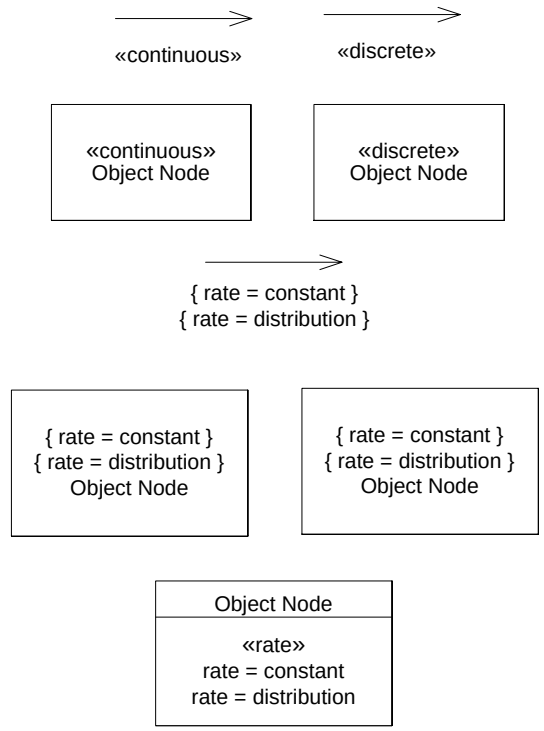
PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Rate		**SysML::«rate», SysML::«continuous», SysML::«discrete»**	**Advanced**
Rate	{ rate = constant } { rate = distribution }	**SysML::«rate»**	**Advanced**

Table 11. *Graphical paths included in activity diagrams.*

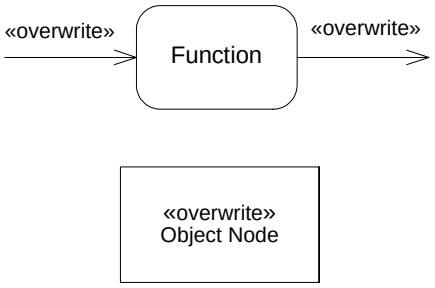
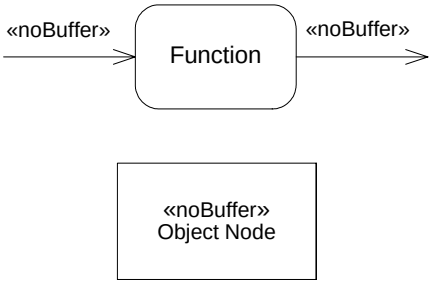
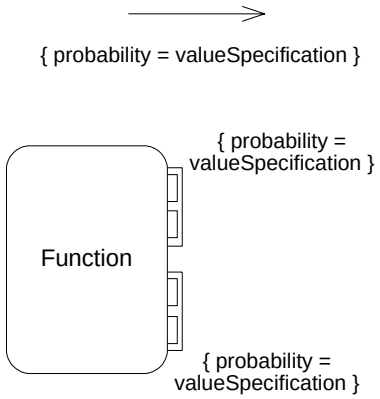
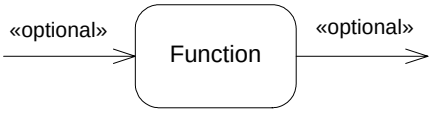
PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
OverWrite		**SysML::«overwrite»**	**Advanced**
NoBuffer		**SysML::«noBuffer»**	**Advanced**
Probability		**SysML::«probability»**	**Advanced**
Optional		**SysML::«optional»**	**Basic**

Table 11. *Graphical paths included in activity diagrams.*

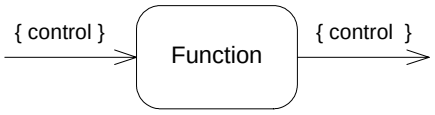
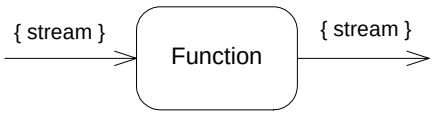
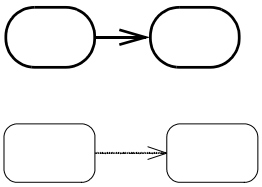
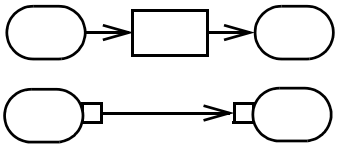
Path Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
isControl		**UML::Pin.isControl**	**Advanced**
isStream		**UML::Parameter.isStream**	**Advanced**
ActivityEdge	**See ControlFlow and ObjectFlow.**	**UML::ActivityEdge**	**Basic**
ControlFlow		**UML::ControlFlow** **SysML::ControlFlow**	**Basic**
ObjectFlow		**UML::ObjectFlow and its children.**	**Basic**

Table 12. *Other graphical elements included in activity diagrams.*

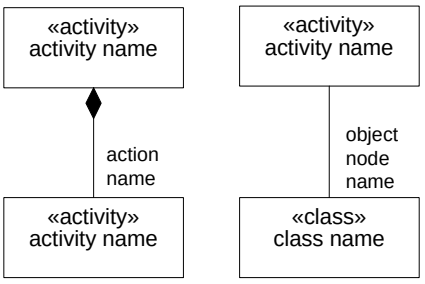
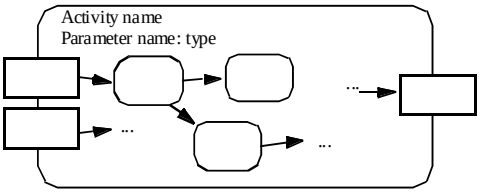
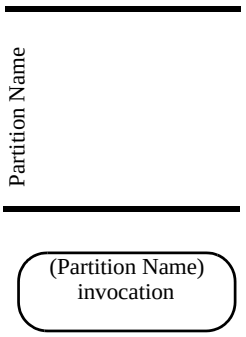
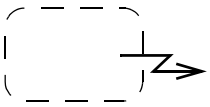
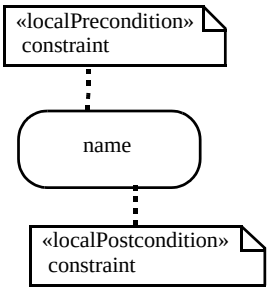
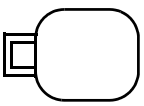
Element Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Activity, ObejctNode, Association on Class Diagram		**SysML::Activity,** **SysML:ObjectNode**	**Advanced**

Table 12. *Other graphical elements included in activity diagrams.*

ELEMENT NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Activity		**UML::Activity**	**Basic**
ActivityPartition		**UML::ActivityPartition**	**Basic**
InterruptibleActivity-Region		**UML:InterruptibleActivityRe-gion**	**Advanced**
Local pre- and post-conditions.		**UML:Action**	**Advanced**
ParameterSet		**UML::ParameterSet**	**Advanced**

12.3 Package structure

SysML Activities depends on UML Activities, as shown in Figure 12-1. It also depends on the Quantities model library, used here at the metalevel, and a metamodel version of the Distributions stereotypes. The Activities model library has elements that are instances of UML BasicActivities metaclasses.

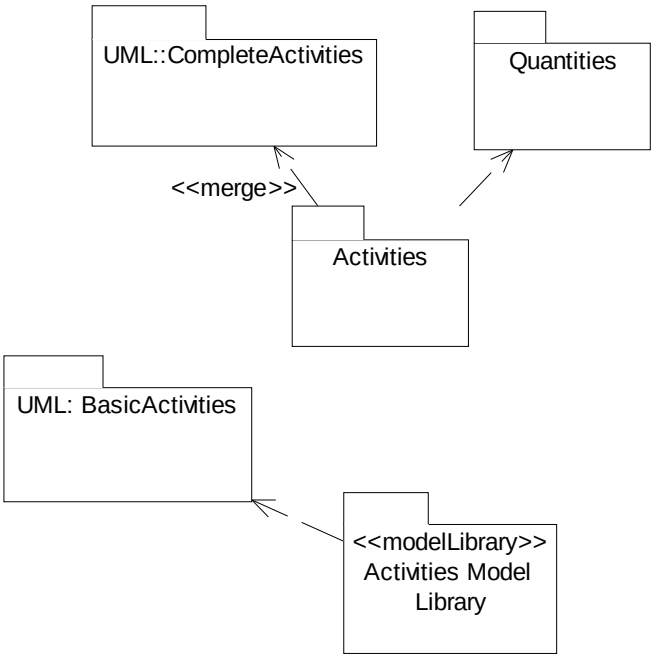


Figure 12-1. Package structure for SysML activities.

12.4 UML extensions

12.4.1 Stereotypes

See constraints on application of stereotypes in the entry for each stereotype, in particular, the absence of a stereotype can sometimes indicate a constraint.

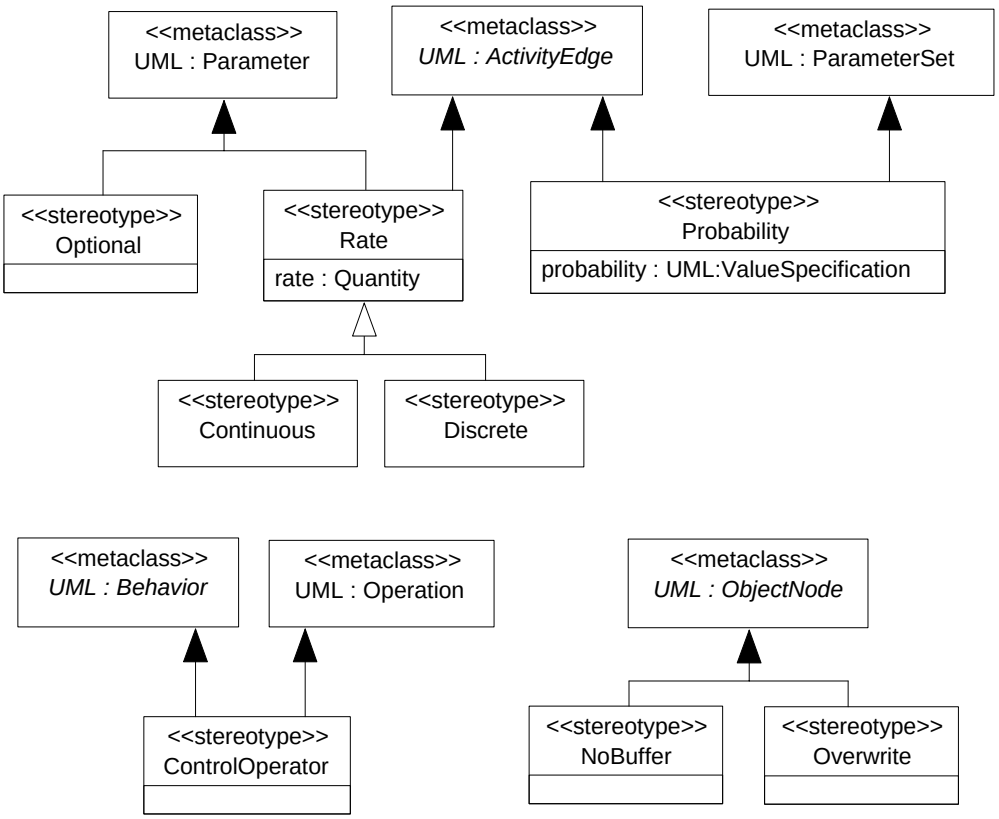


Figure 12-2. Stereotypes for behavior, behavior elements, and activity elements.

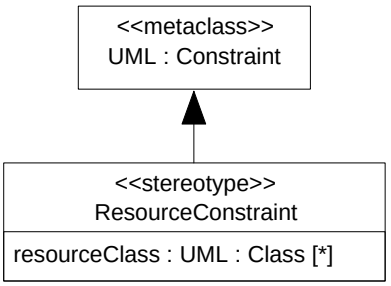


Figure 12-3. Stereotypes for resources.

12.4.1.1 Continuous

This is a kind of Rate stereotype representing a rate of flow for items treated as infintesimals for the purpose of the application, for example, water flowing a pipe, or entities the application considers small enough to treat as continuously flowing, such as ball bearings in a factory that produces them. It is independent from UML streaming, see “Rate”. A streaming parameter may result in continuous flows or not, and a continuous flow may involve streaming parameters or not.

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 kinds 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.

Constraints

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

[2] The «nobuffer» stereotype always applies to object nodes that have an incoming edge stereotyped by «continuous».

12.4.1.2 ControlOperator

Description

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” in section 12.4.3.1). The control values 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 is 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. 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.

Constraints

[1] When the «controlOperator» stereotype is applied, the behavior or operation must have at least one parameter typed by ControlValue, otherwise the behavior or operation must have no parameter typed by ControlValue.

[2] A method behavior must have the «controlOperator» stereotype applied if its operation does.

12.4.1.3 Discrete

This is a kind of Rate stereotype representing a rate of flow for items treated as individuals for the purpose of the application, for example, cars in a car factory.

Constraints

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

12.4.1.4 NoBuffer

Description

When the «nobuffer» stereotype is applied to object nodes, tokens arriving at the node that are refused by outgoing edges, or refused by actions for object nodes that are input pins, are discarded. This is typically used with fast or continuously flowing

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. 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.

12.4.1.5 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 nondeterministic. 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. When the stereotype is not applied, the semantics is as in UML, specifically, tokens arriving an 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..

12.4.1.6 Optional

Description

When the «optional» stereotype is applied to parameters, the lower multiplicity must be equal to zero. Otherwise, the lower multiplicity must be greater than zero, which is called “required”.

Constraints

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

12.4.1.7 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 output parameter set, all of the output parameters of the behavior owning the parameter set must be in some parameter set.

12.4.1.8 Rate

Description

When the «rate» stereotype is applied to an activity edge, it specifies the rate over time that objects and values traverse the edge. When the stereotype is applied to a parameter, the parameter must be streaming, and the stereotype gives the rate over time that objects or values are expected to flow in or out of the parameter. 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 12.4.3, and section 12.4.1.3. The «rate» stereotype has a rate property of type Quantity, which includes distributed quantities, see Chapter 18. The quantity must have units and dimensions appropriate to rates of flow.

Constraints

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

[2] The denominator for units used in the rate property must be time units.

12.4.1.9 ResourceConstraint

Description

When the «resourceConstraint» stereotype is applied to a constraint, the constraint can specify the types of resources used by the model element it constrains.

12.4.2 Diagram extensions

Notation for the stereotypes in section 12.4.1 follow the UML standard conventions for stereotypes, as extended by SysML, and shown in section 12.2. The entries below give notational extensions for some UML elements.

12.4.2.1 Activity

Notation

In UML 2, all behaviors are classes, including activities, and their instances are executions. The semantics for composition associations between activities is the same as the UML semantics for composite associations and activity termination: terminating the activity, that is, destroying an instance of that activity, causes the termination of the executions that it started through CallBehaviorAction, that is, destroying the composed instances also. The upper multiplicity on the part end restricts the number of concurrent synchronous executions of the behavior that can be invoked by the containing activity. See Constraints, below.

Activities in class diagrams appear as regular classes, optionally using the «activity» keyword for clarity, as shown in Figure 12-4. See example in section 12.6. They may also use the same notation as CallBehaviorAction, except the rake notation can be omitted, if desired. Also see use of activities in class diagrams at ObjectNode.

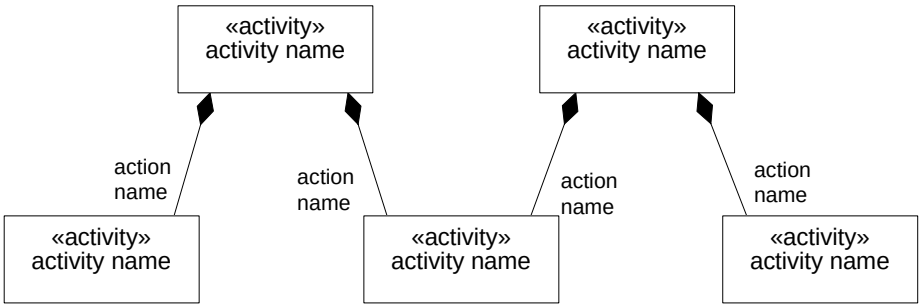


Figure 12-4. Activities as classes.

CallBehaviorActions in activity diagrams can optionally show the action name with the name of the invoked behavior using the colon notation shown in Figure Figure 12-5.

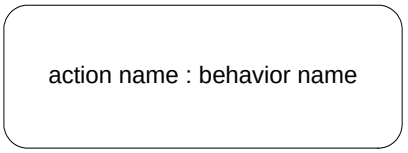


Figure 12-5. CallBehaviorAction notation.

Constraints

The following constraints apply when composite associations in class diagrams are defined between activities:

[1] The part end name must be the same as the name of a synchronous CallBehaviorAction in the composite 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.

12.4.2.2 ControlFlow

Presentation Option

Control flow may be notated with a dashed line and stick arrowhead.

12.4.2.3 ObjectNode

Notation

Classes that type object nodes may appear in class diagrams associated with the activity containing the object node, as shown in Figure 12-6. The associations may be composite if the intention is to delete instances of the class flowing the activity when the activity is terminated. See example in Section 12.6 .

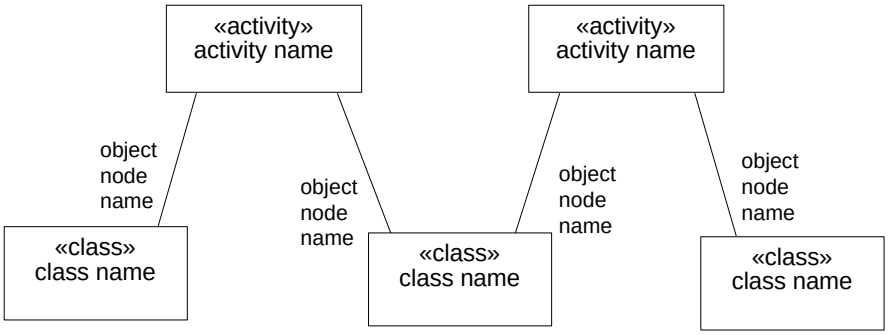


Figure 12-6. Activities as classes associated with types of object nodes.

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 12-7.

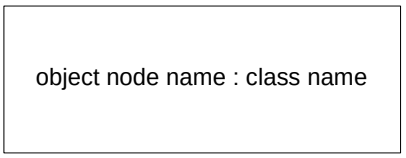


Figure 12-7. ObjectNode notation.

Constraints

The following constraints apply when associations in class diagrams are defined between activities and classes 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 class 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.

12.4.3 Model library

Editorial Comment: Guidelines for model library definitions are still being established.

The SysML model library for activities is shown below.

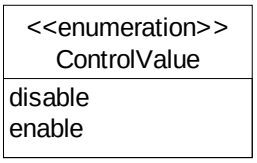


Figure 12-8. Control values.

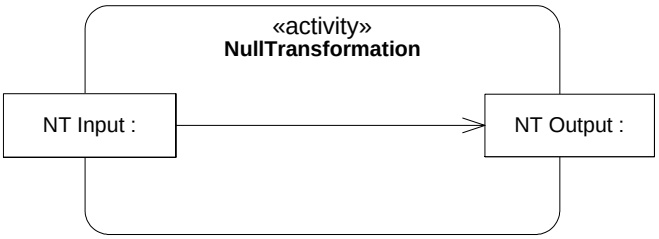


Figure 12-9. Null transformation.

12.4.3.1 ControlValue

Description

The ControlValue enumeration is a type available for modelers to apply when control is to be treated as data (see section 12.4.1.2) and for UML control pins. It can be used for 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] UML::ObjectNode::isControlType is true for object nodes with type ControlValue.

12.4.3.2 NullTransformation

Description

NullTransformation is an activity that takes any type of instance as input and returns it, doing nothing else. Parameter type is omitted so that any type can be accepted.

12.4.4 EFFBD extensions

The following optional extensions and translations are provided for applications requiring adherence to Enhanced Functional Flow Block Diagram conventions and semantics (see “EFFBD extensions” in section 12.1 for overview).

12.4.4.1 Constraints

EFFBD's place the following constraints on the use of activity diagrams:

[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.

[3] (On Action) All actions require exactly one control edge coming into them, and exactly one control edge coming out.

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

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

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

[7] The following SysML stereotypes cannot be applied: Rate, ControlOperator, NoBuffer, Overwrite.

12.4.4.2 Notation

The EFFBD extension provides the following notational additions to activity diagrams to align with EFFBD. The notation does not affect the metamodel and is interchanged with UML 2 Diagram Interchange. See Appendix C.

1. Object flow arrows have double arrowheads when they target pins corresponding to parameters with minimum multiplicity greater than zero.
2. Decision and merge nodes used in cycles where the number of iterations is determined once before the first iteration are labelled with the letters "IT" inside the node.
3. Decision and merge nodes used in cycles where the terminination of the iteration is determined by a decision node or parameter sets in the cycle are labelled with the letters "LP" inside the node.
4. Edges coming out of forks can be annotated with the label "kill".

Other EFFBD notation than the above is not part of the EFFBD extension. Most EFFBD notation is different in activity diagrams, but most of the translation is one-to-one and follows the translation of of terms.

12.5 Compliance levels

This is additional detail on the compliance levels described in section 12.2.

The following are basic:

- The EFFBD subprofile (section 12.4.4).

The following are advanced:

- Actions that are not basic or not required.
- joinSpec on JoinNode
- upperbound and state on ObjectNode
- pre/postcondition on Activity
- CentralBufferNode
- isDimension and isExternal on ActivityPartition
- ValuePin

The following are not required:

- StructuredNode's that are actions.
- weight on ActivityEdge
- transformation and multicase/receive on ObjectFlow
- isSingleExecution on Activity
- isException on Parameter

12.6 Usage examples

The following examples illustrate modeling continuous systems (see "Continuous systems" in section 12.1). Figure 12-10 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, which are the responsibility of the Driver and Brake System respectively, as shown by partitions. These behaviors execute until the key is turned off, using streaming parameters to communicate with other functions. 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 detemrined 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 12-11 and 12-12. An alternative notation for this function decomposition is shown in Figure 12-13.

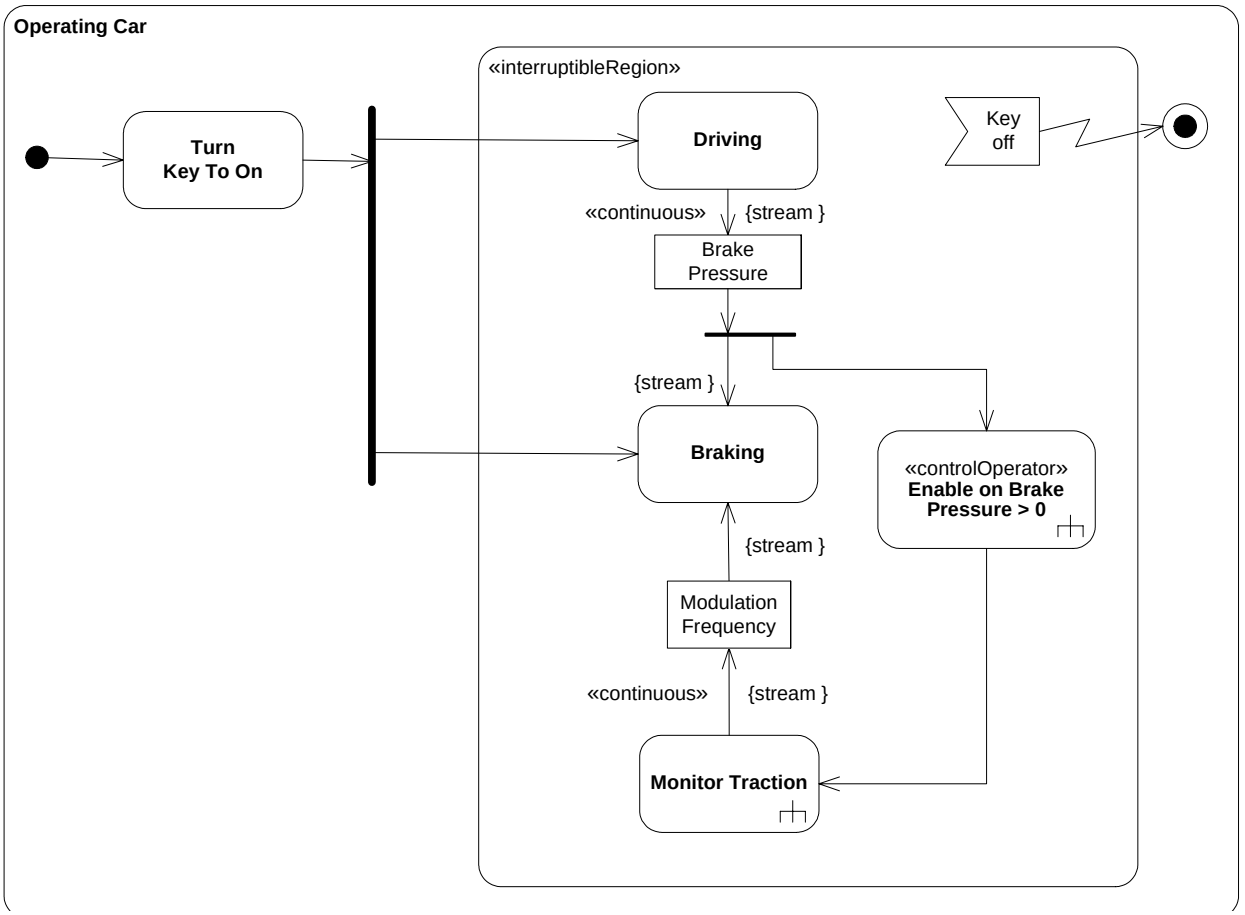


Figure 12-10. Continuous system example 1.

The activity diagram for Monitor Traction is shown in Figure 12-11. When Monitor Traction is enabled, it begins listening for signals coming in from the wheel and accelerometer. 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.

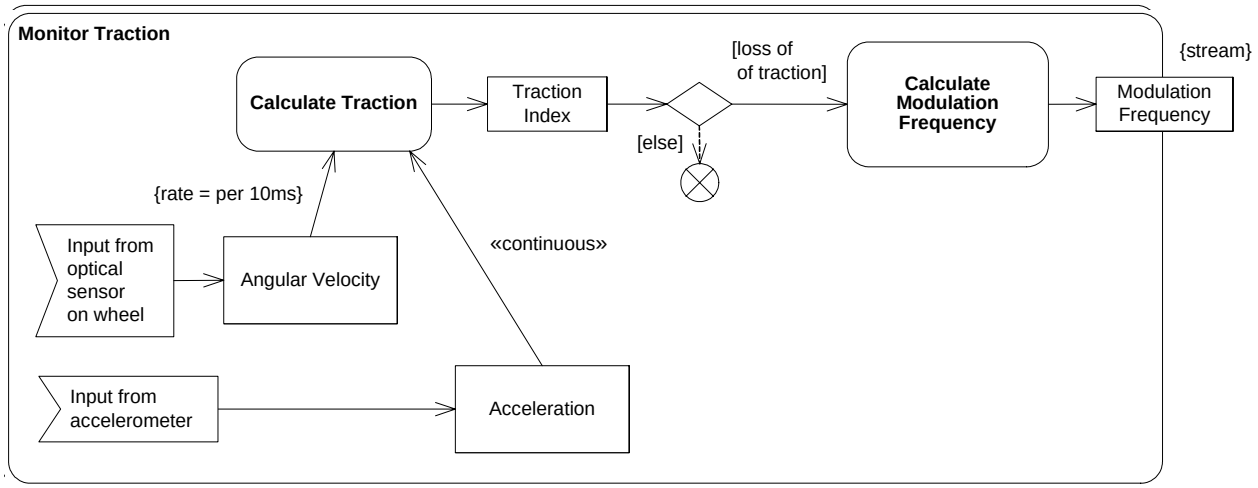


Figure 12-11. Continuous system example 2.

The activity diagram for the control operator Enable on Brake Pressure > 0 is shown in Figure 12-12. 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.

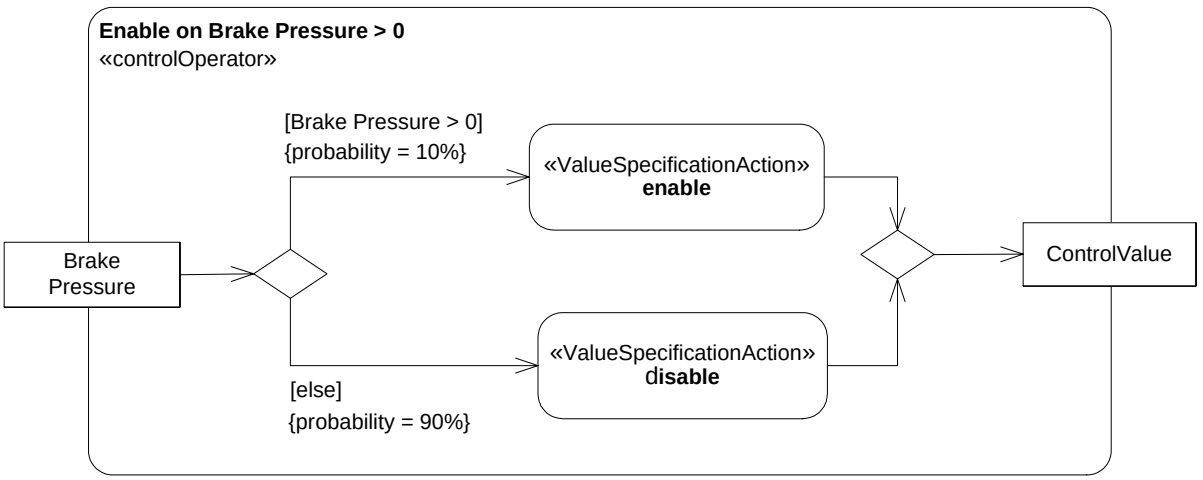


Figure 12-12. Continuous system example 3.

Figure 12-13 shows a class diagram with composition associations between the activities in Figures 12-10, 12-11, and 12-12, as an alternative way to show the functional decomposition of Figures 12-10, 12-11, and 12-12. Each instance of Operating Car is an executon of that behavior. It owns the executions of the behaviors it invokes synchronously, such as Driving. Like

all strong composition, if an instance of Operating Car is destroyed, terminating the execution, the executions it owns are also terminated.

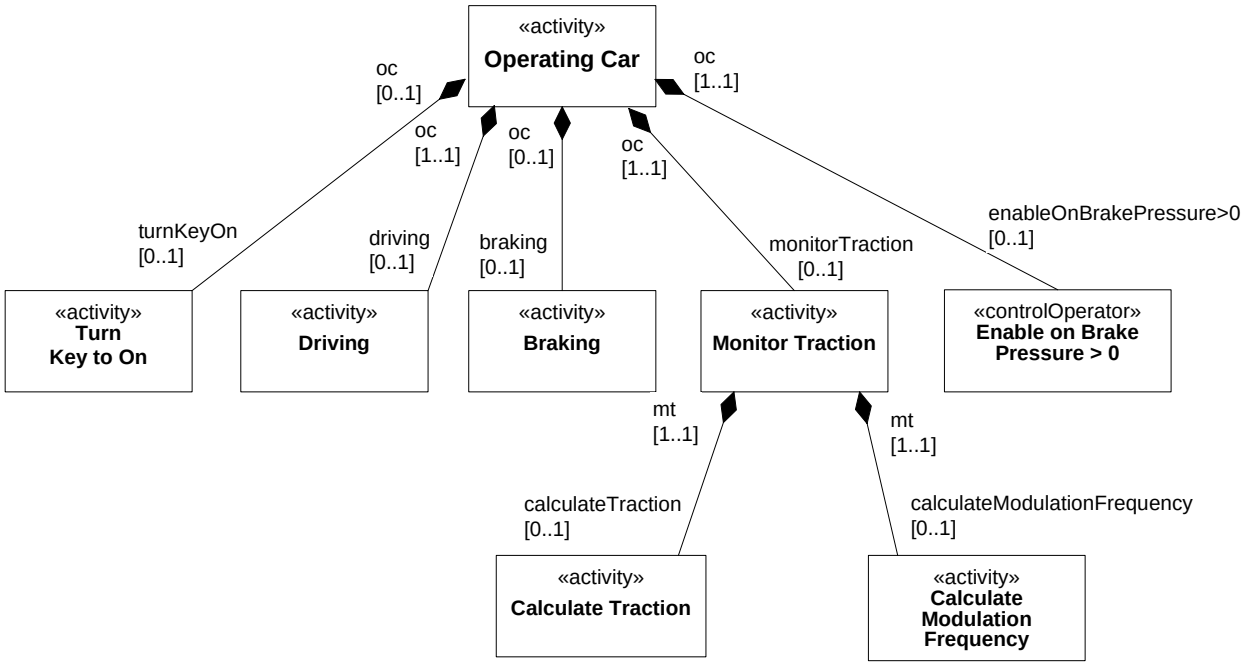


Figure 12-13. Example class diagram for functional decomposition

Figure 12-14 shows a class diagram with composition associations between the activity in Figure 12-10 and the classes that type 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.

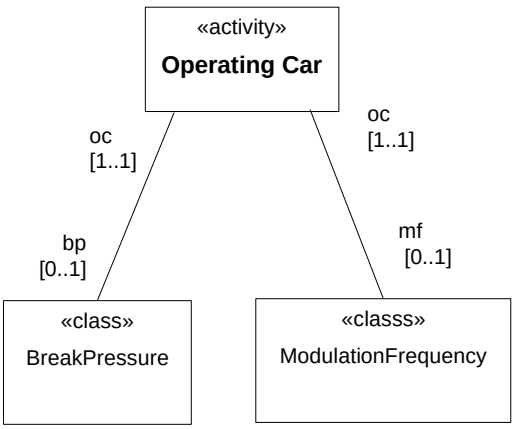


Figure 12-14. Example class diagram for object node types

13 Interactions

13.1 Overview

The SysML interaction diagrams include the sequence diagram and timing diagram, and are unchanged from UML 2, except for some minor notational enhancements to timing diagrams. The UML 2 communication diagrams and interaction overview diagram are not part of SysML. The following is a brief overview of each diagram type.

Sequence diagram

The sequence diagram specifies a series of interactions in terms of control flow. The control flow is defined by sending and receiving messages between lifelines. A message combines control and dataflow. It initiates behavior in the object receiving the message and passes inputs to the behavior. The time ordering of the messages is associated with the vertical placement of the message on the diagram. Complex sequences are abstracted into a reference sequence diagram. Conditional logic can be included to represent alternative, sequential flows, and loops. Gates provide interaction points with external lifelines. Lifelines can be decomposed into their constituent parts.

Interaction overview diagram

The interaction overview diagram represents a sequence of interactions, including logic to describe alternative or concurrent interactions. It is not part of SysML at this time. However, an interaction overview diagram can be approximated by a constrained use of an activity diagram where an action invokes an interaction versus an activity.

Timing diagram

The Timing diagram represents the change over time associated with changes in states, activities, or property values. The Timing diagram is a 2-dimensional graph that represents time on one axis, and state, activity, and or property along the other axis. The timing diagram can also represent the time ordering of events and/or the time that an expression evaluates to true. The graph depicts the event or change in the state, activity, property value (or expression value) over time. The property values can be either discrete or continuous.

13.2 Diagram elements

The graphic nodes that can be included in structural diagrams are shown in Table 14.

Table 14 - Graphic nodes included in sequence diagrams

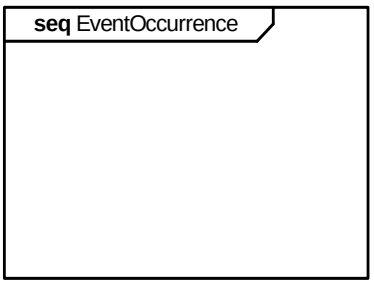
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 419.	**Basic**

Table 14 - Graphic nodes included in sequence diagrams

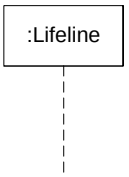
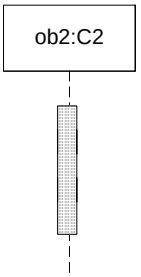
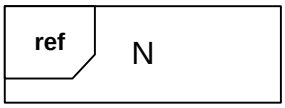
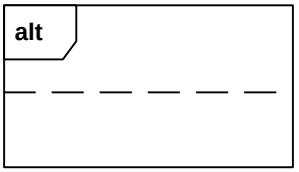
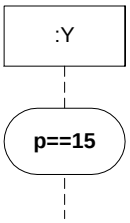
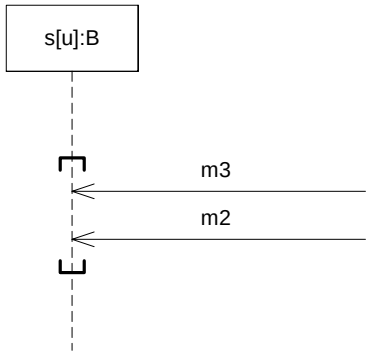
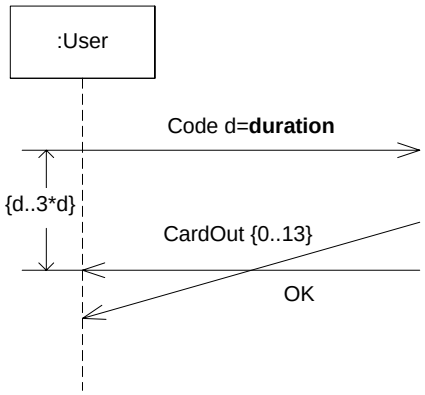
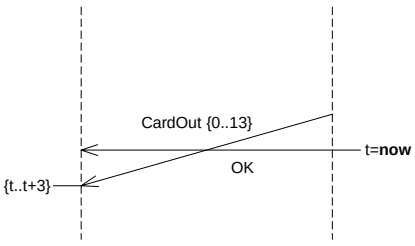
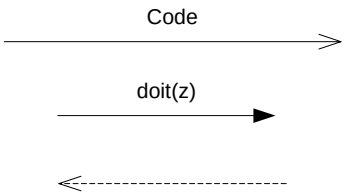
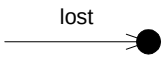
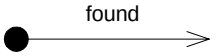
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Lifeline		See “Lifeline (from BasicInteractions, Fragments)” on page 427	**Basic**
ExecutionOccurrence		See “CombinedFragment (from Fragments)” on page 409. See also “Lifeline (from BasicInteractions, Fragments)” on page 427 and “ExecutionOccurrence (from BasicInteractions)” on page 417	**Basic**
InteractionOccurrence		See “InteractionOccurrence (from Fragments)” on page 423.	**Basic**
CombinedFragment		See “CombinedFragment (from Fragments)” on page 409	**Basic**
StateInvariant / Continuations		See “Continuation (from Fragments)” on page 414 and “StateInvariant (from BasicInteractions)” on page 433	**Advanced**

Table 14 - Graphic nodes included in sequence diagrams

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Coregion		See explanation under *parallel* in "CombinedFragment (from Fragments)" on page 409	**Advanced**
Stop		See Figure 333	**Basic**
Duration Constraint **Duration Observation**		See Figure 347	**Basic**
Time Constraint **Time Observation**		See Figure 347	**Basic**

The graphic paths between the graphic nodes are given in Table 15.

Table 15 - Graphic paths included in sequence diagrams

Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all *complete* messages. See “Message (from BasicInteractions)” on page 428.	**Basic**
Lost Message		Lost messages are messages with known sender, but the reception of the message does not happen. See “Message (from BasicInteractions)” on page 428	**Advanced**
Found Message		Found messages are messages with known receiver, but the sending of the message is not described within the specification. See “Message (from BasicInteractions)” on page 428	**Advanced**
GeneralOrdering		See “GeneralOrdering (from BasicInteractions)” on page 418	**Advanced**

13.3 Package structure

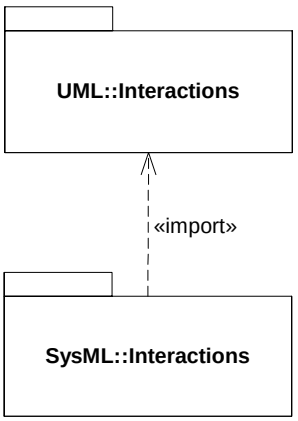


Figure 13-1. Package structure for SysML interactions.

13.4 UML extensions

No UML extensions in this section.

13.5 Compliance levels

13.6 Usage examples

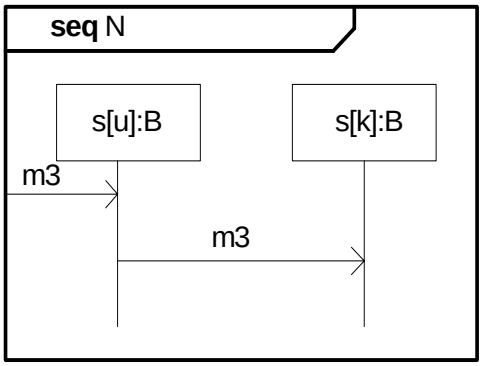


Figure 13-2. Simple Interaction Diagram

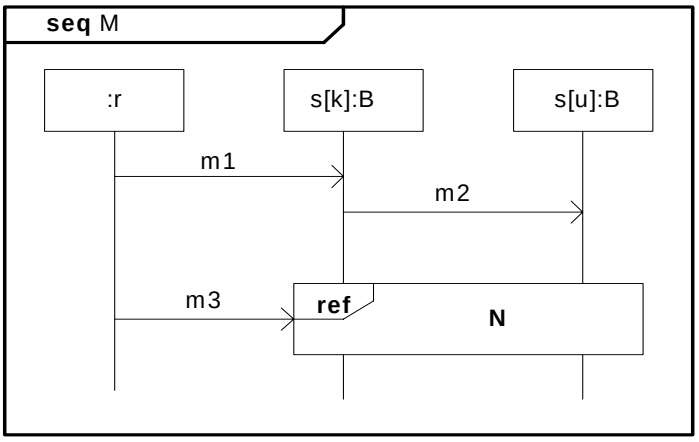


Figure 13-3. Interaction Diagram using an interactionOccurance

Sample interaction overview diagram for “Drive Vehicle” depicts the the top level flow of control. This diagram is a restricted use of an activity diagram where the actions can invoke an interaction occurence as well as an activity.

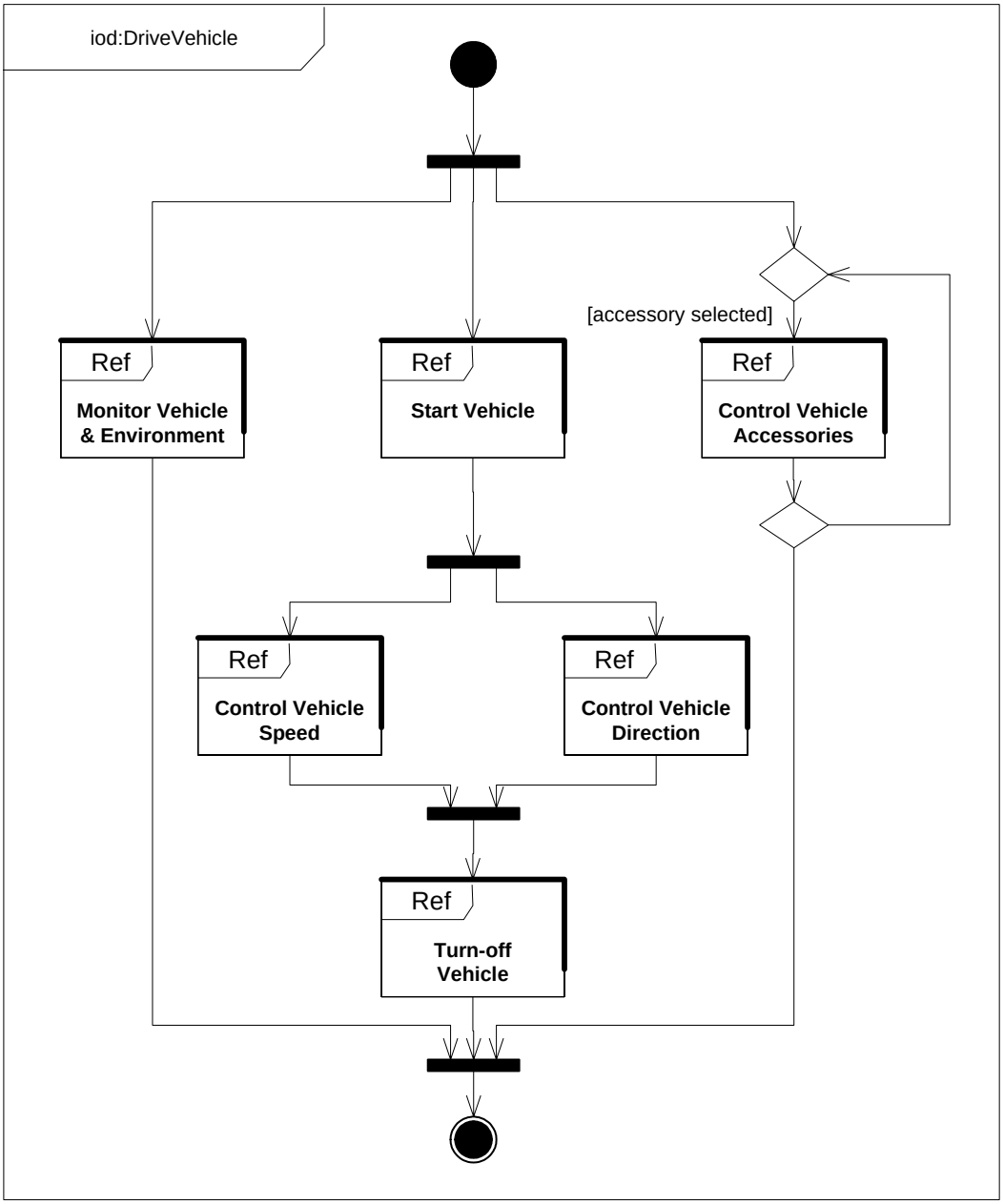


Figure 13-4. Interaction Overview Diagram for “Drive Vehicle”

Sample interaction overview diagram for “Start Vehicle” depicts the behavior which was referenced in the higher level interaction overview diagram in Figure 13-4. This diagram references more detailed behaviors.

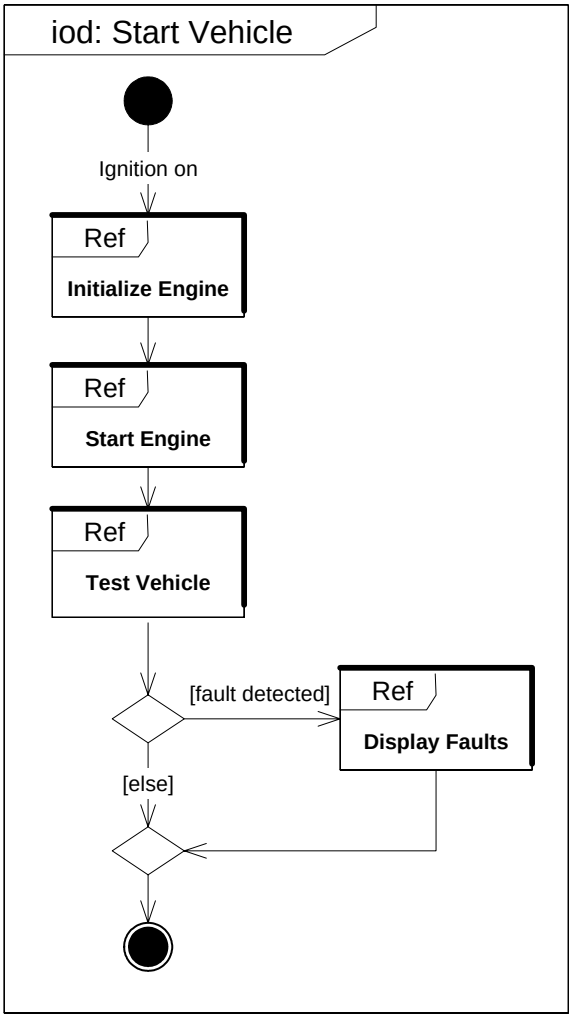


Figure 13-5. Interaction Overview Diagram for “Start Vehicle”

Sample timing diagram for the “Vehicle Performance Timeline” depicts the vehicle speed as a *function of time.*

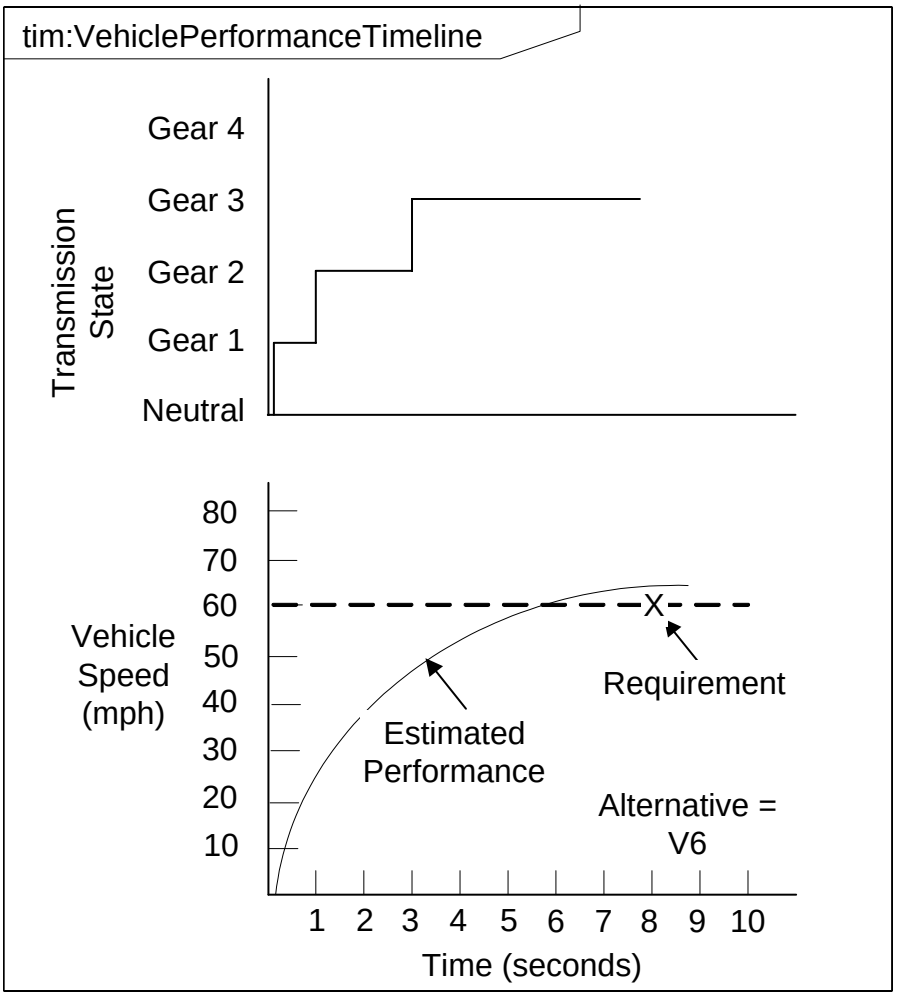


Figure 13-6. Timing Diagram for the “Vehicle Performance Timeline”

14 State Machines

14.1 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.

In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system. The two kinds of UML state machines are referred to as *behavioral state machines* and *protocol state machines*. Protocol state machines are not included as part of SysML.

14.2 Diagram elements

The concrete syntax for state machine diagrams is included in the following Tables and is unchanged from UML 2. This table outlines the graphic elements that may be shown in state machine diagrams, and provides cross references where detailed information about the semantics and concrete notation for each element can be found.

The graphic nodes that can be included in state machine diagrams are shown in Table 16.

Table 16: Graphic nodes included in state machine diagrams.

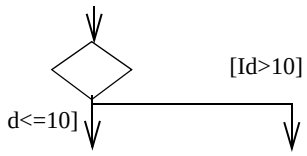
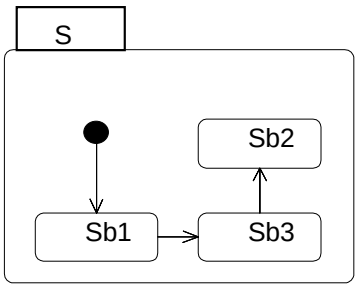
NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERANCE*	*COMPLIANCE*
Choice pseudo state		UML::ChoicePoint pseudostate.	Basic
Composite state		UML::State	Basic
Entry point		UML::EntryPoint pseudostate	Basic

Table 16: Graphic nodes included in state machine diagrams.

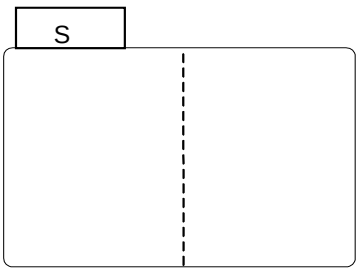
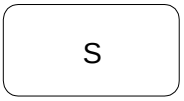
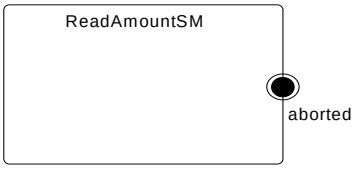
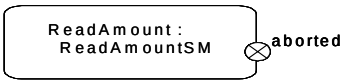
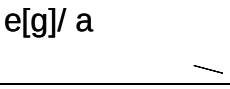
NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERANCE*	*COMPLIANCE*
Exit point		UML::ExitPoint pseudostate	Basic
Final state		UML::FinalState pseudostate	Basic
History, Shallow pseudo state		UML::ShallowHistory pseudostate	Basic
Initial pseudo state		UML::Initial pseudostate	Basic
Junction pseudo state		UML::Junction pseudostate	Basic
Region		UML::Region pseudostate	Basic
Simple state		UML::State	Basic

Table 16: Graphic nodes included in state machine diagrams.

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERANCE*	*COMPLIANCE*
State Machine		UML::Statemachine	Basic
Terminate node		UML::TerminateNode	Basic
Submachine state		UML::State	Basic

The graphic paths that can be included in state machine diagrams are shown in Table 17.

Table 17: Graphic paths included in state machine diagrams.

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERANCE*	*COMPLIANCE*
Transition		Transition	Basic

14.3 Package structure

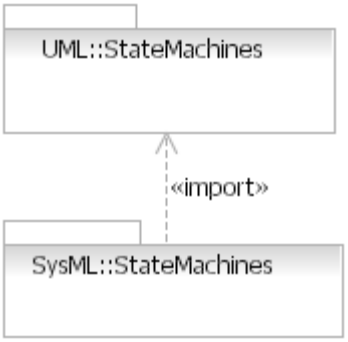


Figure 14-1. Package structure for SysML state machines.

14.4 UML extensions

No UML extensions in this section

14.5 Compliance levels

The compliance level for the Statechart section is assigned to the basic compliance level.

14.6 Usage examples

The following are examples of state machine diagrams.

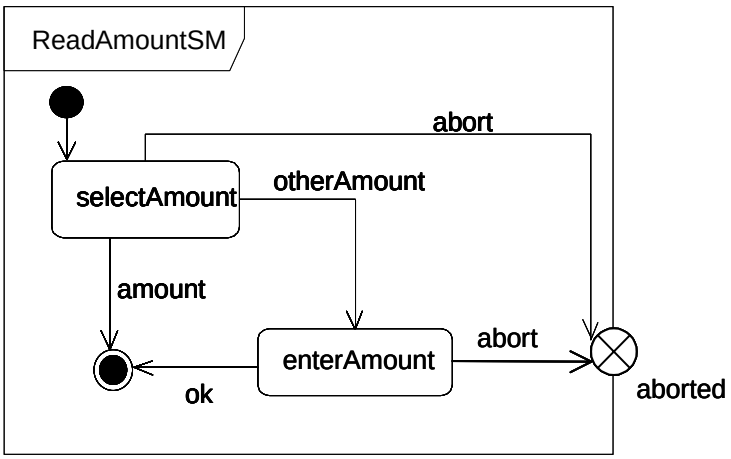


Figure 14-2. State machine with exit point definition.

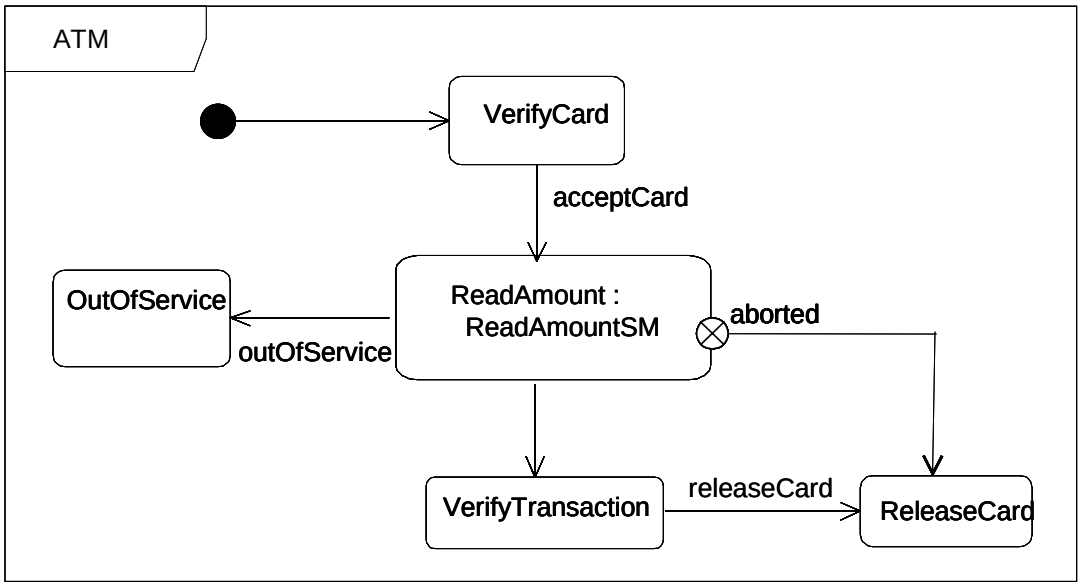
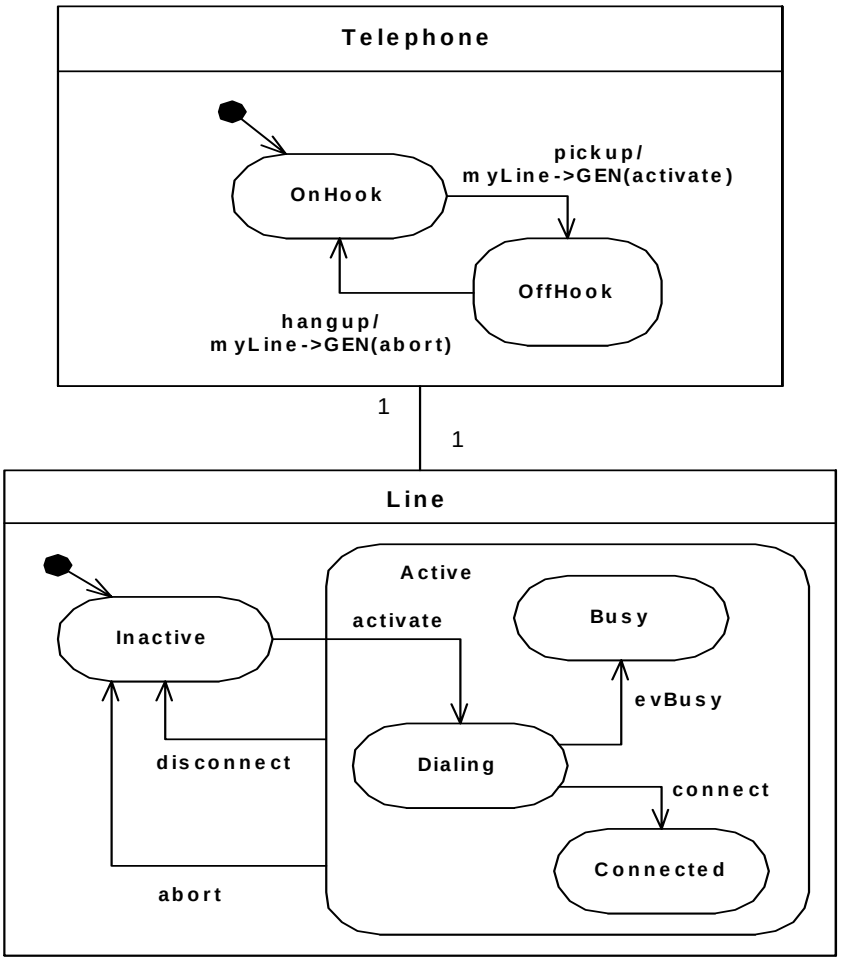


Figure 14-3. SubmachineState with usage of exit point.


Telephone
OnHook
pickup/
myLine->GEN(activate)
OffHook
hangup/
myLine->GEN(abort)
1
1
Line
Active
Inactive
activate
Busy
evBusy
disconnect
Dialing
connect
Connected
abort

15 Use Cases

15.1 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 is 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 "include", "extend", and "generalization". 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.

There are currently no changes to UML use cases.

15.2 Diagram elements

Graphical nodes that can be included in use case diagrams are shown in Table 18.

Table 18. *Graphical nodes included in Use Case diagrams.*

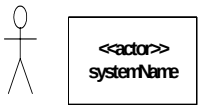
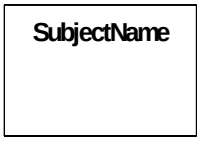
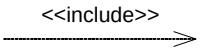
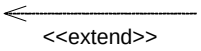
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Use Case		**UML::UseCase**	**Basic**
Actor		**UML::UseCase**	**Basic**

Table 18. *Graphical nodes included in Use Case diagrams.*

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Subject		**Role name on Classifier**	**Basic**

Graphical paths that can be included in use case diagrams are shown in .Table 19.

Table 19. *Graphical nodes included in Use Case diagrams.*

PATH TYPE	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Communica-tions path		**UML::Association**	**Basic**
Include		**Subclass of UML::Directed Relationship**	**Basic**
Extend		**Subclass of UML::Directed Relationship**	**Basic**

15.3 Package structure

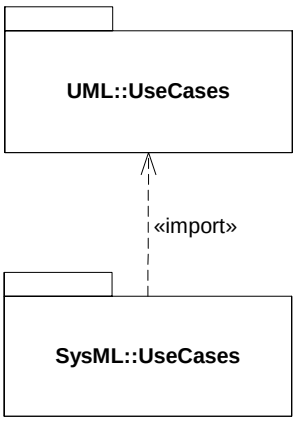
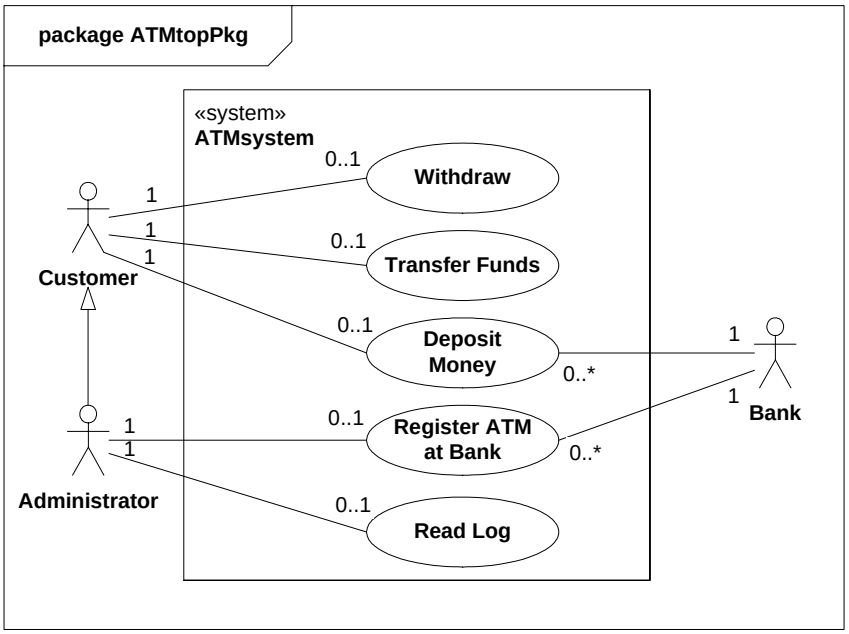


Figure 15-1. Package structure for SysML Use Cases.

15.4 UML extensions

15.5 Compliance levels

15.6 Usage examples



Part IV - Crosscutting Constructs

This Part specifies generic constructs that apply to both structure and behavior. It includes Allocations, Requirements, and Auxiliary Constructs. The Allocations chapter defines constructs that can be used to allocate a set of model elements to another (e.g., deploying software artifacts to hardware nodes). The Requirements chapter specifies constructs for system requirements and their relationships. The Auxiliary Constructs chapter specifies constructs for models and views, item flows between parts, and quantities in terms of units, values, and dimensions.

16 Requirements

16.1 Overview

A requirement states a capability or condition that a system must satisfy. Arequirement may specify a function that a system must perform or a performance condition a system must satisfy. SysML provides modeling constructs to represent requirements and relate them to other modeling elements.

A requirement can be decomposed into subrequirements, so that multiple requirements can be organized as a tree of compound requirements. Requirements can be related to each other, as well as to analysis, design, implementation and testing elements. A requirement can be generated or deduced from another requirement using the «derive» relationship. A requirement can be fulfilled by other model elements using the «satisfy» relationship. A requirement can be verified by test cases using the «verify» relationship. All of these are specializations of the UML «trace» relationship, which is used to track requirements and changes across models.

Modelers can customize requirements taxonomies by defining additional subtypes of the SysML «requirement» stereotype. For example, a modeler may want to define the following subtypes of «requirement»: OperationalRequirement, FunctionalRequirement, InterfaceRequirement, PerformanceRequirement, etc. Specialized requirements may restrict the types of model elements that may be assigned to satisfy the requirement. For example, a PerformanceRequirement may require a parametric relation along with an associated tolerance and probability distribution on the properties that satisfy the requirement.

A requirement can define its own properties, thereby providing a computable value to accompany the textual statement of the requirement. Initial values for properties can be assigned to requirements, and are inherited by specialized requirements.

16.2 Diagram elements

Table 20. *Graphical nodes included in Requirement diagrams*

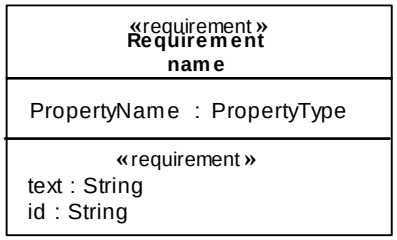
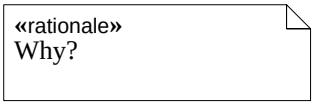
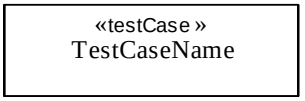
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Requirement		**SysML::«requirement»**	**Basic**
Rationale		**SysML::«rationale»**	**Basic**
TestCase		**SysML::«testCase»**	**Basic**

Table 21. *Graphical paths included in Requirement diagrams*

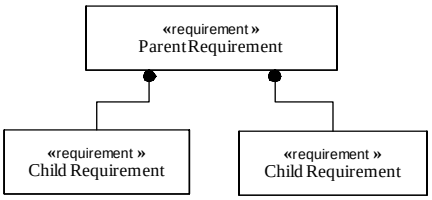
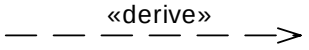
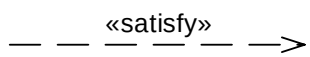
Path Type	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Namespace containment relationship		**UML::Kernel::Class::nested-Classifier**	**Basic**
Derive		**SysML::«derive»**	**Basic**
Satisfy		**SysML::«satisfy»**	**Basic**

Table 21. *Graphical paths included in Requirement diagrams*

PATH TYPE	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Verify	«verify»	**SysML::«verify»**	**Basic**
Trace	«trace»	**UML::«trace»**	**Basic**

16.3 Package structure

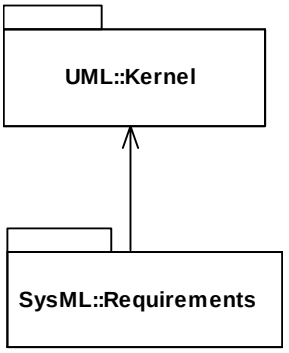


Figure 16-2. Package structure for SysML Requirement.

16.4 UML extensions

16.4.1 Stereotypes

Abstract syntax

Package Requirement

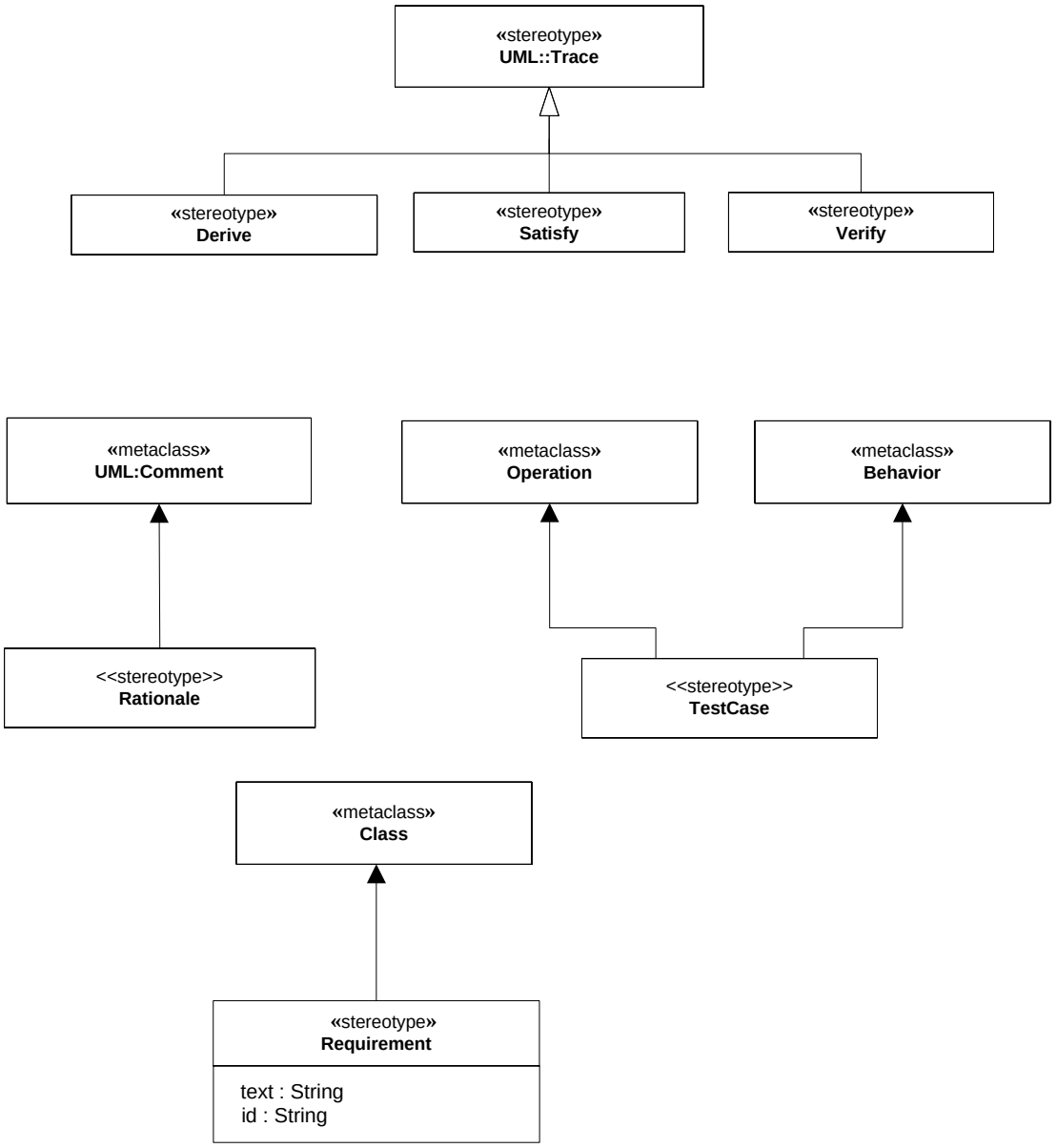


Figure 16-3. Abstract Syntax for Requirements Metamodel.

16.4.1.1 Derive

Description

A dependency relationship between two requirements in which a client requirement can be generated or inferred from the supplier requirements or additional design information. For example, an analysis requirement may derive from a business need, or

a design requirement in turn may derive from an analysis requirement. Derived requirements may refine or restate a requirement to improve stakeholder communications or to track design evolution. 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».

[2] The client must be an element stereotyped by «requirement».

Editorial Comment: Since Rationale applies to all model elements, it should be migrated to Classes or Auxiliary Elements.

16.4.1.2 Rationale

Description

An element that documents the principles or reasons for a modeling decision, such as an analysis choice or a design selection. It provides or references the basis for the modeling decision, and can be attached to any model element.

16.4.1.3 Requirement

Description

A requirement states a capability or condition that a system must satisfy. 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 UML::Kernel::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.

Constraints

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

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

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

[4] The subtypes of a class stereotyped by «requirement» must also be stereotyped by «requirement».

[5] A class stereotyped by «requirement» can participate in a «trace» dependency only if the other end of the dependency is not stereotyped by «requirement».

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

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

16.4.1.4 Satisfy

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».

16.4.1.5 Verify

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».

[1] The client must be an element stereotyped by «testCase».

16.4.1.6 TestCase

Description

A process or activity that is used to determine whether a system has fulfilled its requirements.

16.5 Compliance levels

The compliance levels are specified by the tables in Section 18.2.

16.6 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 16-5. 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.

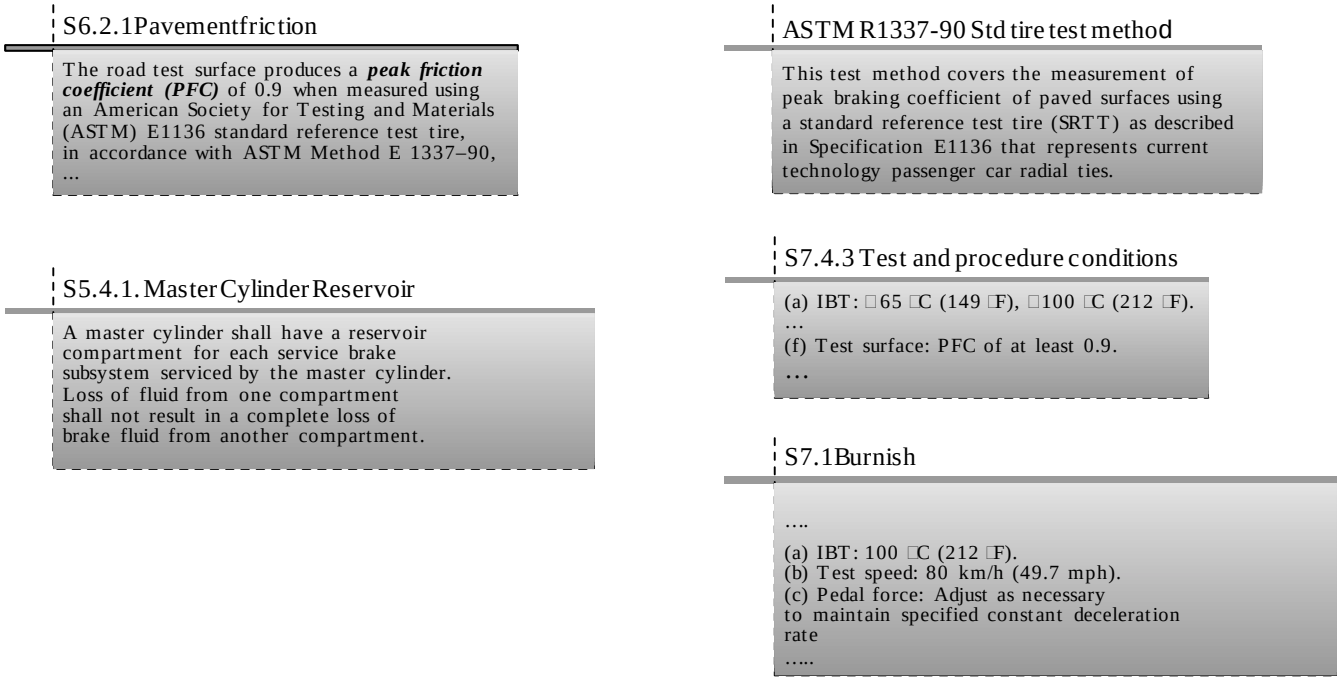


Figure 16-4. Source requirements from NHTSA specification 49CFR571.135.

16.6.1 Requirement decomposition

The diagram in Figure 16-5 shows an example of a compound requirement decomposed into multiple subrequirements.

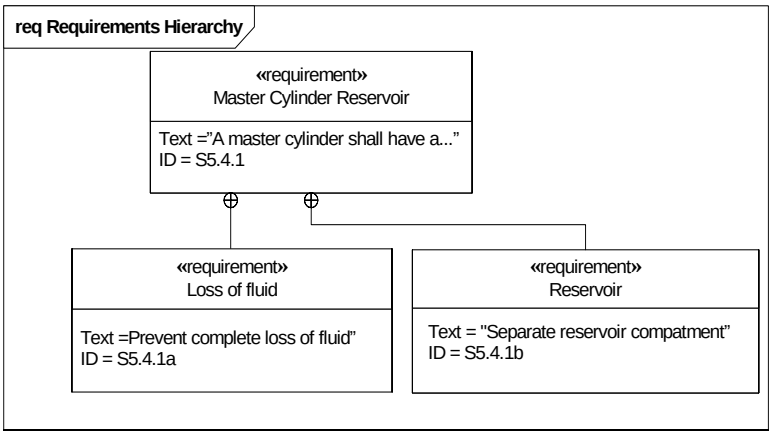


Figure 16-5. Decomposition of a compound requirement.

16.6.2 Requirements and design elements

The diagram in Figure 16-6 shows how traceability links between requirements and design—the «satisfy» relationship—can be shown.The design decision is further documented using a Rationale comment attached to the link..

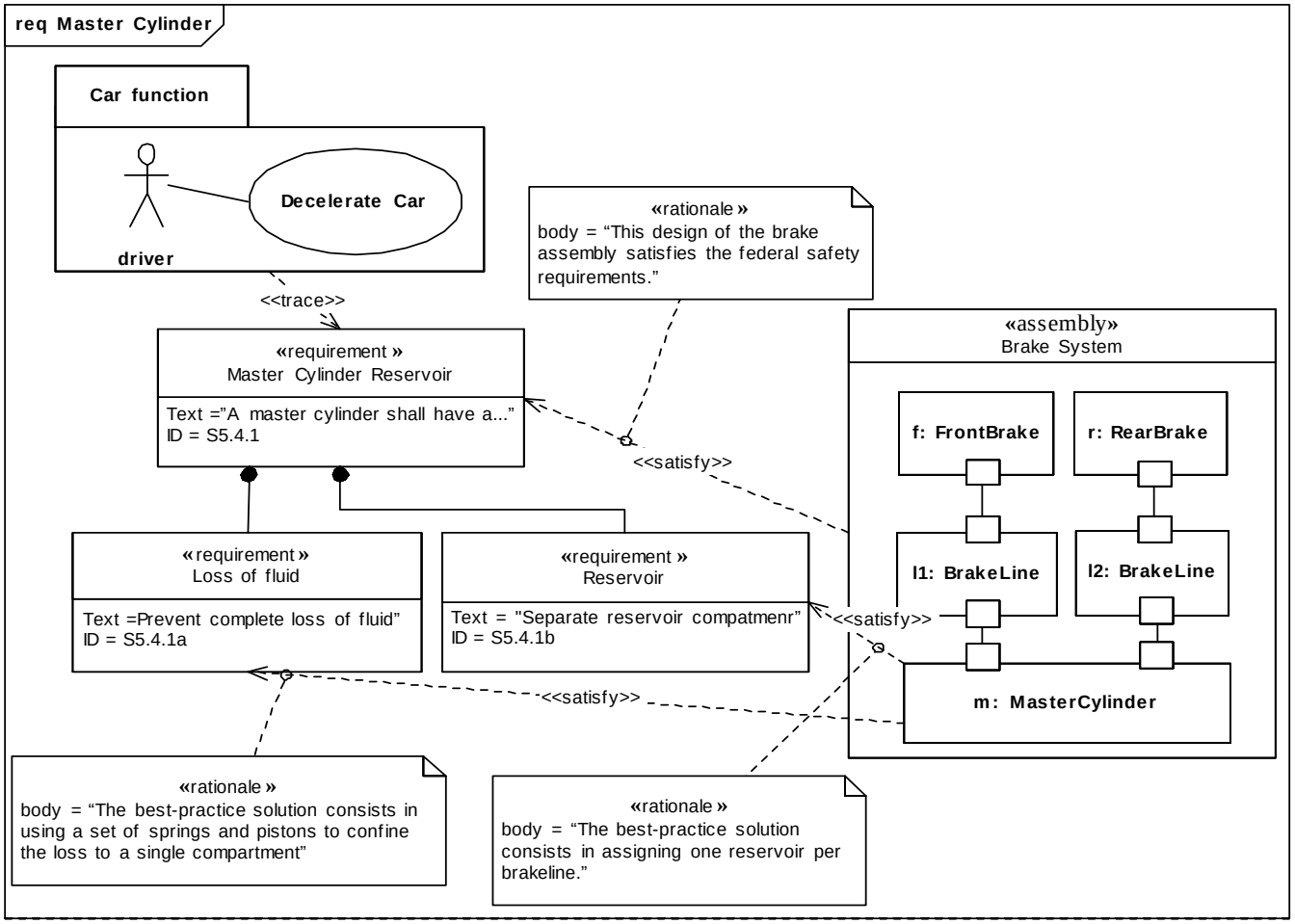


Figure 16-6. Links between requirements and design.

16.6.3 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 peformance 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.

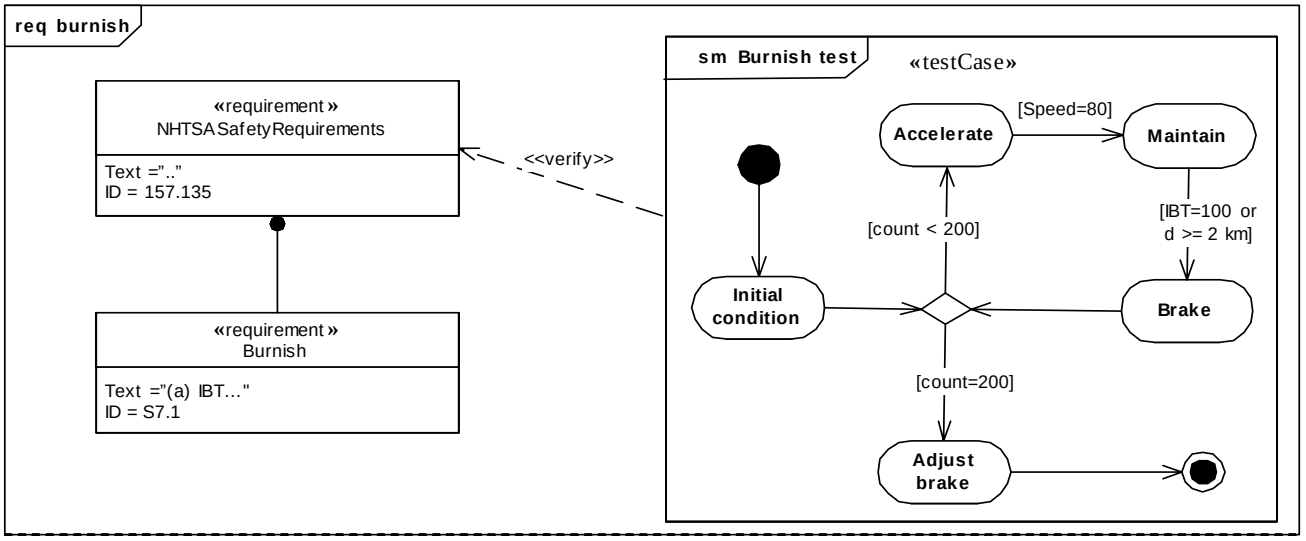


Figure 16-7. Linkage of a Test Case to a requirement.

16.6.4 Requirement specialization and properties

The diagram in Figure 16-8 shows how to specialize a requirement and specify properties. A requirements for radar tracking is defined with properties some of which have initial values. It is specialized for the two modes of of air tracking, with different maximum diostances in each case.

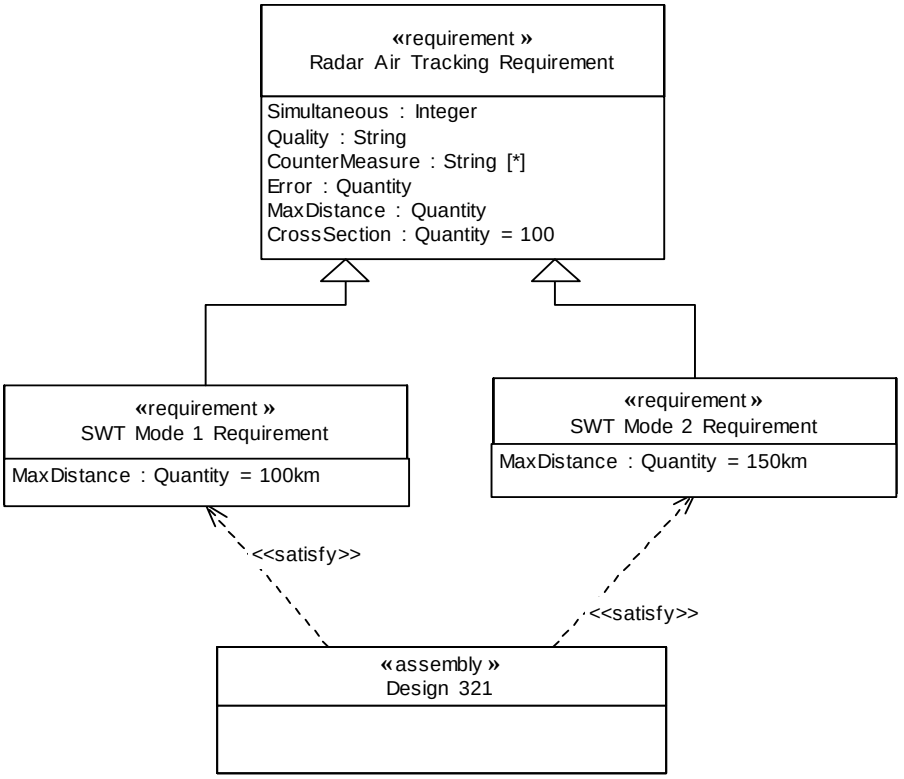


Figure 16-8. Specializing requirements and propertied requirements.

17 Allocations

17.1 Overview

Allocation is the term used by systems engineers to denote the organized cross-association of elements within the various structures or hierarchies of a user model. The systems engineer's concept of "allocation" requires a flexibility of expression suitable for abstract system specification, rather than a particular constrained method of system or software design. When searching for system organizational concepts, systems engineers are often required to associate various elements in a user model in abstract, preliminary, and sometimes tentative ways. It is inappropriate to force the systems engineer into the detail of rigorous methods too early in the development of a system architecture or design. The application of rigorous methods will surely follow, once the organizational concepts are more fully expressed. In the meantime, it is important and appropriate for systems engineers to use the notion of allocation to assess just how well a developing user model "hangs together".

The various types of elements generally associated with one another in practice have given rise to various uses of the word "allocation". This chapter does not try to limit the use of the term "allocation", but to provide a basic capability to support allocation in the broadest sense. See Appendix E for a brief overview of the types of allocations specifically considered in the development of the SysML language. This chapter focuses on the syntax with which these types of allocations may be expressed in a SysML model.

17.2 Diagram elements

Table 22. *Graphical nodes included in allocation diagrams*

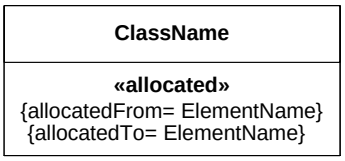
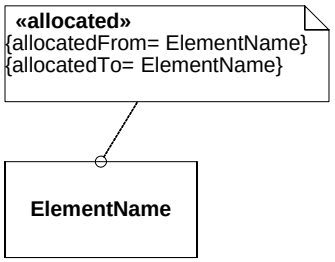
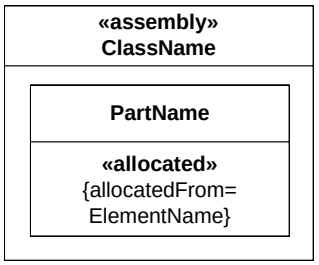
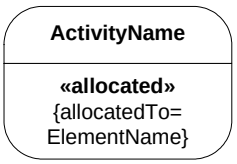
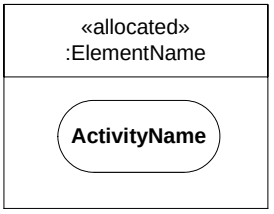
NODE TYPE	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX*	*COMPLIANCE*
Allocation derived properties displayed in compartment of Class.		SysML::Allocation::Allocated	Basic
Allocation derived properties displayed in Comment.		SysML::Allocation::Allocated	Basic
Allocation derived properties displayed in compartment of Part on Assemby Diagram.		SysML::Allocation::Allocated	Advanced
Allocation derived properties displayed in compartment of Action on Activity Diagram.		SysML::Allocation::Allocated	Advanced
Allocation Activity Partition		SysML::Allocation::Allocated	Advanced

Table 23. *Graphical paths included in allocation diagrams*

PATH TYPE	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX*	*COMPLIANCE*
Allocation (general)	«allocation»	SysML::Allocation::Allocated	Basic
Allocation (one to many)	Named Element «allocation» Named Element, Named Element	SysML::Allocation::Allocated	Basic

Tabular representations

Allocations can also be represented in tabular format. See section 16.6 Usage Examples for an example of a tabular representation of allocation.

17.3 Package structure

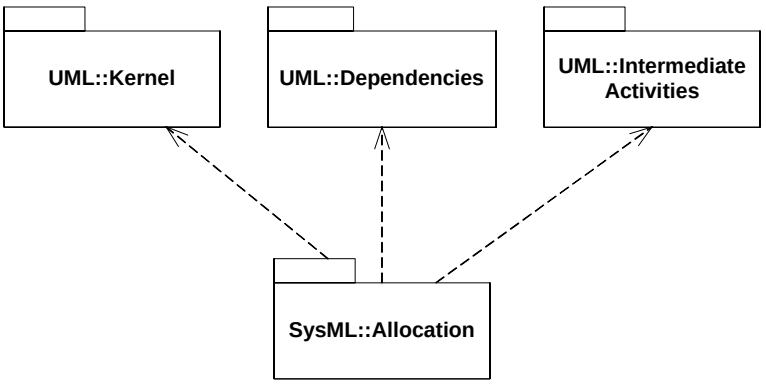


Figure 17-1. Package Structure for SysML Allocation

17.4 UML extensions

17.4.1 Stereotypes

Abstract syntax

Package Allocation

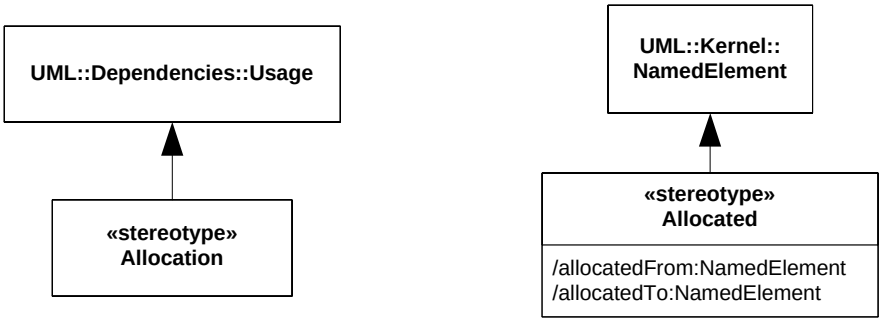


Figure 17-2. Stereotypse for Allocation (Basic)

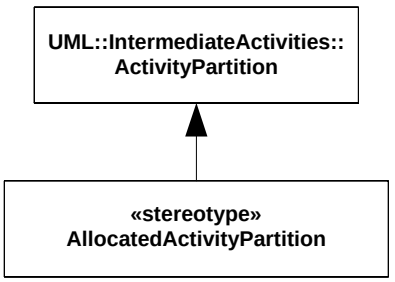


Figure 17-3. Allocation Activity Partition (Advanced)

17.4.1.1 Allocation

Description

Allocation is a mecanism for associating elements of different types, or in different hierarchies, at an abstract level. Allocation is used for assessing user model consistency and directing future design activity. It is expected that an «allocation» relationship between model elements is a precursor to a more concrete relationship between the elements, their properties, operations, attributes, or sub-classes.

Allocation is a stereotype of a dependency permissable between any two NamedElements. It is directional - one NamedElement must be designated client, and at least one NamedElement must be designated supplier.

Per systems engineering convention, the arrowhead end of the «allocation» dependency must be the element “allocated to”. This makes «allocation» an unconventional dependency, in that the client is the more abstract element. This is appropriate for the system engineering domain, and is allowed in UML2 (Superstructure Specification section 7.3.12). The supplier (arrowhead end of the dependency) can thus be viewed as “supplying” concreteness to the client in an unspecified way.

The «allocation» dependency may be further subtyped by the user with particular constraints regarding element type and attributes (see Appendix E for examples of «allocation» dependency subtypes).

Attributes

An «allocation» dependency may be named or numbered for traceability purposes.

Constraints

A single «allocation» dependency shall have only one client (no arrowhead), but may have one or many suppliers (arrowhead).

Subtypes of the «allocation» dependency should have constraints applied to supplier and client as appropriate. See Usage Examples section for examples.

17.4.1.2 Allocated

Description

Allocated applies to model elements that have at least one allocation relationship with another model element. Allocated elements may be either supplier or client of an «allocation» 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 «allocation» dependency. This stereotype provides for the properties “allocatedFrom” and “allocatedTo”, which are derived from the «allocation» dependency.

Attributes

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

/allocatedTo – the set of elements that are suppliers of an «allocation» 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 is only one /allocatedTo property per allocated model element.

/allocatedFrom – reverse of allocatedTo: the set of elements that are clients of an «allocation» whose supplier is extended by this stereotype (instance). The same characteristics apply as to /allocatedTo.

Note that the supplier or client may be an element (e.g. Activity, Class), or it may be a property (e.g. Action, Part, Connector). For this reason, it is important to use compound element names when deriving /allocatedFrom and /allocatedTo properties. An example of a compound element name is the form (PackageName::ElementName.PropertyName). Use of such compound names makes it clear that the «allocation» is refering to the definition of the element, or to it’s specific usage as a property of another element.

17.4.1.3 AllocatedActivityPartition

Description

AllocatedActivityPartition is used to depict an «allocation» relationship on an Activity diagram. The AllocatedActivityPartition is a standard UML2::ActivityPartition, with modified constraints such that any Actions within the partition must result in an «allocation» dependency between the Activity used by the Action, and the Classifier typing the partition.

Attributes

Same as UML2::ActivityPartition

Constraints

An Action appearing in an «AllocatedActivityPartition» will be the client of an «allocation» dependency. The Classifier typing the «AllocatedActivityPartition» will be the supplier of the «allocation» dependency.

17.4.2 Diagram extensions

An «allocation» relationship is represented diagrammatically by dependency. If an allocation has been subtyped (e.g. functionalAllocation), then only the subtype will be displayed in guillemets on the allocation dependency in the diagram (e.g. «functionalAllocation»).

The properties /allocatedFrom and /allocatedTo may be displayed in a compartment, or in a comment. In both cases, curly braces {} are used to express the property, and the ElementType is expressed in guillemets prior to the ElementName, as listed below:

{allocatedFrom= «Stereotype» ElementName}

{allocatedTo= «Stereotype» ElementName}

Use of «Stereotype» is optional. ElementName should be of the compound form (PackageName::ElementName.PropertyName) as necessary to distinguish the supplier/client as being an element or a property.

When applied to a classifier, an additional compartment is used by the «allocated» stereotype. This compartment is used specifically for display of allocatedFrom and allocatedTo properties, as described above. The allocation compartment may be elided from the diagram. Basic compliance requires the allocation compartment for Classes only. Advanced compliance requires the allocation compartment for Actions and Parts.

A comment may be used with any NamedElement. When used, the stereotype «allocated» will be used to distinguish the it from a constraint. Comments may be used on associations, including ActivityEdges and Connectors. (Basic compliance)

«AllocatedActivityPartition» is identified by the word allocation in guillemets («allocated») at the top of the name compartment of the partition.

17.5 Compliance levels

The basic compliance level includes all features necessary for use and display of Allocation dependencies.

The advanced compliance level includes graphical features providing a more intuitive or compact representation. Compartments on Actions in Activity Diagrams, and Parts in Assembly Diagrams fall into this category.

17.6 Usage examples

The following examples are provided as an overview for representing allocation in SysML diagrams. More complete examples describing the systems engineering use of allocation may be found in Appendix E Usages.

17.6.1 Allocations of Actions, Parts, and Classes

The following example depicts allocation relationships as property callout boxes (basic), property compartment of a Class (basic), and property compartments of Activities and Parts (advanced).

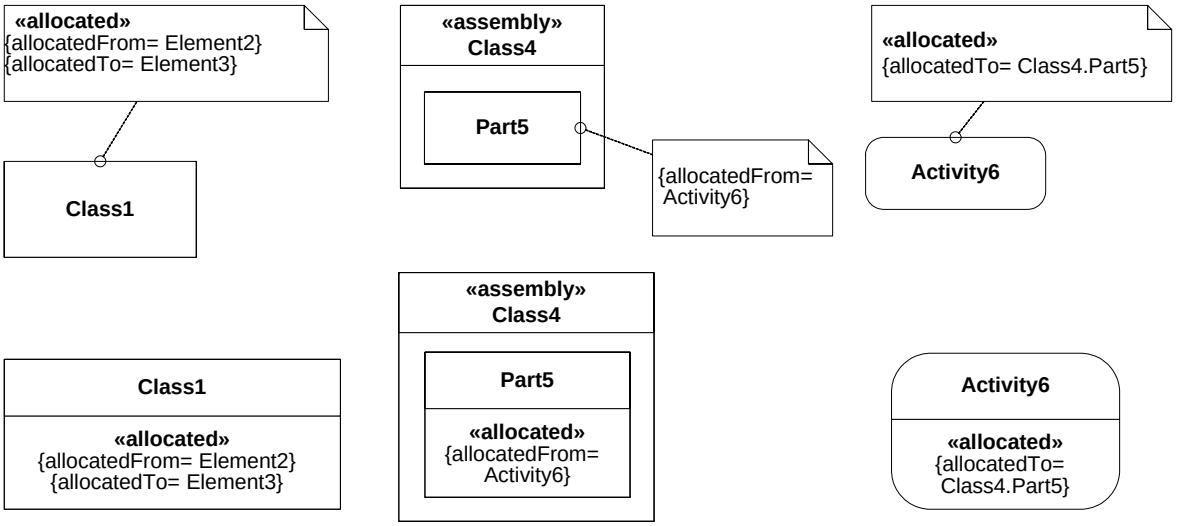


Figure 17-4. Allocation on Actions, Parts, Classes

17.6.2 Flow Allocations

The following example shows ObjectFlow allocation 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.

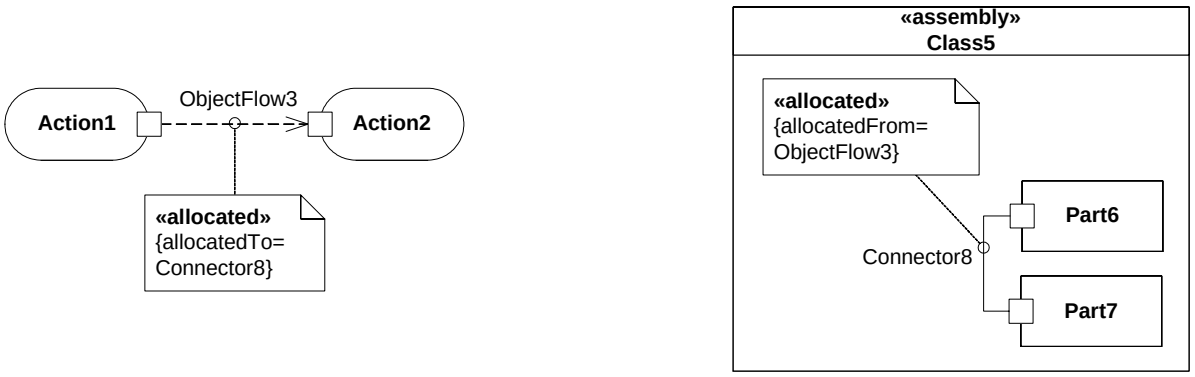


Figure 17-5. Example of Allocation from ObjectFlow to Connector

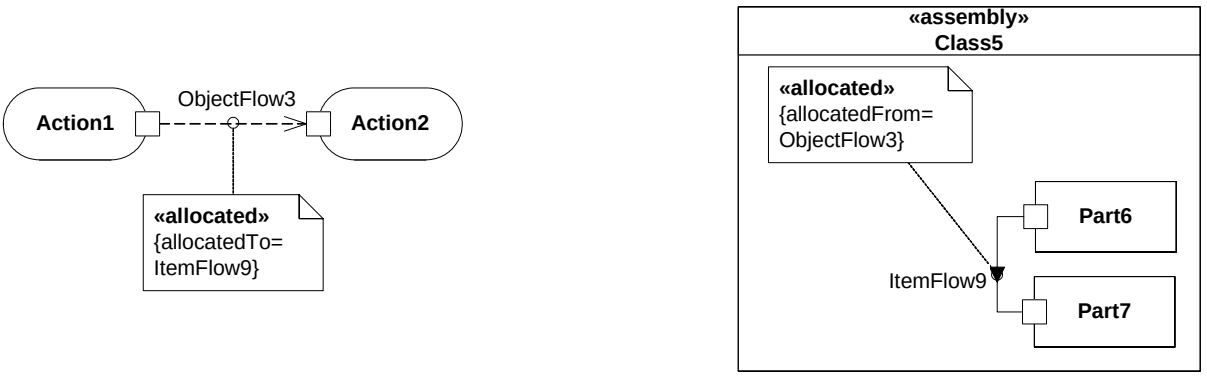


Figure 17-6. Example of Allocation from ObjectFlow to ItemFlow

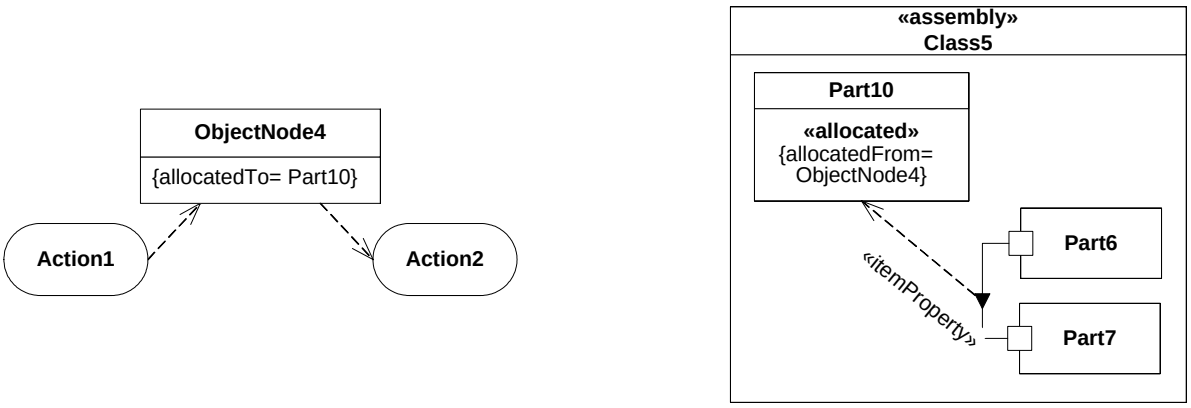


Figure 17-7. Example of Allocation from ObjectNode to ItemProperty

17.6.3 Tabular Representation

The table below is provided as a generic example of how the «allocation» dependency may be depicted in tabular form. ID refers to a unique number for each «allocation» dependency. AllocatedFrom is the client of the «allocation» dependency, and AloocatedTo is the supplier. Both ElementType and ElementName for client and supplier appear in the table.

Allocation Type is a user-defined subtype of allocation. See Appendix E for a discussion of possible Allocation Types, and more specific example of a tabular representation of allocation.

Table 24 Example Allocation Table

ID	Allocation Type (User Extension)	ElementType AllocatedFrom	ElementName AllocatedFrom	ElementType AllocatedTo	ElementName AllocatedTo
1	AllocationType1	ElementType1	ElementName1	ElementType2	ElementName2
2	AllocationType2	ElementType3	ElementName3	ElementType4	ElementName4

18 AuxiliaryConstructs

Editorial Comment: Auxiliary Constructs contains a mix of crosscutting constructs that are being refined and tested for usability.

18.1 Overview

This chapter defines elements and notations for item flows, model reference data, views and viewpoints, additional data types, dimensioned quantities, probability distributions, and property value constraints.

Input and output (I/O) items represent generic definition of things that flow and may include mass or energy flow as well as signals that contain information. Item flows represent the flow of an item between parts in the context of an enclosing assembly. Item flows are typed by the item which may have properties, and may be decomposed and specialized. The items and item flows are different from UML InformationItems and InformationFlows to accomodate the ability to allocate the items that type an item flow with the same item that types an object node in an activity diagram. Thus, the type of the output from an activity can be directly related to the type of the flow between parts. In addition, this approach enables the properties of items that flow to be used in parametric relationships.

SysML refines the UML concepts of viewpoints and views and adds notation to show views with a list of stakeholders and their concerns.

SysML adds real and complex numeric types to the Integer, Boolean, Enumerated, and String data types already defined by UML. Vector and compound structures can be modeled with standard the multiplicity elements of UML 2.

A model library declares classes that a user model may use to express quantities and distributions. A quantity is a subclass of primitive type real and references a a unit and a dimension.

18.2 Diagram elements

Table 25. *Graphical nodes to express auxiliary constructs*

Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Viewpoint	«viewpoint» stakeholders= <comma-separated list of stakeholders> concerns= <stakeholder concerns>	**SysML::Views::Viewpoint**	**Basic**

Table 25. *Graphical nodes to express auxiliary constructs*

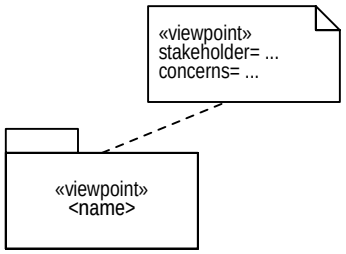
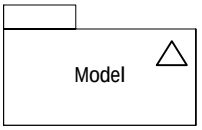
Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
View		**SysML::Views::View**	**Basic**
Model		**UML::AuxiliaryConstructs::Models::Model**	**Basic**

Table 26. *Graphical paths to express auxiliary constructs, at Basic compliance level*

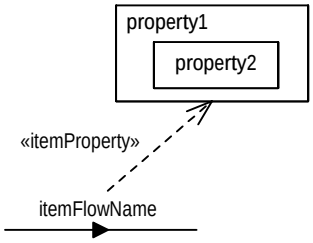
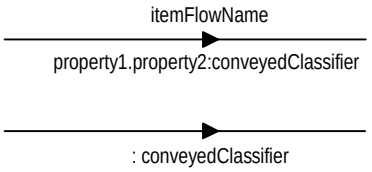
Path Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Item Flow		**SysML::ItemFlows::ItemFlow**	**Basic**
Item Flow		**SysML::ItemFlows::ItemFlow**	**Advanced**

Table 26. *Graphical paths to express auxiliary constructs, at Basic compliance level*

PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Item Flow		**SysML::ItemFlows::ItemFlow**	**Advanced**

18.3 Package structure

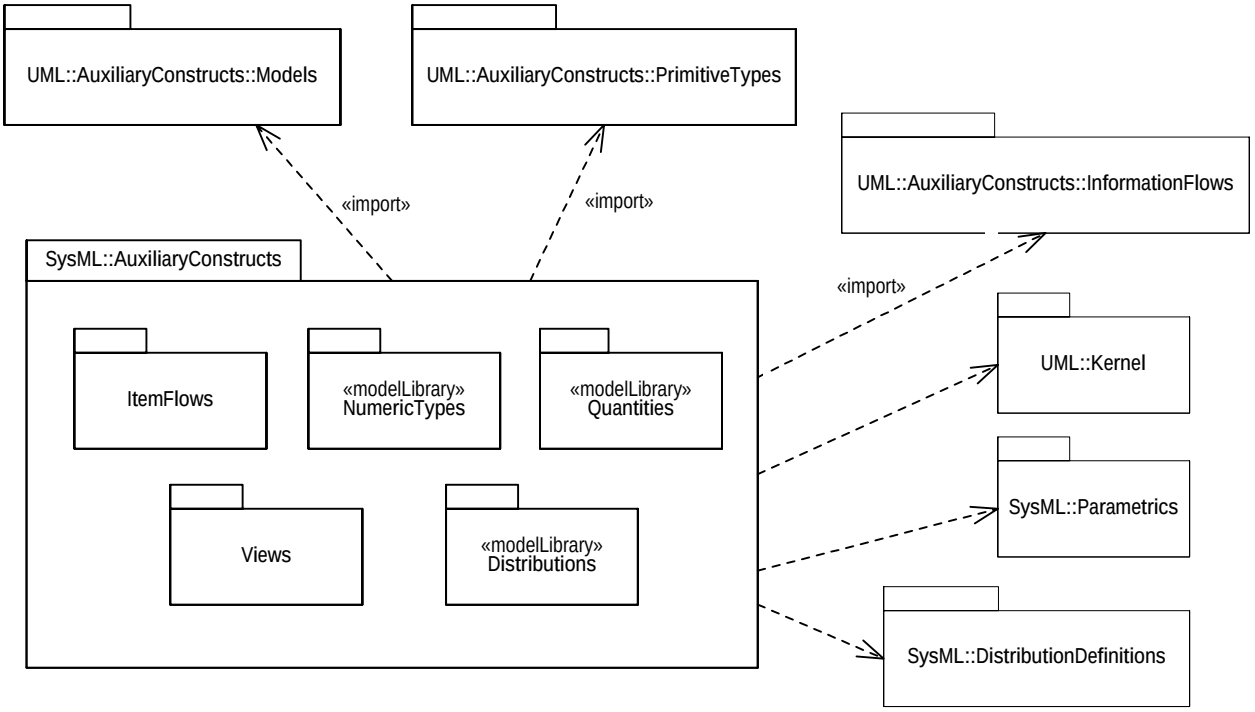


Figure 18-1. Package structure of SysML auxiliary constructs

18.4 UML extensions

18.4.1 Stereotypes

Abstract Syntax

Package ItemFlows

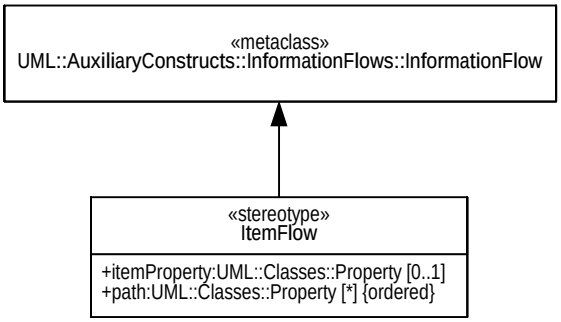


Figure 18-2. Stereotypes defined in package ItemFlows.

Package Views

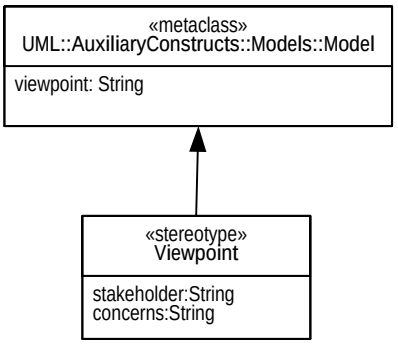


Figure 18-3. Stereotypes defined in package Views.

18.4.1.1 ItemFlow

Description

An Item Flow describes the flow of items within a connector of an assembly. An item is a classifier that represents the type of the thing that flows.

Attributes

- itemProperty: Property [0..1] A property that relates the instances of the item to the instances of its enclosing class. Many item flows can pass through the same property.
- path:Property[*] {ordered} This is an (optional) ordered set of properties which define a path from the context of the item flow to the item property that relates the instances of the item to the instances of its enclosing class.

Constraints

[1] An ItemFlow has no more than one conveyed classifier that types the item property.

18.4.1.2 Viewpoint

Editorial Comment: The Model, View and Viewpoint constructs are still being unified.

Description

A viewpoint describes a view. A view is a representation of a model from a particular viewpoint.

A viewpoint is defined as a stereotype of UML::Model, which itself is a specialization of UML::Package. A view is formed by importing the elements relevant to its viewpoint from other packages.

Attributes

- stakeholder — Set of stakeholders.
- concerns — The stakeholder concerns according to the viewpoint.

Constraints

[1] A viewpoint can only import elements, but does not own elements.

18.4.2 Diagram extensions

There are not diagram extensions defined.

18.4.3 Model Libraries

Editorial Comment: Guidelines for model library definitions are still being established.

18.4.3.1 Numeric Types

Abstract syntax

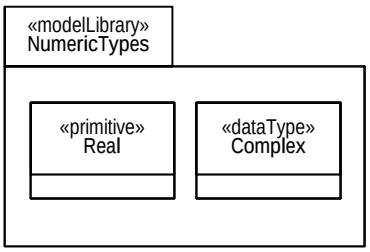


Figure 18-4. Content of NumericTypes model library

18.4.3.1.1 Complex

Description

A complex is a data type representing complex values.

18.4.3.1.2 Real

Description

A real is a data type representing real values.

18.4.3.2 Quantities

Abstract syntax

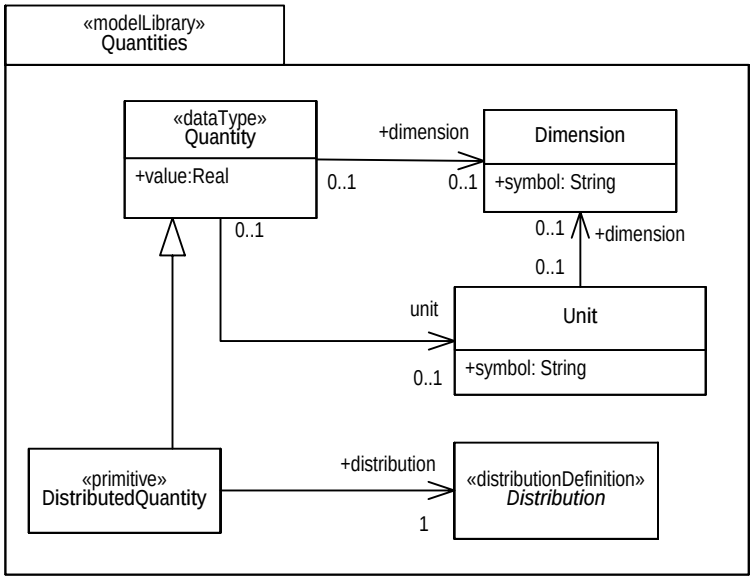


Figure 18-5. Model library Quantities

18.4.3.2.1 Dimension

Description

A dimension is a fundamental type that defines a basic type of value expressed by a quantity.

Dimensions specify fundamental types to express quantities. The SI unit system defines six dimensions: length, mass, temperature, time, electric current, and luminous intensity. See Appendix F for a model library defining these specific dimensions.

Attributes

- symbol:String Specifies the symbol of the dimension, e.g. L for length or M for mass.

18.4.3.2.2 DistributedQuantity

Description

A distributed quantity is a quantity with an associated distribution.

Associations

- distribution:Distribution Specifies the distribution of the quantity.

Constraints

[1] The dimension and unit of the distribution value must be compatible with the unit and dimension of the quantity.

Diagram extensions

The value of a distributed quantity is shown as follows:

<distributed quantity value> ::= '('<property name>'='<property value>
[',distribution='<distribution value>][','<property name>'='<property value>]')'*

where

- <distributed quantity value> is special notations for an instance value, i.e. it can be used as a <default> in the property BNF.
- <distribution value>::= '('<distribution type>')('<property name>'='<property value>[','<property name>'='<property value>]*')'
- <distribution type> is the name of the Distribution class, e.g. Uniform
- <property name> is the name of a property of DistributedQuantity
- <property value> is an expression that evaluates to the values of the property

It is allowed to omit an entry in the property list.

18.4.3.2.3 Quantity

Description

A quantity is a data type that specifies a value with a dimension and a unit of measure.

Associations

- dimension: Dimension Specifies the dimension of the value.
- unit: Unit Specifies the unit of the value.

Constraints

[1] If both dimension and unit are specified for a quantity, the dimension of the quantity must be the same as the dimension of the associated unit.

Diagram extensions

The value of a quantity is shown as follows:

<quantity value> ::= '('<property name>'='<property value> [','<property name>'='<property value>]')'*

where

- <quantity value> is special notations for an instance value, i.e. it can be used as a <default> in the property BNF.
- <property name> is the name of a property of Quantity
- <property value> is an expression that evaluates to the values of the property

It is allowed to omit an entry in the property list.

18.4.3.2.4 Unit

Description

A standardized quantity used as a unit of reference to express a quantity.

Note that compound units have to be defined in addition to simple ones. Future versions of SysML may support compound units as a relationship of simple units.

Attributes

- symbol: String Specifies the unit symbol, e.g. m for meter or kg for kilogram.

Associations

- dimension: Dimension [0..1] Specifies the dimension of the unit.

18.4.3.3 Distributions

The Distributions package provides a standardized framework for parametric constraints that constrain properties. It includes two stereotypes and a model library

Stereotypes

Abstract Syntax

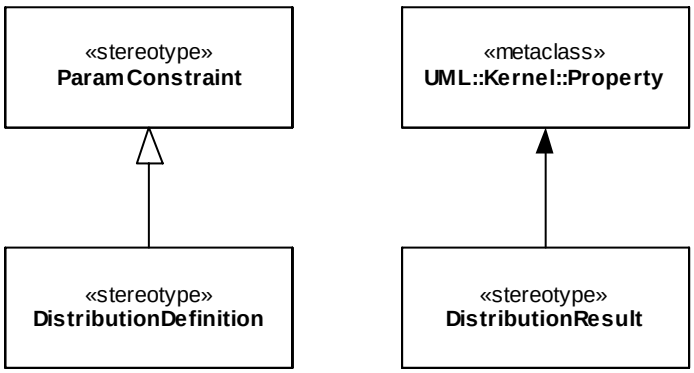


Figure 18-6. Distribution stereotypes.

18.4.3.3.1 DistributionDefinition

Description

A DistributionDefinition is a parametric constraint that constrains the value of one of its properties to be selected from within a range of possible values.

A «distributionDefinition» provides a reusable definition of the rule by which a value for the property may be selected within the range of its permissible values. The parameters of a «distributionDefinition» constraint may be used to specify the range of values or other details of this rule.

In addition to its input parameters that define its range of values and their probabilities, a «distributionDefinition» must be defined as holding a single property to which the stereotype «distributionResult» (see section 18.4.3.3.2) has been applied. The property with this stereotype holds the distributed value determined as a result of the distribution. This value may in turn be bound to other values in any containing context, just as input parameters of the distribution may also be bound or otherwise constrained to provide all required inputs.

Constraints

[1] A «distributionDefinition» must hold exactly one property to which the «distributionResult» stereotype has been applied.

18.4.3.3.2 DistributionResult

Description

DistributionResult is a stereotype that designates the distributed value defined within a DistributionDefinition.

A «distributionResult» stereotype must be applied to one of the parameters of a «distributionDefinition» (see section) to define the distributed value which is determined as a result of the distribution.

Constraints

[1] A «distributionResult» stereotype may be applied only to a property that belongs to a DistributionDefinition.

18.5 Compliance levels

The compliance levels are as defined in the tables in section 18.2.

18.6 Usage examples

18.6.1 Item flows

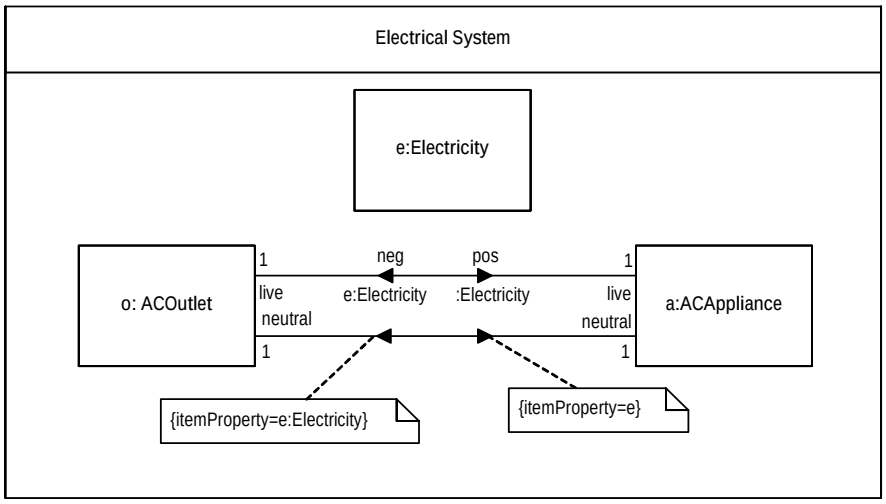


Figure 18-7. Item flows with different notations.

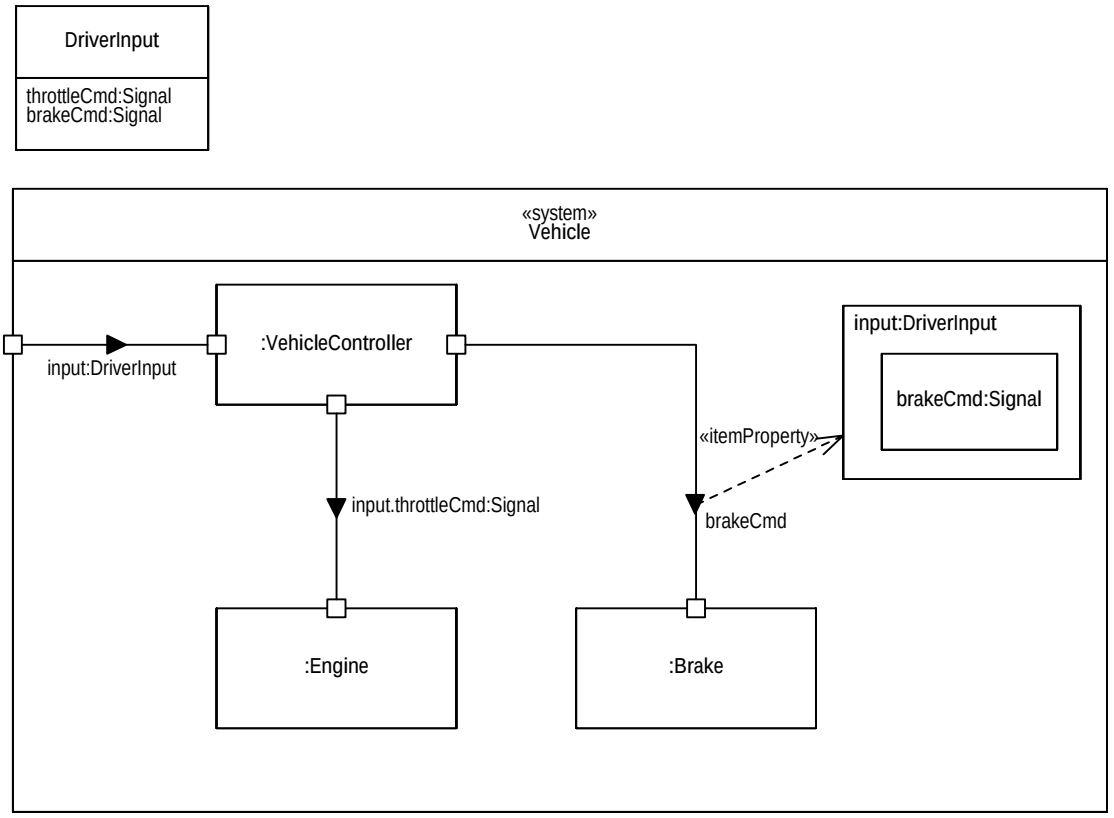


Figure 18-8. Item flows with nested properties

18.6.2 Viewpoints

Editorial Comment: The Model, View and Viewpoint constructs are still being unified.

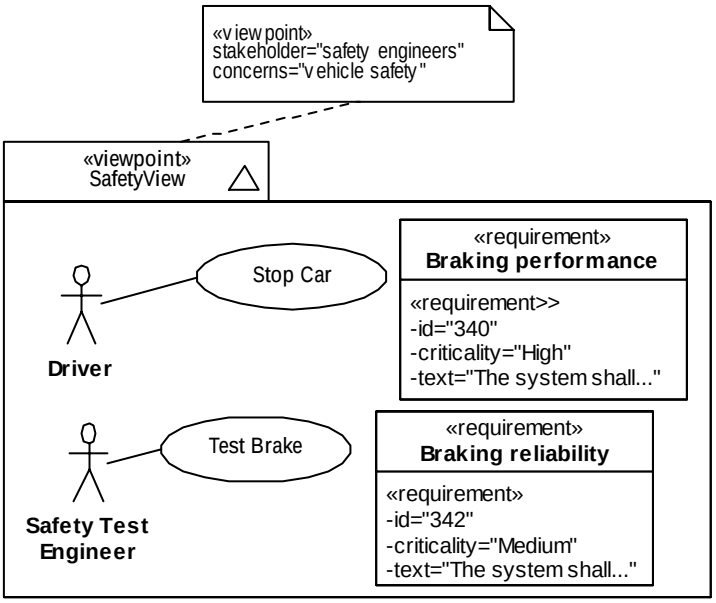


Figure 18-9. Viewpoints

18.6.3 Real types

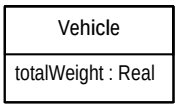


Figure 18-10. Real types

18.6.4 Definition of Quantity subclasses constrained with constant dimensions and units

The specific quantity types defined in this example are used by the Firing Range example in Chapter 11, Parametrics. This example provides local definitions of both the dimensions and units that it needs to define the specific Quantity subclasses that will be constrained to have constant values of these dimensions and units. Even though the Mass unit is defined in the SIUnits model library defined in Appendix F, this example does not import that library so that it may use its own naming convention for dimensions and so reserve their names (without the Dim suffix) for the Quantity subclasses themselves.

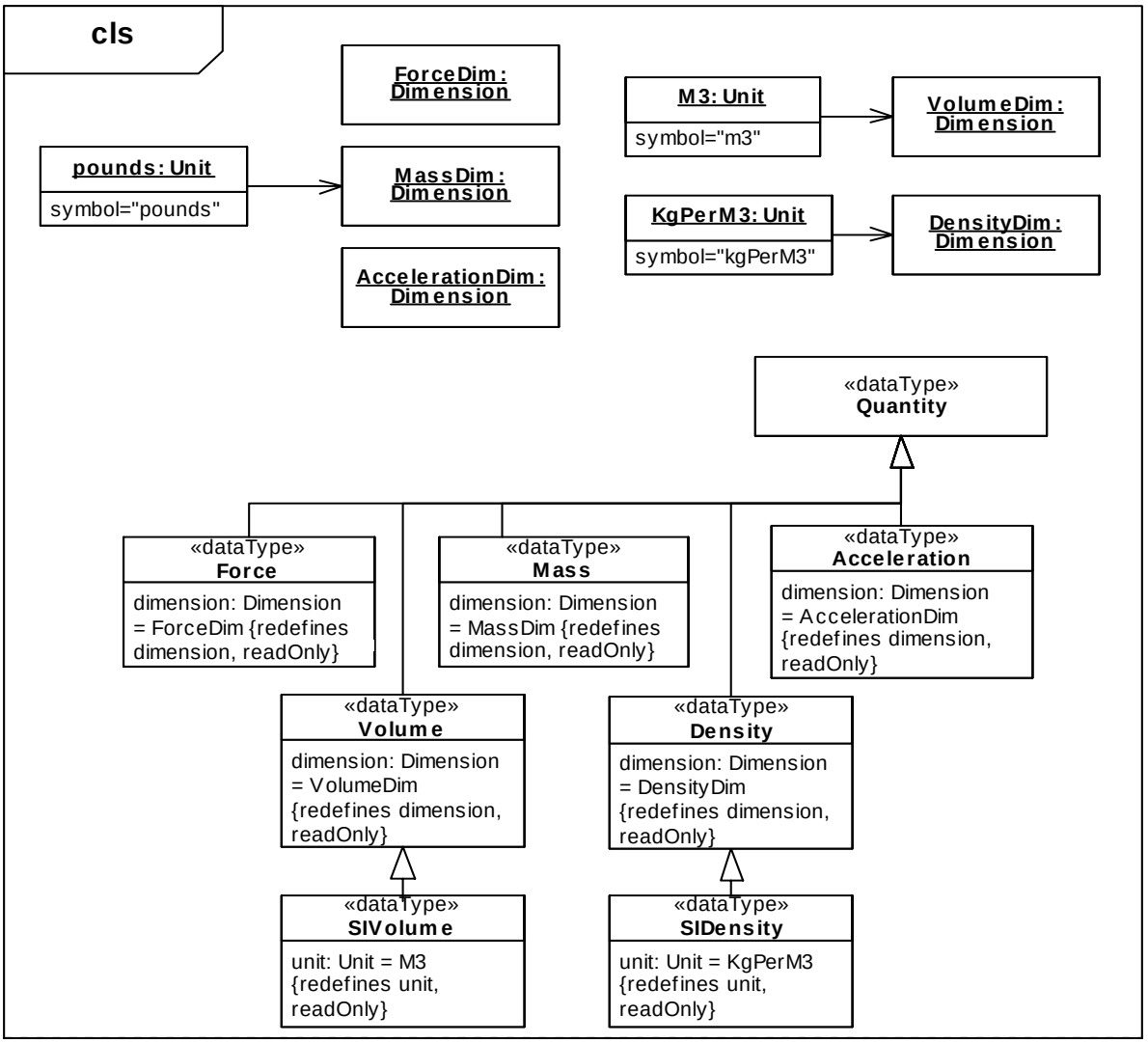


Figure 18-11. Quantity subclasses

18.6.5 Usage of Quantity and Distribution

This example shows the usage of a quantity and the distribution definition *Uniform*.

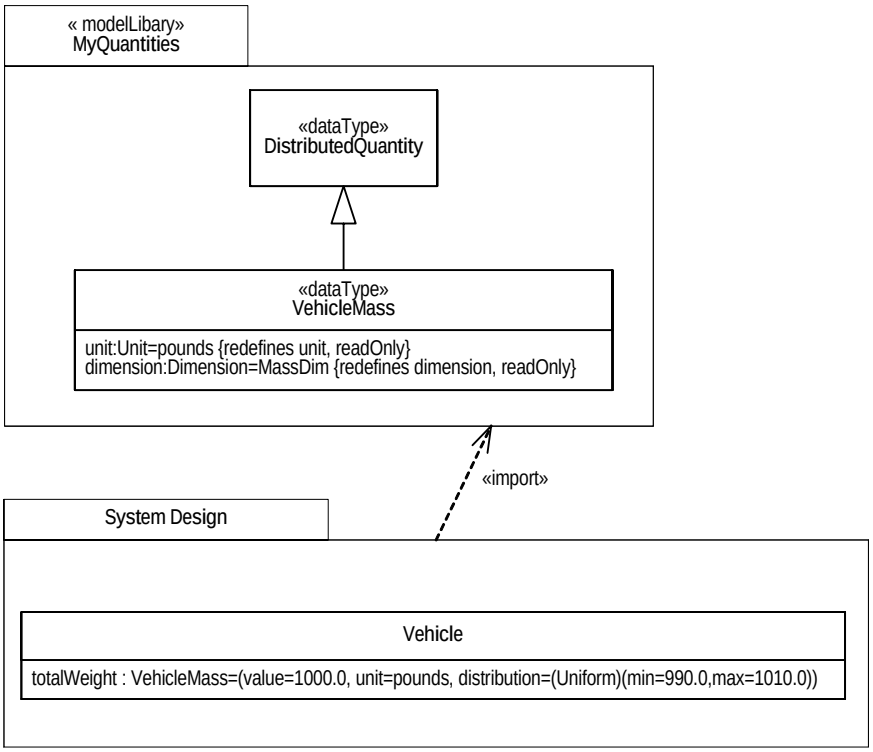


Figure 18-12. Quantities and Distributions

18.6.6 Constant design values

This example extends the Firing Range example in Chapter 11, Parametrics, to make use of constant Quantity values on attributes of the Shot100Kg subclass. The constant values are specified by instance specifications, which are referred to by name in the constant value property strings.

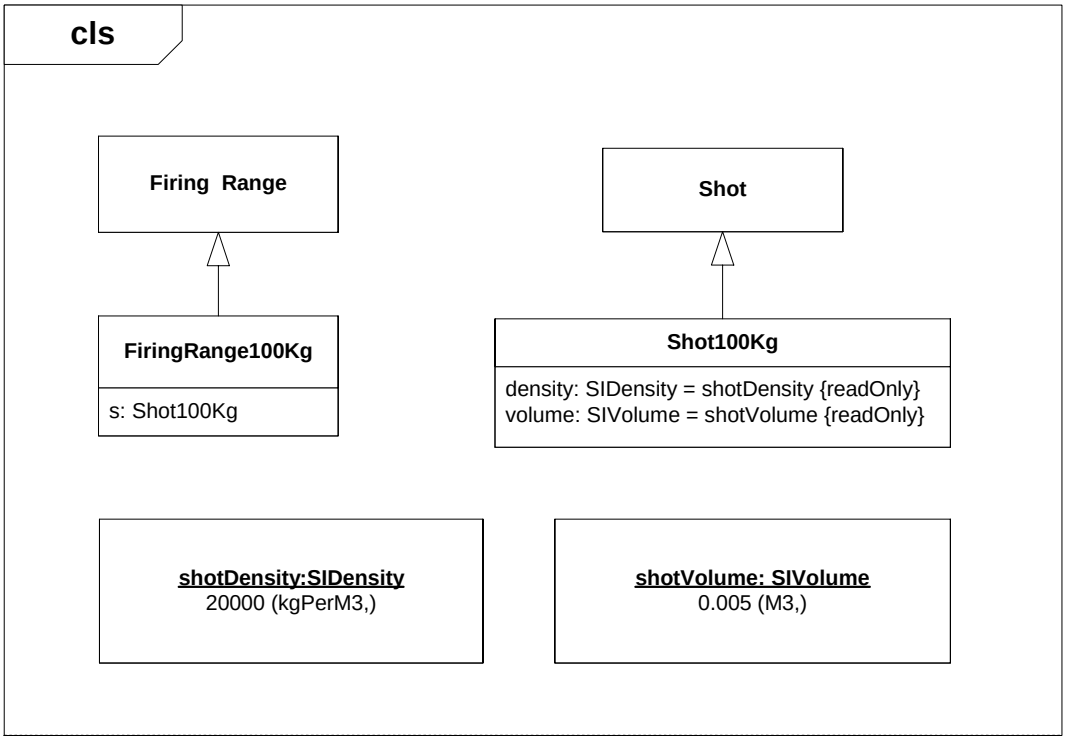


Figure 18-13. Quantities

19 Profiles

Editorial Comment: This chapter is not yet available for public review.

19.1 Overview

19.2 Diagram elements

19.3 Package structure

19.4 UML extensions

19.4.1 Stereotypes

19.4.2 Diagram extensions

19.5 Compliance levels

19.6 Usage examples

Part V - Appendices

This section contains non-normative appendices for this specification.

Appendix A.Diagrams

Editorial Comment: The diagram taxonomy in this appendix is being reconciled with the SysML package and specification structure. A future version of this appendix may be migrated to Part I, since its content is generally relevant to understanding SysML concete syntax.

A.1 Overview

SysML diagrams contains diagram elements (mostly nodes connected by paths) that represent model elements in the SysML model, such as packages, classes, and associations. The diagram elements are referred to as the concrete syntax.

The SysML diagram taxonomy, which reuses many of the major diagram types of UML, is shown in Figure A-1. In some cases, the UML diagrams are strictly re-used in SysML, whereas in other cases they are modified so that they are consistent with SysML extensions. There were some UML diagrams that are not being identified as unqiue diagram types within SysML. For example, the SysML deployment relationship that represents the deployment of software to hardware is integrated with the SysML Assembly diagram. As a result, SysML does not use the UML deployment diagram. Two new diagram types have been added for the initial version of SysML and others are planned for future versions.

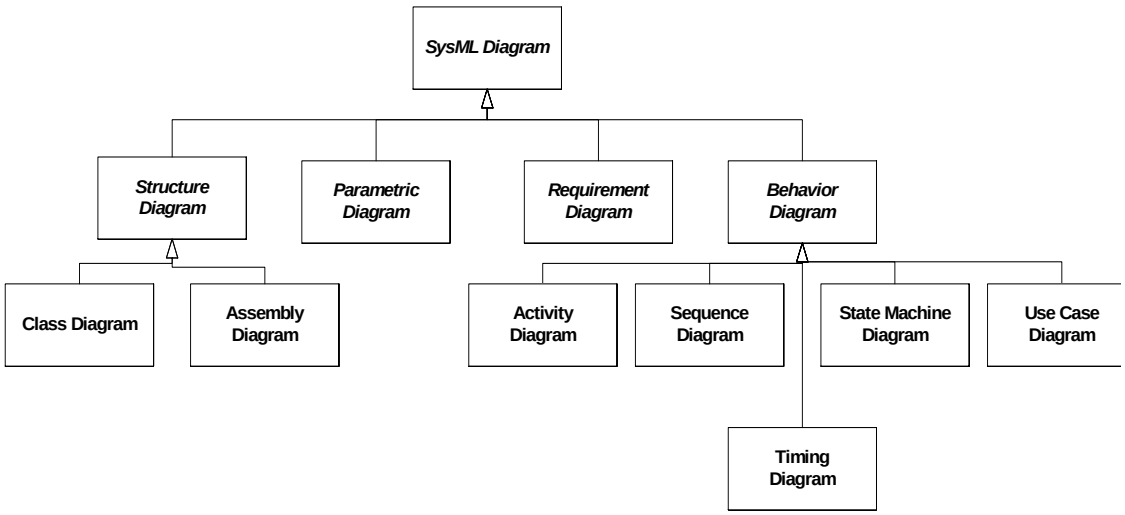


Figure A-1. SysML Diagram Taxonomy

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, such as the trace relationship, and the relationship between the requirements and the model elements that satisfy them.

The parametric diagram is a new SysML diagram type. A parametric diagram describes the parametric relationships among the properties associated with classes, assemblies and items that flow between them. The parametric 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 an internal structure). Consequently, the boundaries between the various kinds of diagram types are not strictly enforced as long as the implementation properly enforces the underlying semantics and constraints.

The model elements and corresponding concrete syntax that are represented in each of the ten SysML diagrams is described in the SysML chapters as indicated below.

- Activity Diagram - Activities chapter
- Assembly Diagram - Assemblies chapter
- Class Diagram - Classses chapter
- Interaction Overview Diagram - Interactions chapter (no longer part of SysML)
- Parametric Diagram - Parametrics chapter
- Requirements Diagram - Requirements chapter
- State Machine Diagram - State Machines chapter
- Sequence Diagram - Interactions chapter
- Timing Diagram - Interactions chapter
- Use Case Diagram - Use Cases chapter
- Other - Auxilliary chapter (item flows, ..), Allocation Chapter (allocation relationship, SysML Deployment, ..)

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

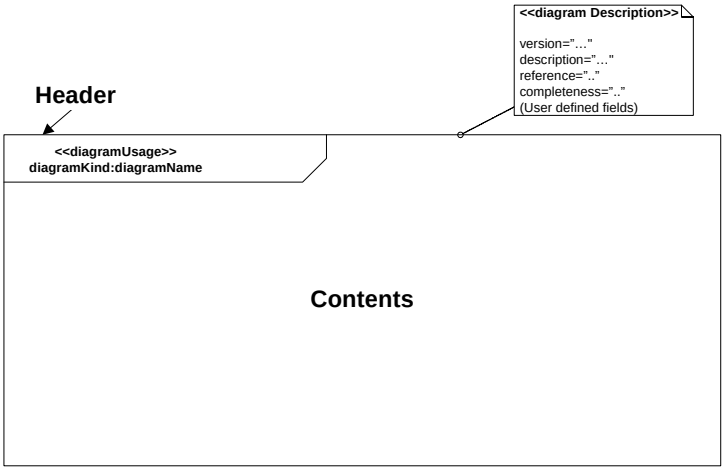


Figure A-2. Diagram Frame

The frame is a rectangle. The frame is sometimes used in cases where the diagrammed element has graphical border elements, like ports for assemblies, entry/exit points on statemachines, gates on interaction fragments, and pins for activities. In these cases, the frame can actually represent the enclosing model element versus just a diagram boundary. The frame should not be omitted but sometimes will 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 class diagram is a diagram where the primary symbols in the contents area are class symbols.

The heading is a string contained in a name tag (rectangle with cutoff corner) in the upper leftmost corner of the rectangle, with the following syntax:

[<kind>]<name>[<parameters>

SysML diagrams types may have the following names as part of the heading:

- activity (act)
- assembly (asm)
- class (cls)
- interaction overview (iov) - no longer part of SysML
- logic (log) - reserved
- parametric (par)
- requirement (req)
- sequence (seq)
- state machine (s-m)

- timing (tim)
- use case (u-c)
- verification (ver) - reserved

The diagram description is defined by a SysML Reference Data that is derived from a UML comment in the auxiliary chapter. The diagram description is attached to a diagram frame 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 user defined fields. These can be used to specify the level of completeness of the diagram or corresponding model or other information. 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 Auxiliary chapter).

SysML also introduces the concept of a diagram usage. This represents a unique usage of a particulr diagram type, such as a context diagram as a usage of an assembly, class, or use case diagram, or an entity relationship diagram as a usage of a class diagram. The diagram usage can be identified in the header above the kind, name and parameters as <<diagram usage>>. 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 “u-c” and <<diagramUsage>> with <<ContextDiagram>>. Applying a stereotype approach to specify a diagram usage could allow a tool implementation to check that the diagram constraints defined by the stereotype are satsified.

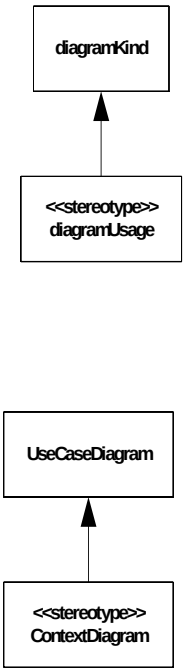


Figure A-3. Diagram Usages

Some of the predefined diagrams usages include:

- Activity diagram usage with control flow only - ControlFlow Diagram
- Activity diagram usage with swim lanes - SwimLane Diagram
- Class diagram usage for a assembly hiearchy - AssemblyHiearchy
- Class diagram usage for a function hiearchy - ActivityHierarchy
- Class diagram usage for an item hiearchy - ItemHierarchy

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 elaboration of the model element can include the following:
  - activity diagram - call behavior actions that call activities that refer to other activity diagrams.
  - assembly diagram - parts can refer to other assembly diagrams.
  - class diagram - structured classes can refer to other class diagrams
  - interaction overview diagram - interaction fragments can refer to other sequence diagrams.
  - parametric diagram - parametric relationships can be decomposed into other parametric diagrams.
  - requirement diagram - requirements can be decomposed into other requirement diagrams.
  - sequence diagram - interaction fragments can refer to other sequence diagrams
  - state machine diagram - states can refer to other state machine diagrams.
  - timing diagram - Segments of a timeline can refer to other timing diagrams
  - use case diagram - use case can refer to 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". Whenver practical decompose a model element 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 are allowed for any diagram type
- SysML provides the capability to represent a document using the UML 2 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 describe in this section should conform to the Diagram Interchange Standard to facilitate exchange of diagram and layout information.
- 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 suchas interface definitions, requirements traceability matrixes, and allocation

matrixes between various types of model elements. They also are often convenient mechanims to represent 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).

Appendix B. Sample Problem

B.1 Purpose

The purpose of this appendix 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.

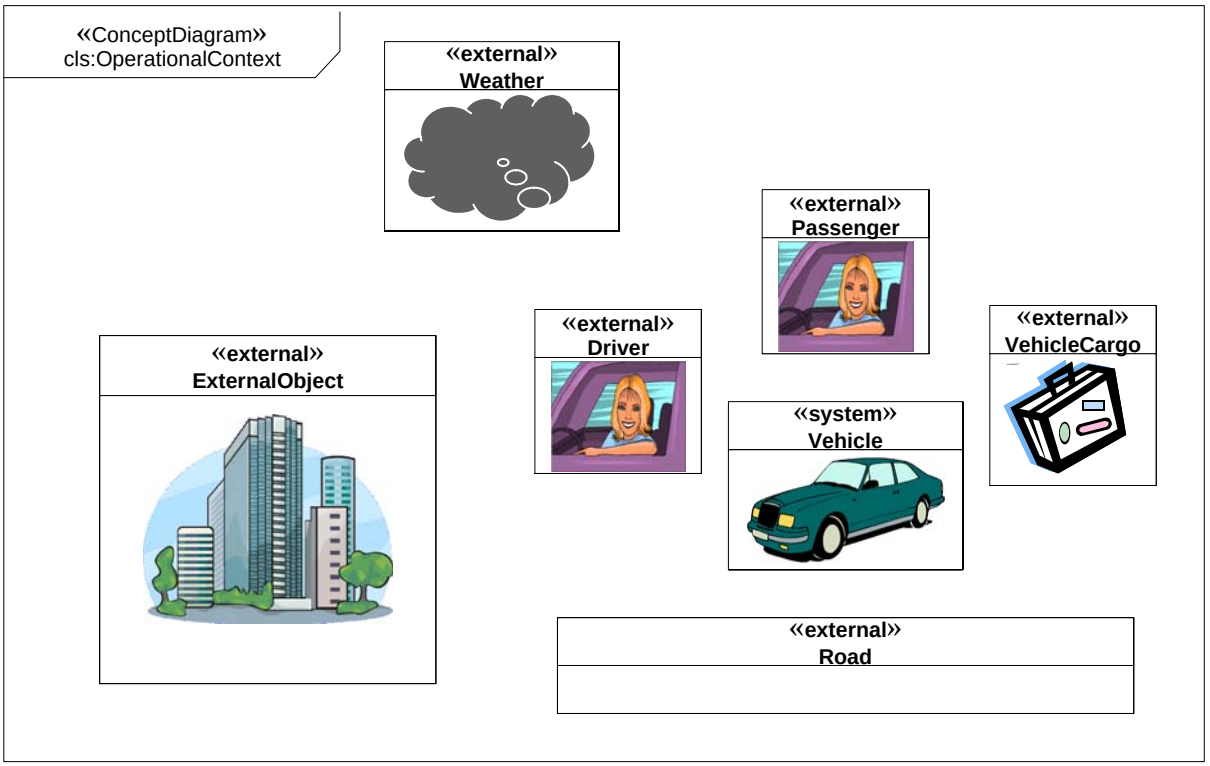
B.3 Problem Summary

The sample problem describes the use of SysML as it applies to the development of an automobile (Vehicle System). The problem is derived from a marketing analysis which indicated the need to increase the acceleration of the automobile from its current capability. Only a small subset of the functionality and associated vehicle system requirements and design are addressed to highlight this application.

B.4 Diagrams

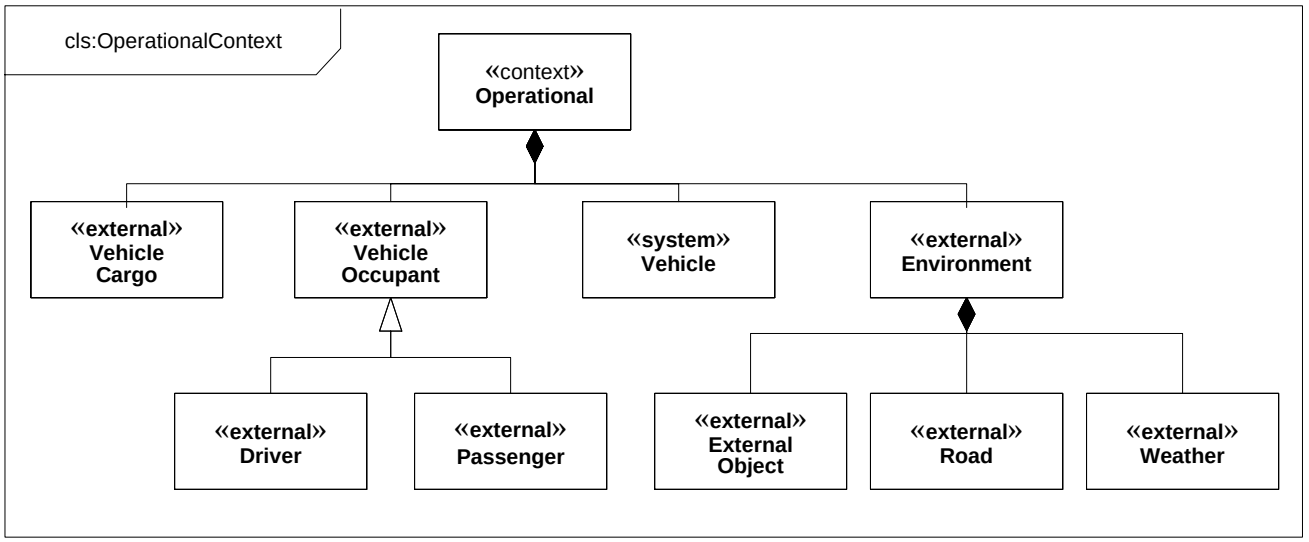
B.4.1 Concept Diagram for the “Vehicle System Operational Context”

The Concept Diagram for the “Vehicle System Operational Context” is a usage of a class diagram that depicts some of the top level entities to be modeled. The diagram usage enables the modeler or methodologist to specify a unique usage of a SysML diagram type using the extension mechanism described in the Diagram Appendix A. 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 are user defined stereotypes that are not part of SysML, but help the modeler to identify the system of interest relative to its environment. Each entity 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 captured in the semantics. Also, a background such as a map can be included to provide additional context. The associations among the classes can represent abstract relationships among the entities (not included here).



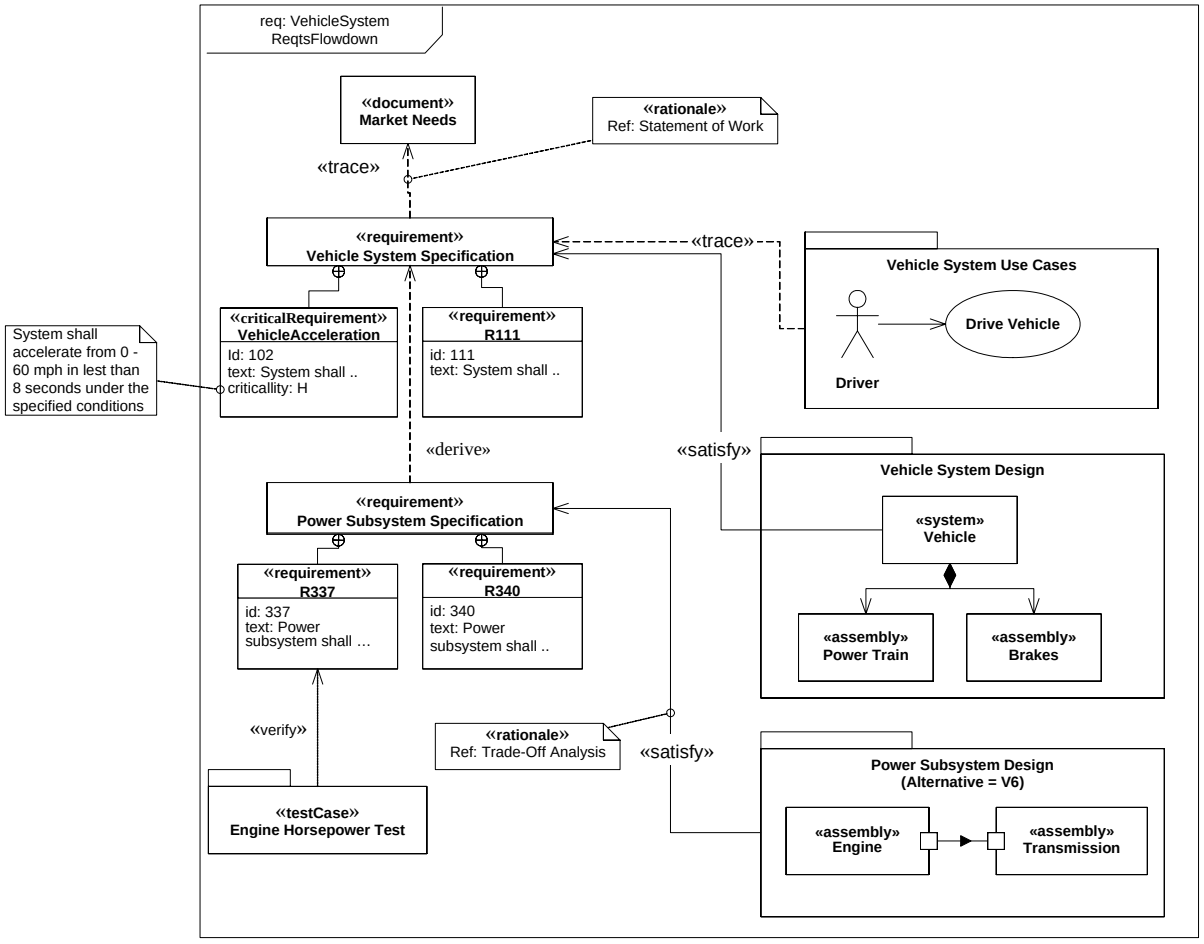
B.4.2 Class Diagram for the “Vehicle System Operational Context”

The Class Diagram for the “Vehicle System Operational Context” depcits a top level depiction of the context for the system under development in terms of the external entities the system will interact with. This refined representation of the “Vehicle System Operational Context” in 0.4.1 includes some composition and specialization relationships between the conceptual entities. In addition, a top level composite class called “Operational” has been added. The context, system, and external are user defined stereotypes that are not part of SysML, but help the modeler to identify the system of interest relative to its environment.



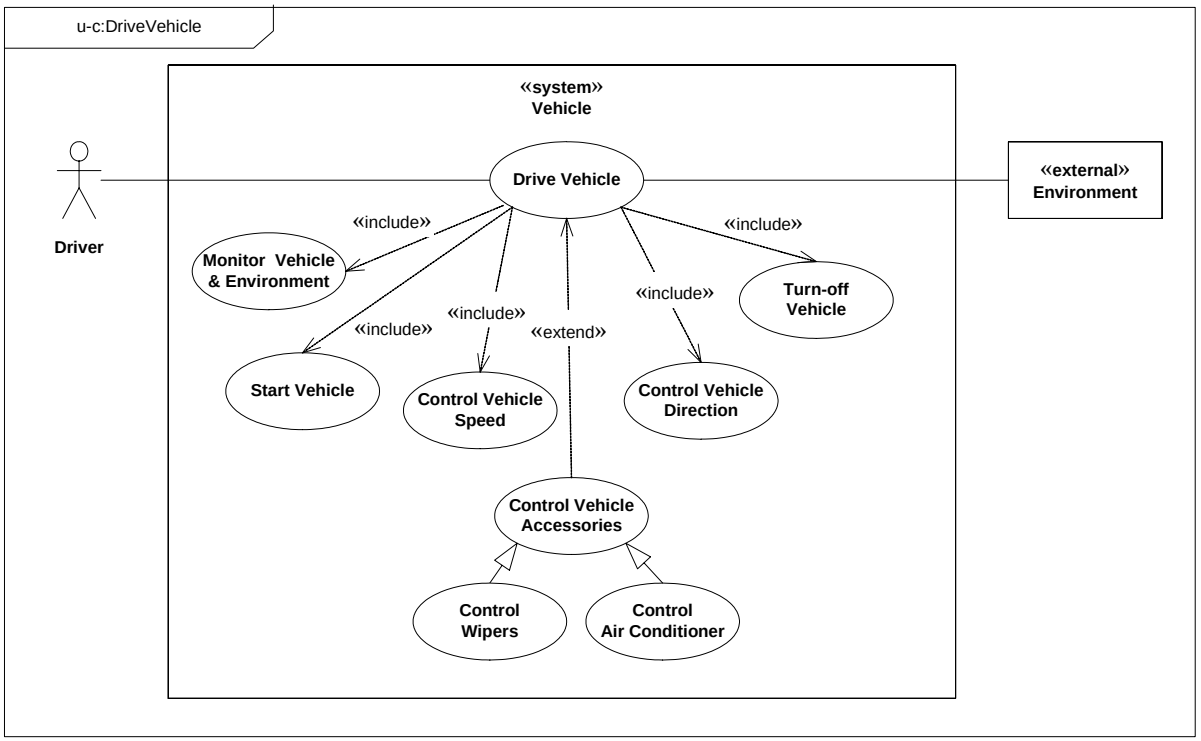
B.4.3 Requirement Diagram for the “Vehicle System Requirements Flowdown”

The requirement diagram for the “Vehicle System Requirements Flowdown” depicts a high level view of the requirements flowdown. This diagram shows the trace relationship between the vehcile system specification and a top level market needs document, along with a reference to the statement of work that provides the rationale for this trace. The vehicle system specification contains many text based requirements. A few requirements are highlighted including the critical requirement for the vehicle to accelerate from 0 - 60 mph in less than 8 seconds. The critical requirement is an example of a requirement sterteotype subclass with a criticality property as described in the Model Library Appendix. A use case traces to the vehicle specification to provide further refinement of the text based requirements. The vehicle system is intended to satsify the vehicle system specification. A power subystem specification is a lower level specification that is derived from the vehicle system specification. The power subystem design package satsifies this power subystem specification. The satsify relationship includes a design rationale that refers to a trade study to support the selection of this design. An engine horsepower test case is also shown to support verification of the power subsystem requirements. The tool implementation is expected to provide both a tree stucture (i.e. an explorer view) of the the requirement specifications and alternative tabular formats as referred to in the diagram appendix.



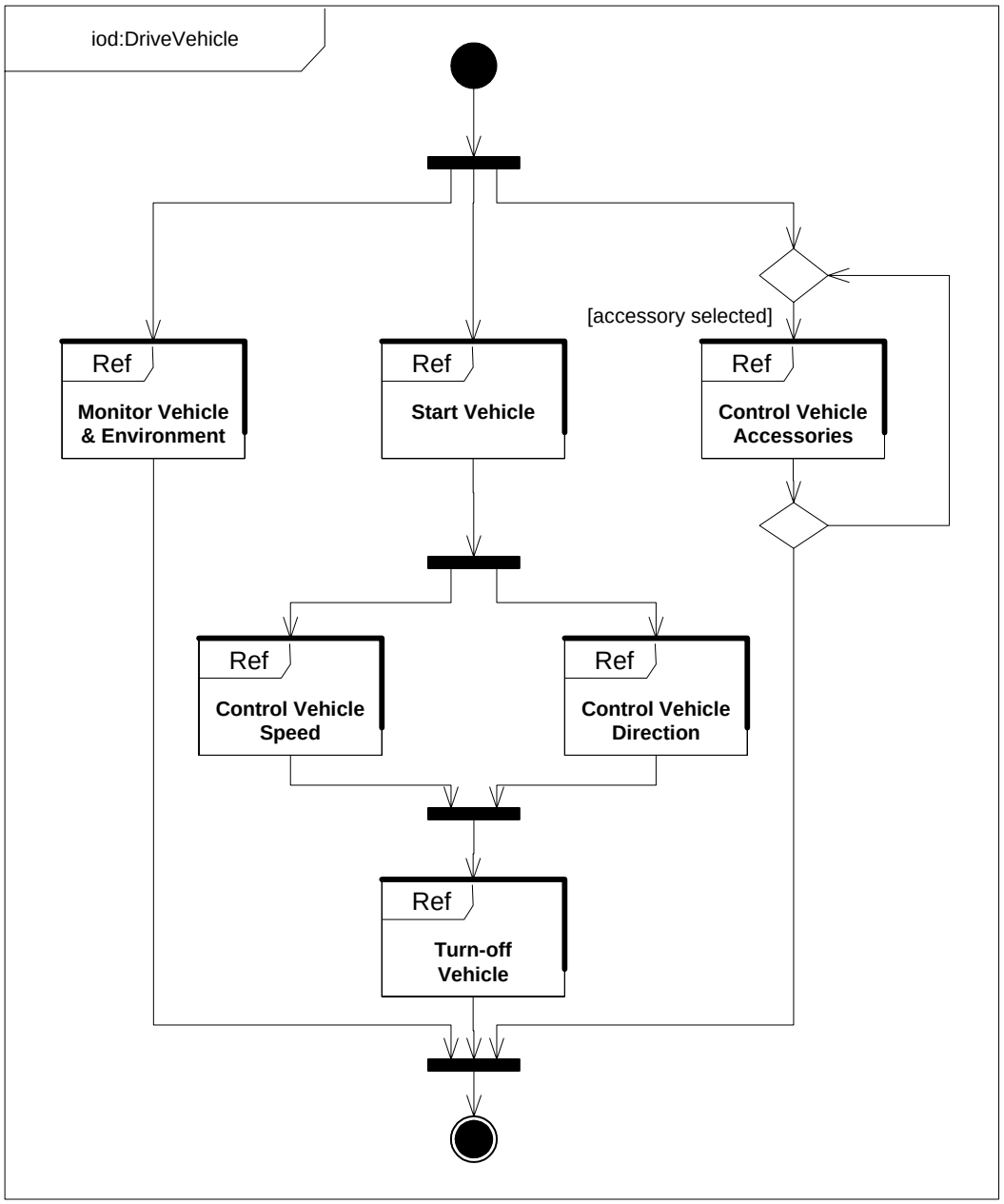
B.4.4 Use Case Diagram for “Drive Vehicle”

The use case diagram for “Drive Vehicle” depicts the drive vehicle usage of the vehicle system. Several <<include>> use cases are shown along with an <<extend>> use case that is optionally performed to support control of vehicle accessories. Specialization of use cases is also shown for controlling vehicle accessories. The subject and the actors (driver and environment) interact with the system to realize the use case. The actors correspond to the external classes in 0.4.2. (Note:UML 2 actors are a subclass of classifiers.)



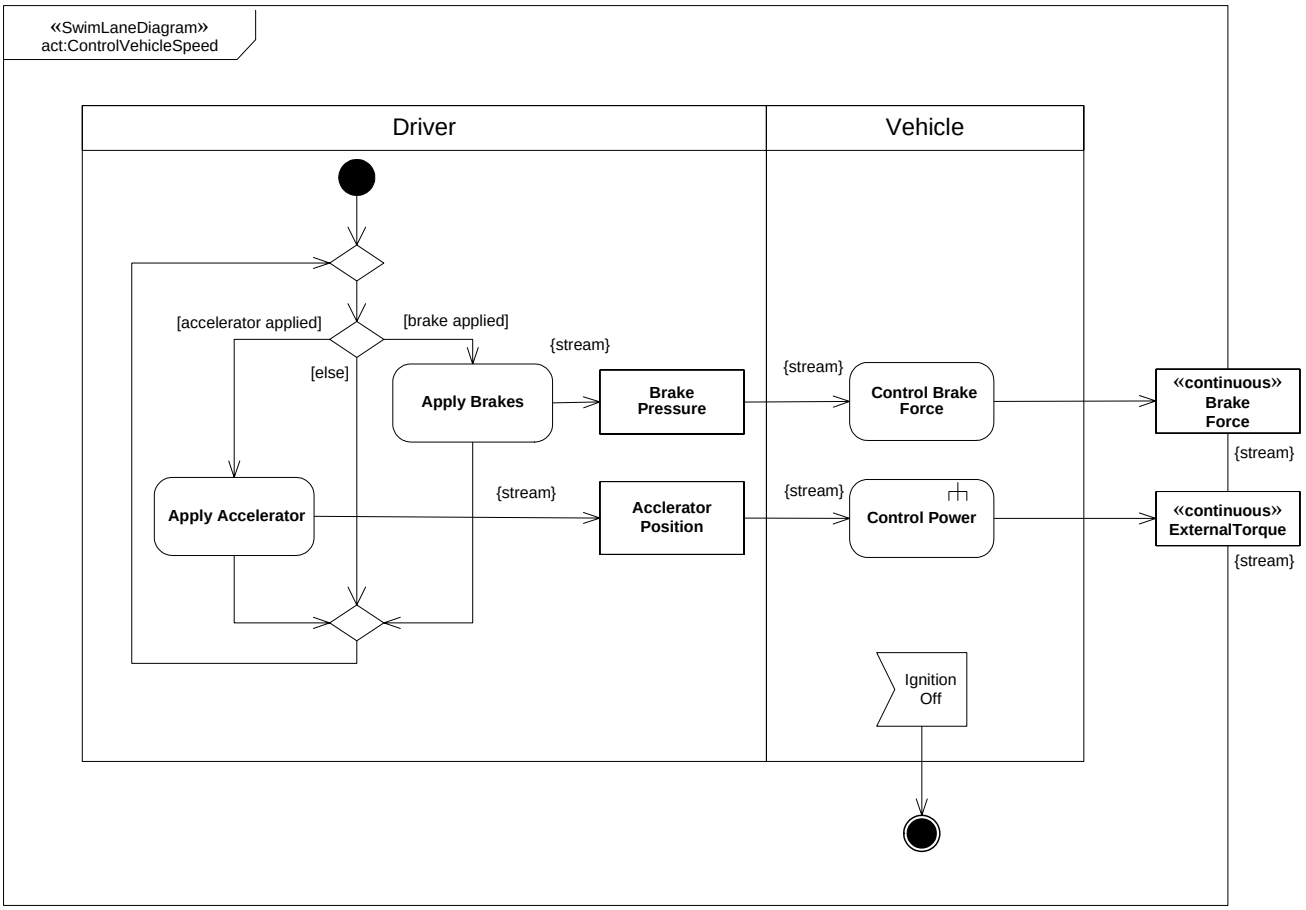
B.4.5 Interaction Overview Diagram for “Drive Vehicle”

The interaction overview diagram for “Drive Vehicle” depicts the the top level flow of control associated with the use cases in 0.4.4. An alternative representation would be to apply a restricted use of an activity diagram where the actions can invoke an interaction occurence as well as an activity. The interaction overview diagram is part of UML 2, but is not required for SysML (refer to Interactions Chapter). However, it is being included as an example of a UML 2 diagram used in conjunction with other SysML diagrams. The” Control Vehicle Speed” and “Start Vehicle” reference more detailed behaviors in 0.4.6 and 04.14 respectively.



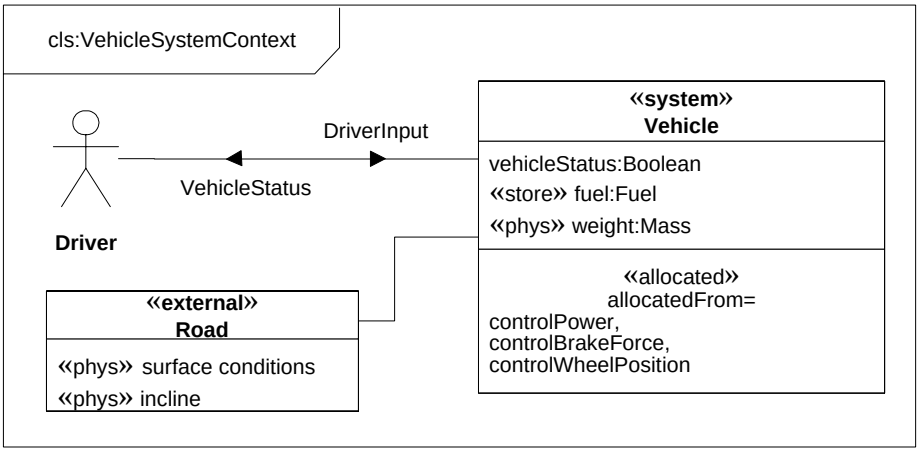
B.4.6 Swim Lane Diagram for “Control Vehicle Speed”

The swim lane diagram for “Control Vehicle Speed” is a usage of an activity diagram (refer to Diagram Appendix for information on diagram usages) that depicts the driver and vehicle as swim lanes. The swim lanes enclose the functions that are performed by the driver and vehicle to control the vehicle speed. This behavior is referenced in the interaction overview diagram in 0.4.5. The vehicle functions are represented as operations of the vehicle class in 0.4.7. The outputs from the driver actions (apply brakes and apply accelerator) are shown to produce streaming outputs that correspond to brake pressure and accelerator position respectively. Streaming inputs and outputs indicate that the function can accept the streaming inputs and produce streaming outputs while the function is executing. The brake pressure and accelrator position outputs are inputs to to the vehicle functions for controlling brake force and power respectively. The Brake Force and External Torque outputs from the vehicle are designated as continuous outputs. When an ignition off signal is received, the vehicle functions are disabled. The rake signal on control power indicates it is further decomposed as shown in (0.4.11).



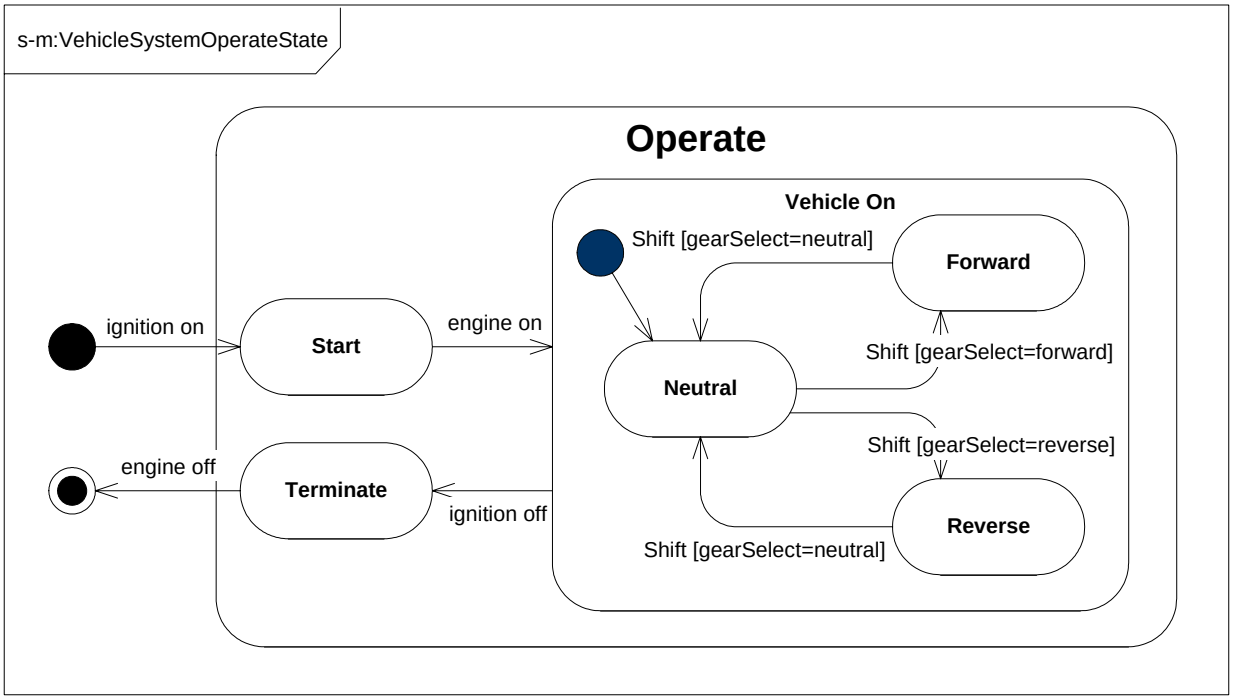
B.4.7 Class Diagram for the “Vehicle System Context”

The class diagram for the “Vehicle System Context” depicts the vehcile system as a black box that interacts with the driver and the road. The vehcile is stereotyped <<system>>, and contains features that represent physical characteristics (weight), physical stores (fuel), and data store (vehicleStatus). The fuel property is typed by a Fuel class that can include properties of its own such as octane level, cost per gallon, etc. Weight is typed by a quantity called mass that is defined in the auxiliary chapter and model library. Mass can include units and values, and be further specialized to include a probability distribution on its values. The Control Brake Force and Control Power functions have been allocated to the vehicle system as indicated by the functions in the vehicle class based on teh swim lane analysis in 0.4.6. This diagram also depicts the item flows that convey the input and outputs between the driver and the vehicle.



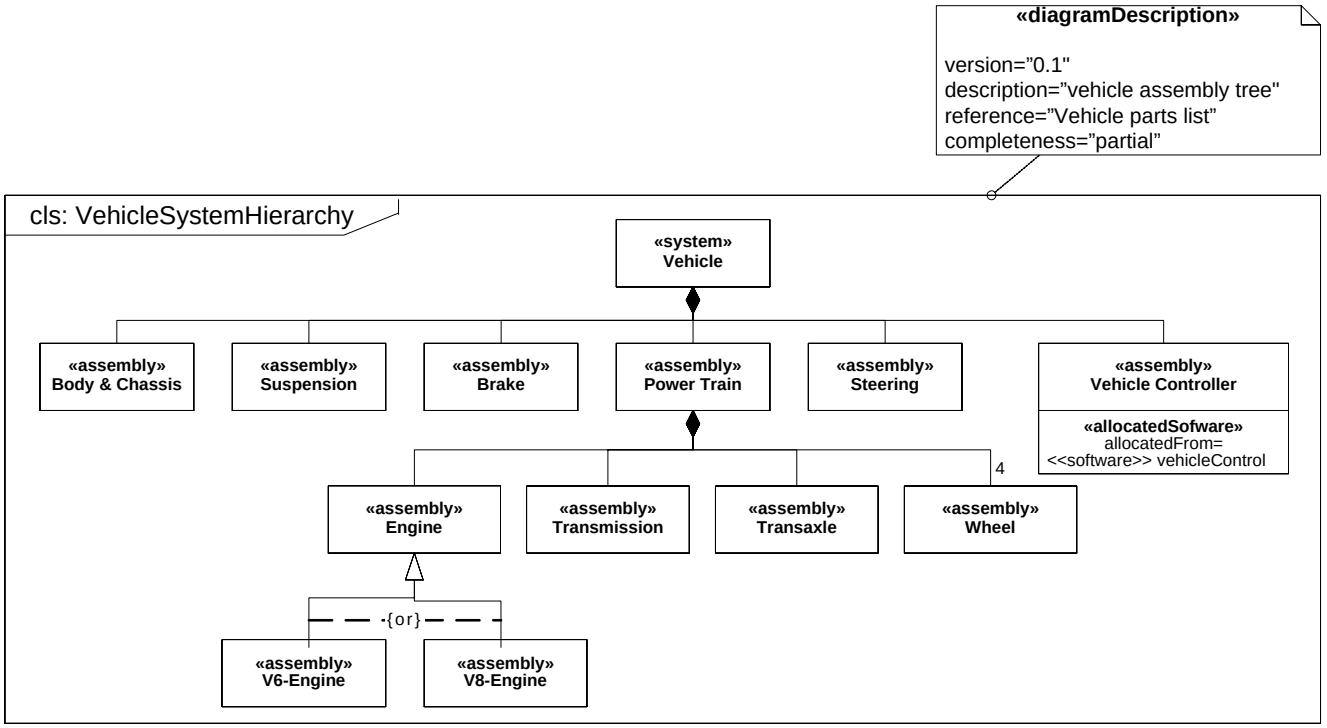
B.4.8 State Machine Diagram for the “Vehicle System Operate State”

The state machine diagram for the “Vehicle System Operate State” depicts the composite operate state and its substates. The events and guard conditions on the transitions are also shown.



B.4.9 Class Diagram for the “Vehicle System Hierarchy”

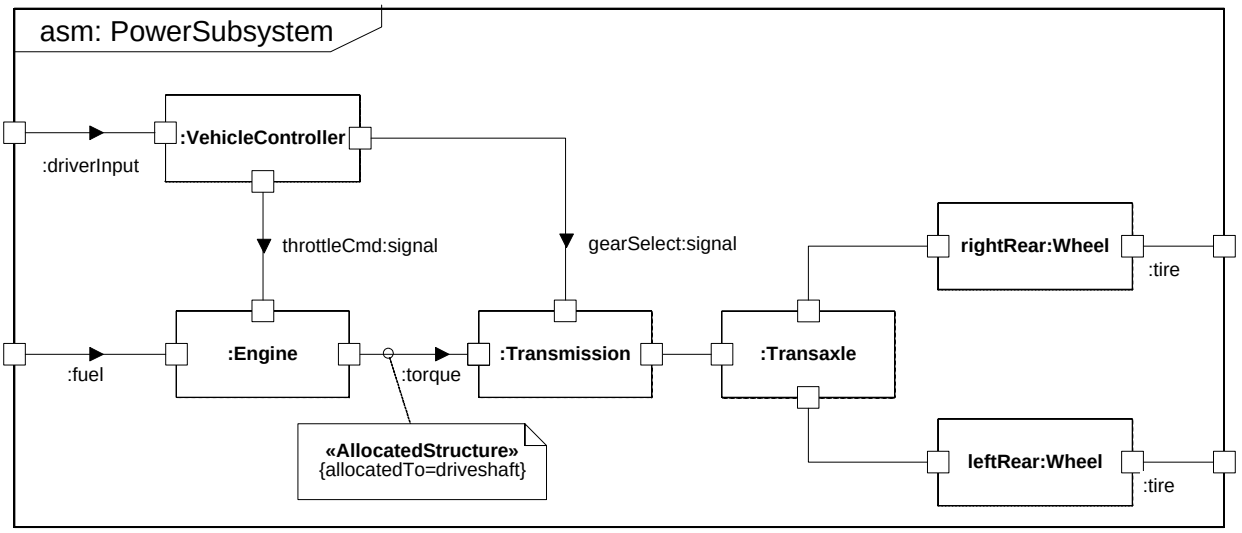
The class diagram for the “Vehicle System Hierarchy” highlights some of the assemblies that compose the vehicle system. The vehicle is a sterotype of system, which in-turn is a sublcass of the assembly stereotype. (Note: <<system>> is not a standard SysML stereotype). The power train is further decomposed into its assemblies. The engine includes one of two types of engine assemblies (a V6 or V8). In addition, the Vehicle Controller assembly is the execution platform for the vehicleControl software component as indicated by the <<allocatedSoftware>>. This represents a subtype of the allocation relationship applied between structural components of hardware and software as indicated in the allocation section of the Special Usages Appendix. The <<allocatedSofware>> and <<software>> stereotypes are not a standard SysML but can be subclassed from the SysML stereotypes for allocation and assembly resepectively. A part of the vehicleControl software component can be allocated if more granulatiry is required by designating the part as vehicleControl.part. The diagram description includes version information, description information, references the vehicle parts list, and states the assembly tree is only paritally complete. (Refer to Diagram Appendix A for explanation).



B.4.10 Assembly Diagram for the “Power Subsystem”

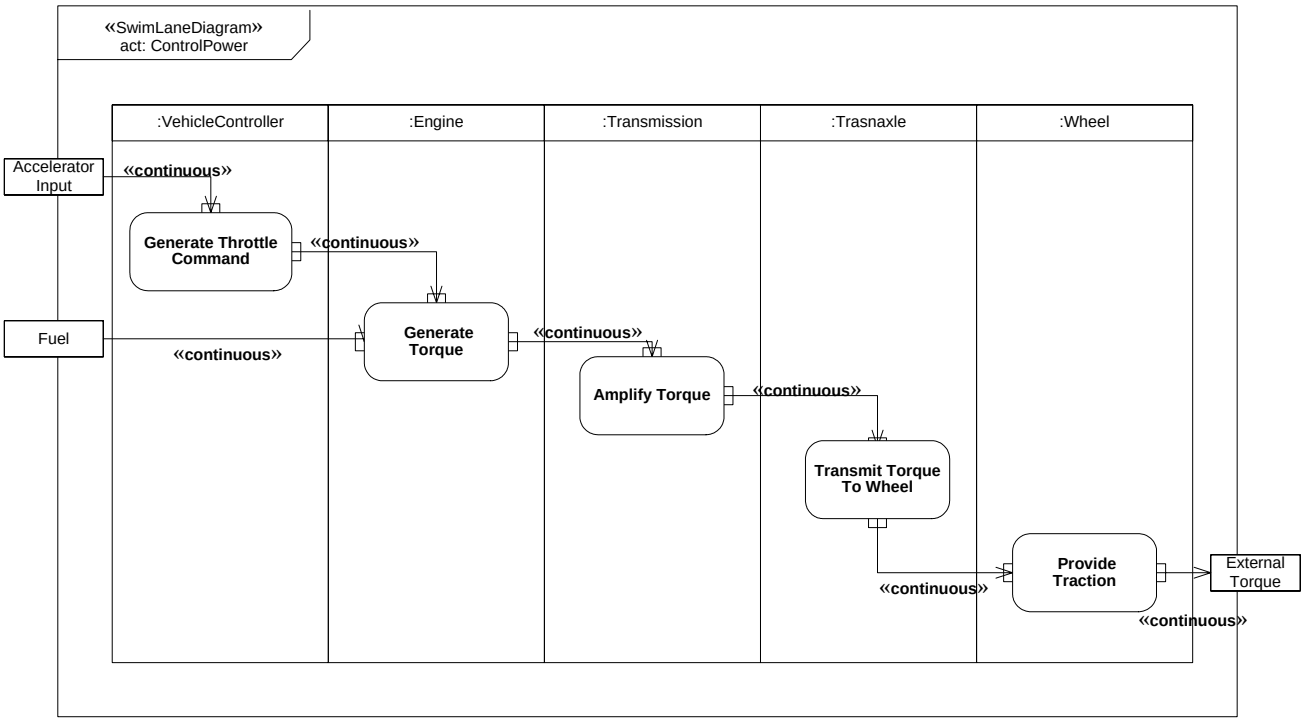
The assembly diagram for the “Power Subsystem” depicts the various parts of the vehicle that realize controlPower from the class diagram in 1.7. The assemblies (classes) that type the parts are indicated by the colon (:) notation in the Figure and are derived from the Vehicle System Hierarchy in 0.4.9. This distinction enables the same generic assembly class to be reused in many different contexts and still be uniquely identified for a particular use. The same instance of the parts, such as the vehicle controller, may play different roles in different subsystems. As a result, the parts are shared and not owned by the enclosing subsystem class, and are represented by dashed outlines instead of a solid line. The item flows between the parts are indicated by the solid arrowheads on the connectors. The ports on the wheels are shown to by typed by tire. The connector between the engine and transmission is shown to be allocated to a driveshaft assembly via the allocation relationship (refer to Allocation chapter and Special Usages Appendix).

NOTE TO THE EDITOR: The parts in the PowerSubsystem should be dashed lines, but this does not appear in this diagram.



B.4.11 Swim Lane Diagram for “Control Power”

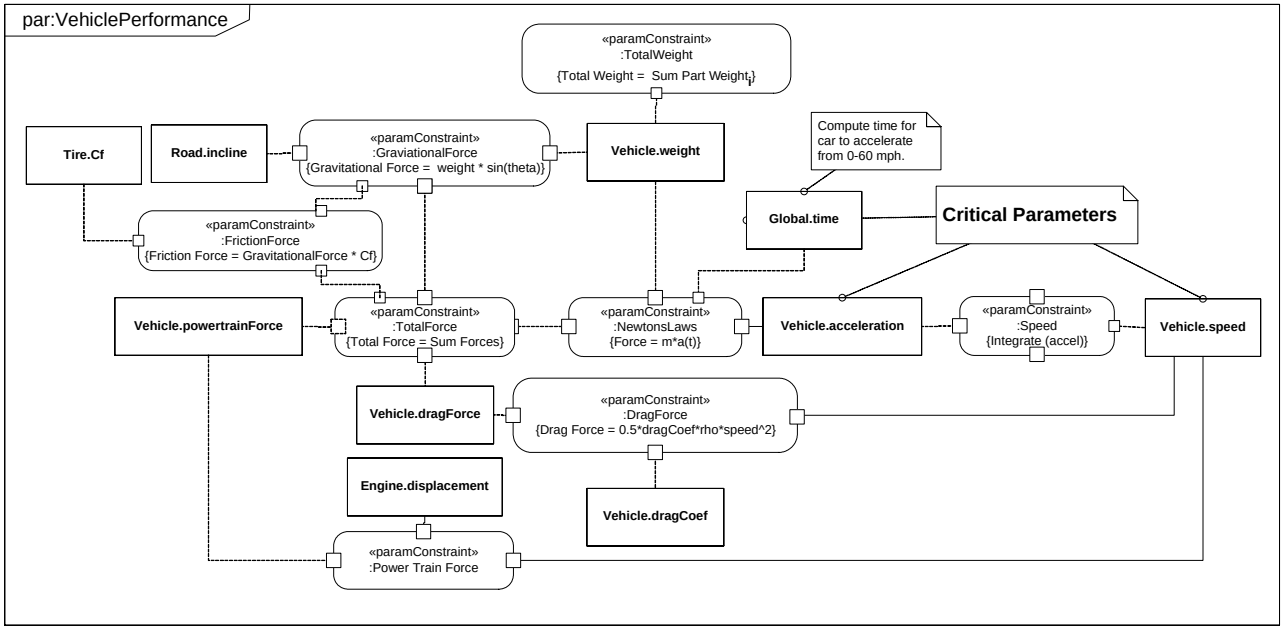
The swim lane diagram for “Control Power” depicts the the functions that are performed by the various parts of the system to control power. The inputs and outputs to the activity are indicated as well using the pin notation for the object nodes. The parts in the swim lane diagram are the parts from the Power Subystem in the assembly diagram in 0.4.10.



B.4.12 Parametric Diagram for “Vehicle Peformance”

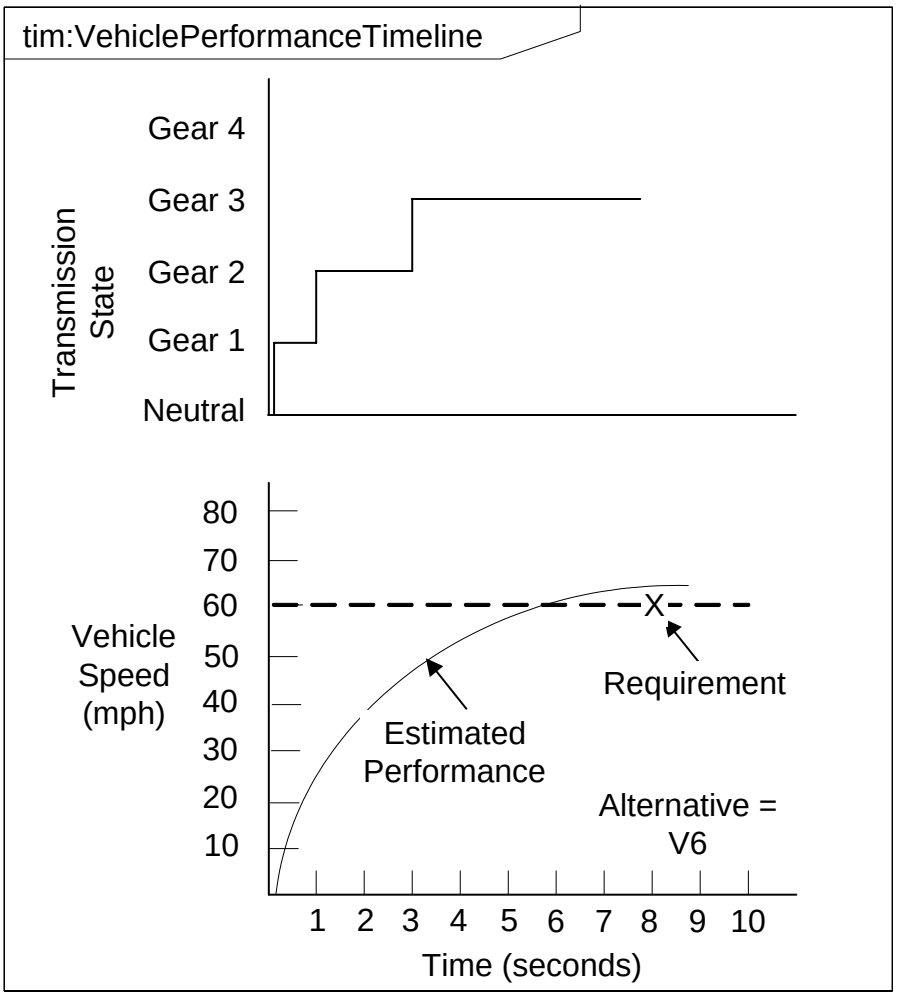
The parametric diagram for “Vehicle Performance” depicts a network of parametric constraints (equations) between properties that impact the critical performance parameters for vehicle acceleration and speed. One of the critical requirements from 0.4.3 was to accelerate from 0 - 60 miles per hour in less than 8 seconds. This diagram describes the parametric relationships between the properties that are associated with the different parts of the vehicle and enviornment. The sum of various types of forces, including gravitational, drag, friction, and powertrain are summed and integrated to determine vehcile acceleration and speed. Time is a property of a global clock that is implicitly used in all the equations. The definitions of the parametric constraints can be specified in a separate class diagram as described in the parametric chapter. The parametric diagram can be provided to the appropriate tool to support detailed performance analysis. The more abstract equations such as the power train equations can be defined in detail and executed in a simulation to determine whether the acceleration requirement is being satsified and/or to perform sensitivity analysis on the various system paraemters.

NOTE TO THE EDITOR: Lines should be dashed.



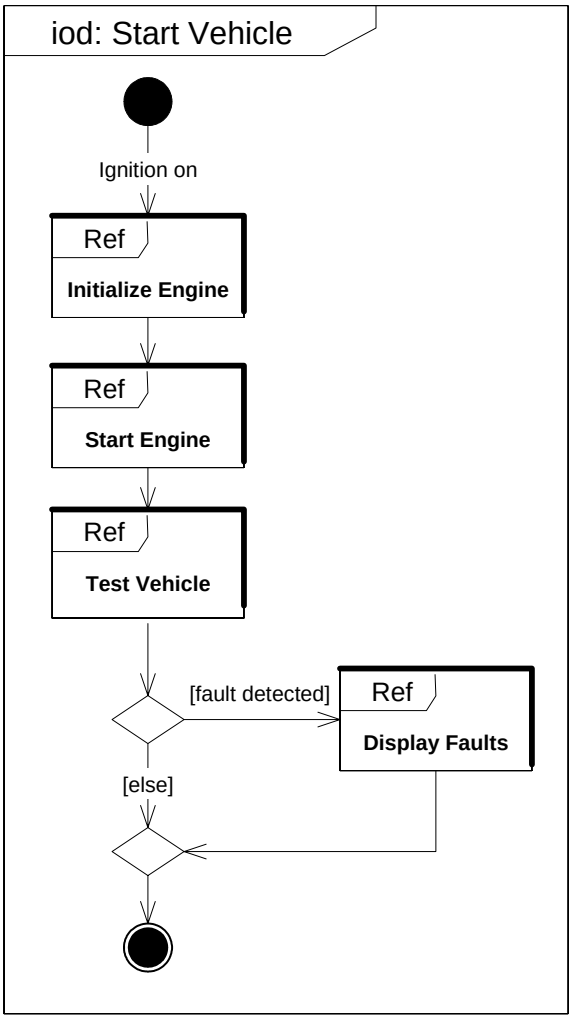
B.4.13 Timing Diagram for the “Vehicle Performance Timeline”

The timing diagram for the “Vehicle Performance Timeline” depicts the vehicle speed property from 1.12 as a function of time. Based on the analysis performed, the vehicle is able to satisfy its acceleration requirement in 0.4.3 using the V6 engine shown in 0.4.9. The vehicle operational state from the state machine diagram in 0.4.8 is also depicted as a function of time.



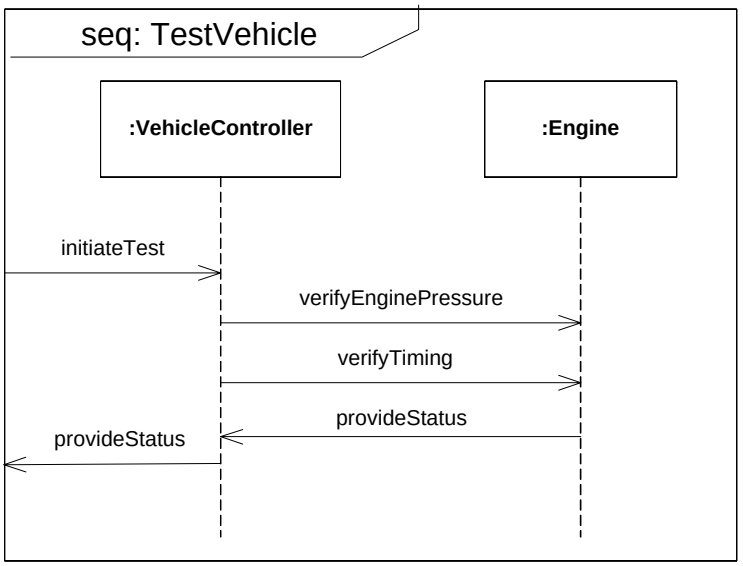
B.4.14 Interaction Overview Diagram for “Start Vehicle”

The interaction overview diagram for “Start Vehicle” depicts the behavior which was referenced in the higher level interaction overview diagram in 0.4.5. This diagram can be thought of as a restricted use of an activity diagram where the actions invoke an interaction occurence. As mentioned previously, this diagram is part of UML 2, but is not required for SysML (refer to Interactions Chapter). This diagram references more detailed behaviors. In particular, the Test Vehicle references the sequence diagram in 0.4.15.



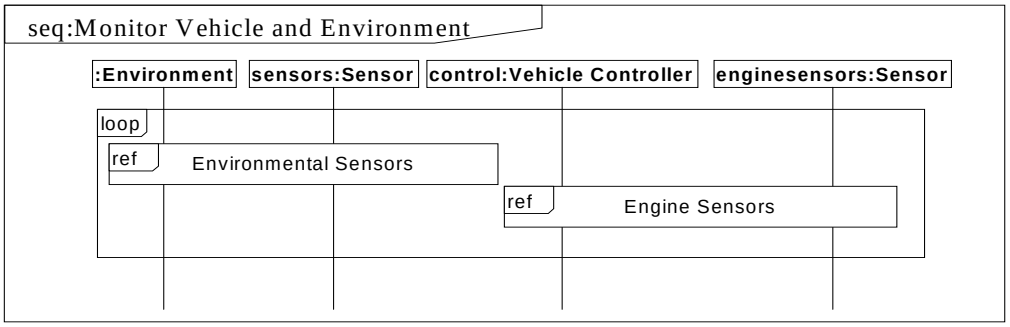
B.4.15 Sequence Diagram for “Test Vehicle”

The sequence diagram for “Test Vehicle” is referenced in the interaction overview diagram in 0.4.14, and depicts the sequence of message flows involved in testing the vehicle. The parts are typed by the assemblies in the system hierarchy in 0.4.9.

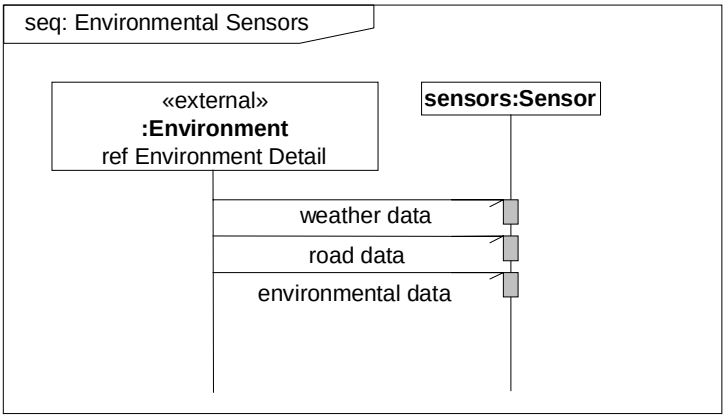


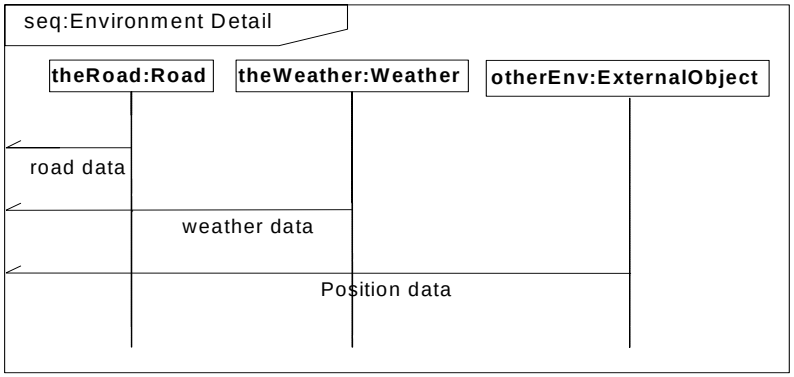
B.4.16 Sequence Diagram for “Monitor Vehicle and Environment”

In order to provide scaleable descriptions of use cases, interactions can be described in a number of diagrams that reference one another. For example, the Monitor Vehicle & Environment interaction from the interaction overview diagram in 0.4.5 can be shown as a sequence diagram. In this case the monitoring is continuously looping, and references further interactions, Environmental Sensors and Engine Sensors.



In the Environmental Sensors interaction, shown here in a referenced sequence diagram, the environment interacts with the vehicle sensors. The Environment includes a reference to Environment Detail which represents a decomposition of the Environment into its constituent parts as shown in the following sequence diagram.





Appendix C. Specialized Usages

C.1 Translating EFFBDs into Activity Diagrams

C.1.1 Overview

Enhanced Functional Flow Block Diagrams (EFFBD) are a kind of control and item/data flow diagram commonly used in the systems engineering community. They are similar to activity diagrams, and the SysML activity chapter gives extensions for EFFBDs that are a separate compliance point from the rest of activities. Even activities with EFFBD extensions differ from EFFBDs in ways that require translation:

1. Translation between EFFBD and UML terminology, where these are different (section C.1.2).
2. Most EFFBD notation is different in activity diagrams, however, the translation is one-to-one in most cases, and follows the terminology translation (section C.1.2). A common notation will facilitate communication between software and systems engineers.
3. Some EFFBD constructs require usage patterns for translation to activity diagrams (section C.1.3).

C.1.2 Teminology and notation

Most EFFBD terminology is different than Activity Diagrams, but most of the translation is one-to-one. The translation between EFFBD and Activity terms is given in Table 1, where these are not the same. Translation of EFFBD notation follows the translation of terms below.

Table 1. *EFFBD - UML Terminology mapping.*

EFFBD Term	*Activity Diagram Term*
External Input/Output	Activity Parameter Node
Item Flow	Object Flow
Item Node	Pin (Object Node notation)
Triggering Item Input	Input parameter with minimum multiplicity greater than zero. The SysML term is “required input”.
Nontriggering Item Input	Input parameter with minimum multiplicity equal to zero. The SysML term is “optional input”.
Select	Decision, Merge
Branch Annotation	Guard
Concurrency	Fork, Join
Multi-exit Function	Activity with output Parameter Sets
Completion Criteria	Postconditions on output Parameter Sets

In addition, systems engineers use different terminology than UML activities in some cases. Tool vendors may account for these differences, but they are not changes to the UML metamodel. The translation to systems engineering terms is given in Table 2, where these are different.

Table 2. *SysML - UML terminology mapping.*

SysML	*UML*
Function or Activity	Activity (a kind of Behavior)
Usage of a Function in an Activity	Action (usually CallBehaviorAction)
Item	Class (object or data), usually flows between system elements.
Input/Output Item	A Class used as the type of a parameter.

C.1.3 Examples

Some EFFBD constructs are modeled less compactly in activity diagrams, even with the EFFBD extensions. Rather than add more shorthands to activity diagrams, these constructs are translated to activities, sometimes by a one-to-many mapping. The execution semantics is the same. Table 3 summarizes these translations from EFFBD constructs to activity diagrams.

Table 3. *EFFBD - UML usage pattern.*

EFFBD Construct	*Activity Diagram Usage Pattern*
Multiple Edges from Item Node	Fork after ObjectNode.
Iteration, Loop	Usage pattern for Flow, Decision, Merge
Kill branch	Usage pattern for Fork, Join, Interruptible Region, and Join Specification.
Iteration/Loop Queuing	Usage pattern for Central Buffer Nodes.

1. When EFFBD item nodes have multiple item flows coming out of them, items leaving the node traverse all item flows at once, whereas for UML object nodes each object token can traverse only one outgoing edge. In UML, this EFFBD semantics requires a fork after an object node to explicitly copy the object token, as shown in Figure C-1.

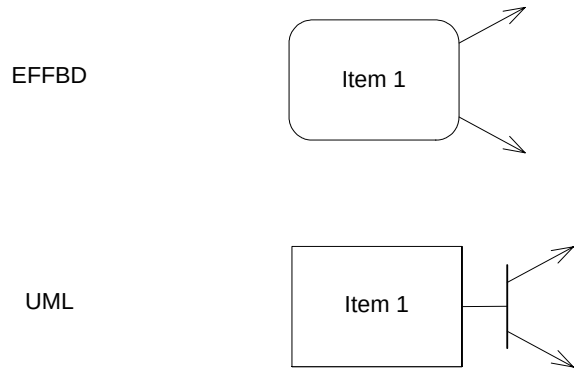


Figure C-1. Activity translation of EFFBD item node with multiple outgoing flows.

2. EFFBD uses a loop node to both start and end a loop, whereas UML uses a merge node to open a loop and a decision node to end a loop. Some implementations of EFFBD have a special node to exit a loop, where as UML does not. in UML, this EFFBD semantics is translated to the pattern shown at the bottom of Figure Figure C-2, using the EFFBD notational extension for loop nodes. A decision at any point can exit from the loop.

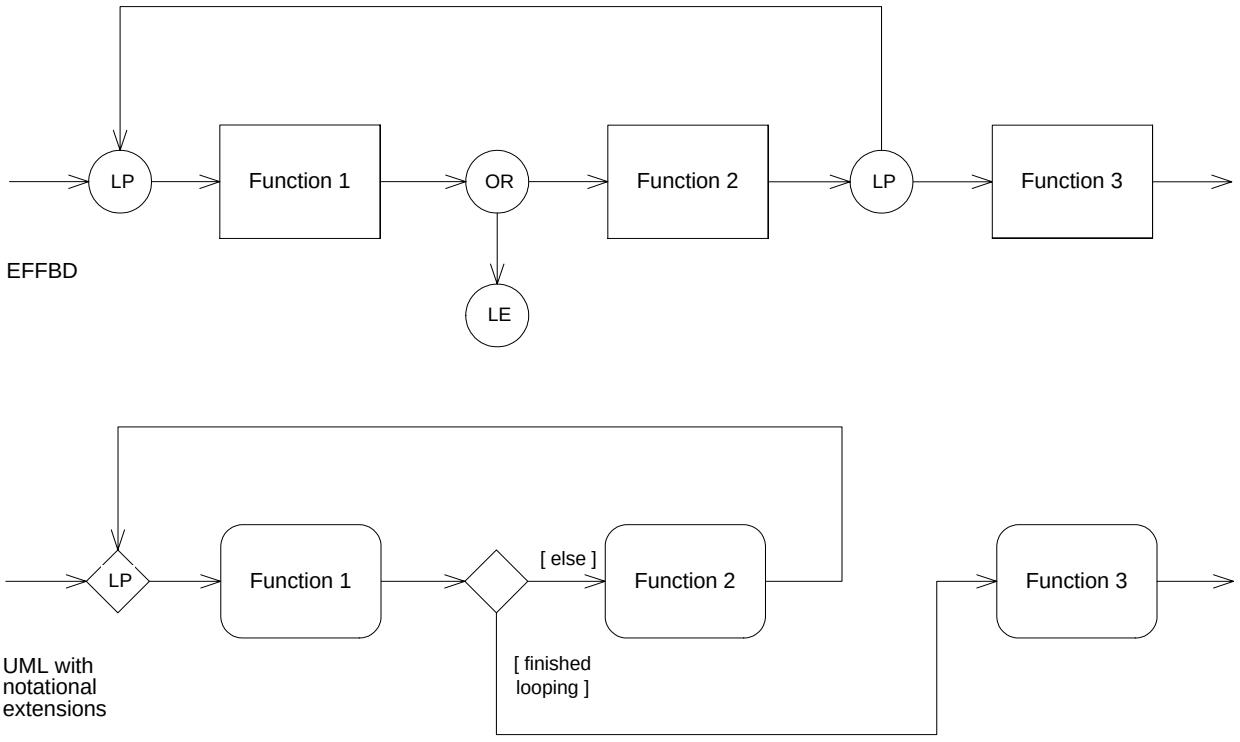


Figure C-2. Activity translation of EFFBD loop node.

3. EFFBD iteration nodes calculate the number of times to cycle only once, at the beginning of the first iteration, whereas UML LoopNode calculates at every cycle. In UML, this EFFBD semantics is translated to the pattern shown at the bottom of Figure Figure C-3, using the EFFBD notational extension for iteration nodes. A function is called to determine how many times to iterate, and this is checked by a decision node.

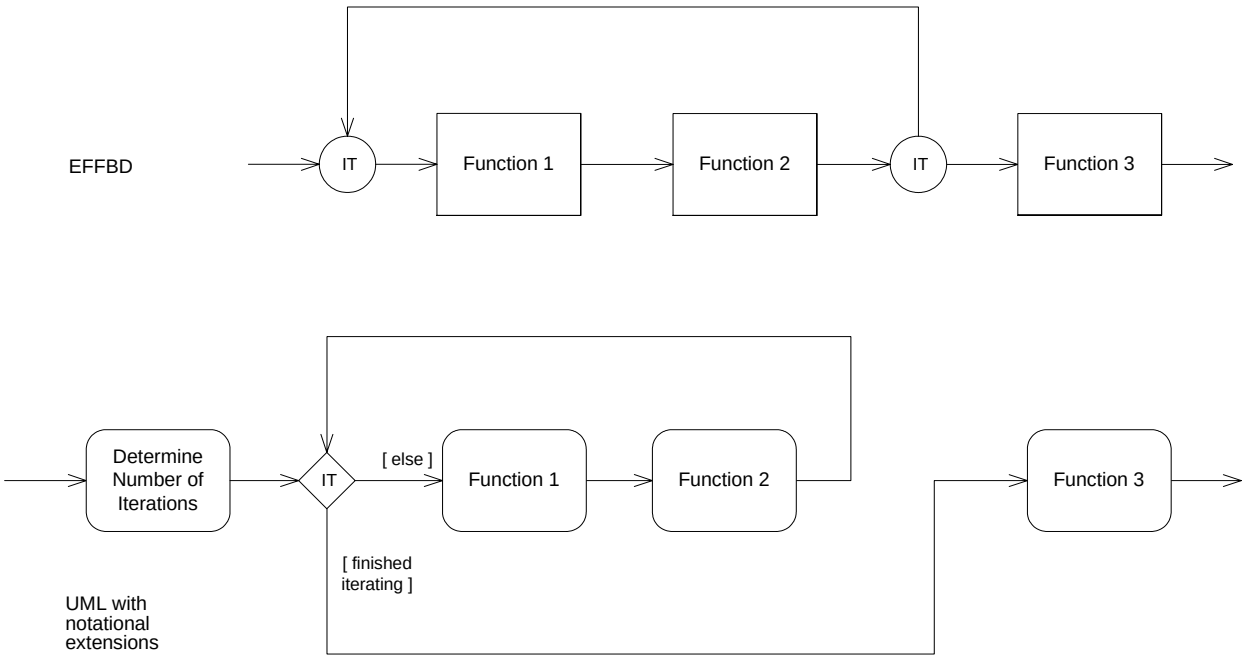


Figure C-3. Activity translation of EFFBD iteration node.

4. EFFBD kill branches come out of start-concurrency nodes to indicate that if the branch reaches the corresponding end-concurrency node before the other branches from the same start-concurrency, then the others are terminated. In UML, this EFFBD semantics is translated to the pattern shown at the bottom of Figure C-4. The dashed box indicates an interruptible region, and the edge labelled A is the interrupting edge. If edge A is traversed before Function 1 is complete, then Function 1 will be terminated, otherwise, both tines of the fork complete and are synchronized at the join. The join specification requires only that edge A be traversed, so will be satisfied whether edge A terminates Function 1 or not. If there

were a third tine that was a kill branch, with an edge named B leading to the join, then the join specification would be A or B.

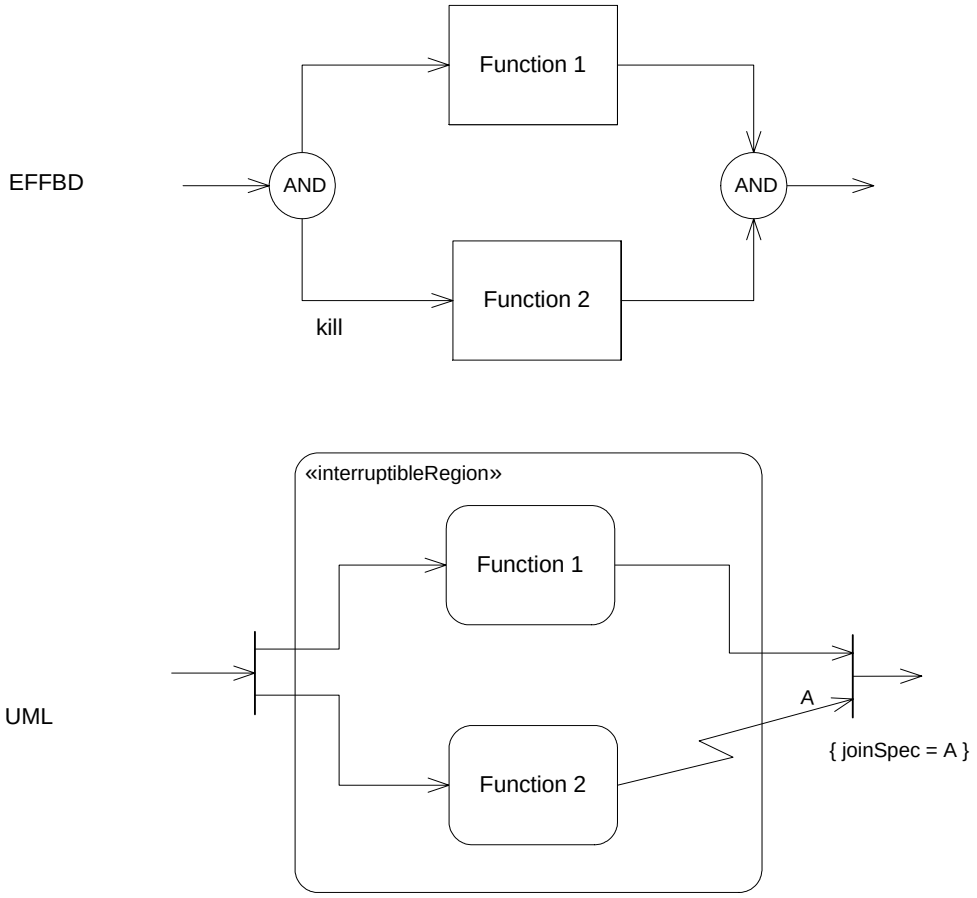


Figure C-4. Activity translation of EFFBD kill branch.

5. EFFBD iteration and loop nodes support queuing, whereas UML decision nodes do not. In UML, this EFFBD semantics requires a central buffer node before the iteration or loop node, as shown in Figure C-5.

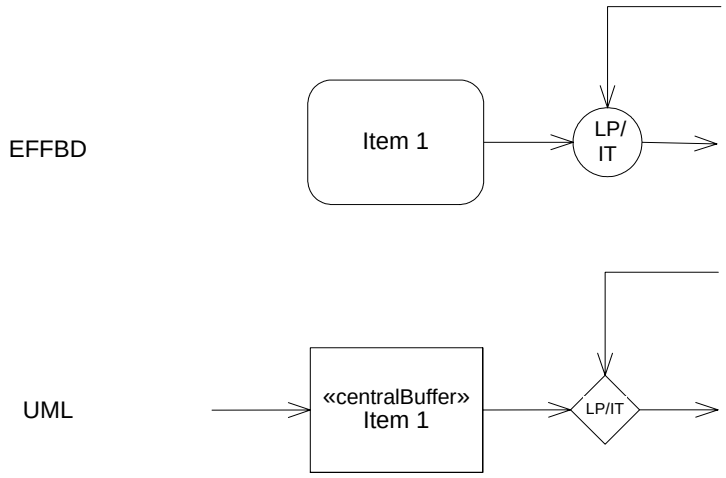


Figure C-5. Activity translation of EFFBD iteration/loop queuing.

C.2 Allocation Usages

C.2.1 Overview

This section provides additional discussion and clarification on the need for and use of allocation by systems engineering, and the support of these concepts in SysML. Various uses of the term “allocation” are discussed below.

C.2.2 Teminology and notation

C.2.2.1 Systems Engineering Uses of the term “Allocation”

"Requirement Allocation" is the allocation of requirements to a system design, implementation, deployment, or any other aspect of a system. "Requirement Allocation" also refers to the establishment of relationships between requirements. This kind of allocation is addressed by SysML::trace and SysML::satisfy relationships in Chapter 17 of this document.

"Functional Allocation" relates to the systems engineering concept segregating form from function. This concept requires independent models of "function" (behavior) and "form" (structure, object), 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.

“Flow Allocation” specifically maps flows in functional system representations to flows in structural system representations.

"Structural 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.

"Deployment Allocation" is the association, sometimes preliminary, of conceptual, abstract software assemblies with conceptual, abstract hardware assemblies. This is related to, but larger in scope than, the UML 2 concept of Deployment. UML2 allows only that SoftwareArtifacts be deployed to Nodes. In forming the system organizing concepts related to generalized systems including hardware, personnel, environment, and other elements, a more flexible mechanism for allocation of software to hardware is required. This is primarily because the form, or even existence, of software artifacts may not be understood at any given point in the system development, especially during the earlier phases.

"Property Allocation" focuses on allocation of system performance to various elements in the system model. Decomposing and allocating performance budgets, estimates, and measures of performance fall under this type of allocation. These relationships are addressed in Chapter 11 of this document.

The following table outlines the various relationships implied by the term “allocation” as typically employed by systems engineers. It also shows how these relationships are depicted in SysML. Note that Requirement Allocation and Property Decomposition/Allocation are not covered in this chapter, but the necessary relationships are described in other chapters.

Table 4 *Uses of the term “Allocation”, and corresponding relationships in SysML* **.**

Employment	*Relationship*	*From*	*To*	*Chap*
1. Requirement Allocation	*UML::trace* *SysML::satisfy*	*Requirement* *Packageable Element*	*Requirement* *Requirement*	*17*
2. Behavioral Allocation	*(see below)*			
2.1 Functional Allocation	*ownedBehavoir* *ownedOperation* *SysML::allocation*	*Behavior* *Operation* *Function (activity)*	*BehavioredClassifier* *Class* *Assembly/Class*	
2.2 Flow Allocation	*(see below)*			
2.2.1 Object Flow	*SysML::allocation* *UML::realizingConnector* *UML::realizingActivityEdge*	*ObjectNode* *Connector* *ObjectFlow*	*ItemProperty* *ItemFlow* *ItemFlow*	
2.2.2 Control Flow	*UML::realizingActivityEdge*	*ControlFlow*	*ItemFlow*	
2.2.3 Pin to Port	*(deferred)*	*Activity Pin*	*Port*	
2.3 Interaction Allocation	*UML::realizingMessage*	*Message*	*ItemFlow*	
3. Structural Allocation (e.g. Logical-Physical)	*(see below)*			
3.1 Assemblies	*SysML::allocation*	*Assembly (e.g. logical)*	*Assembly (e.g. physical)*	
3.2 Flows	*SysML::allocation*	*ItemFlow (e.g. logical)* *ItemProperty*	*ItemFlow (e.g. physical)* *ItemProperty*	
3.3 Connectors	*SysML::allocation*	*Connector (e.g. logical)*	*Assemblies and Connectors (e.g. physical)*	

Employment	*Relationship*	*From*	*To*	*Chap*
3.4 Ports (see 3.1)	*SysML::allocation*	*Port (e.g. logical)*	*Port (e.g. physical)*	
4. Deployment Allocation (e.g. Software to Hardware) (subset of Structural Allocation)	*SysML::allocation*	*Assembly (software)* *Connector (software)* *ItemFlow (software)* *ItemProperty (software)*	*Assembly (hardware)* *Connector (hardware)* *ItemFlow (hardware)* *ItemProperty (hardware)*	
5. Property Decomposition/ Allocation	*VariableBinding* *PropertyBinding*	*Property* *Property*	*ParametricRelation* *Element*	*11*

C.2.2.2 Allocation of Definition vs. Allocation of Usage

Allocation is used to relate model elements in different kinds of diagrams in a very general, flexible way. SysML supports allocation both of the generic definition of a model element, and of the specific usage of a model element (e.g. as a property (Part) of an assembly, or Action of an Activity). The definition of a model element (for example as expressed in a class, assembly, or action), establishes characteristics that the element will exhibit regardless of where it is used. The usage of a model element (for example as expressed in a property, part, or action), establishes how the model relates to other model elements for a specific purpose. Class diagrams depict the definition of model elements, where activity and assembly diagrams depict the usage of model elements. This is shown in the following figure.

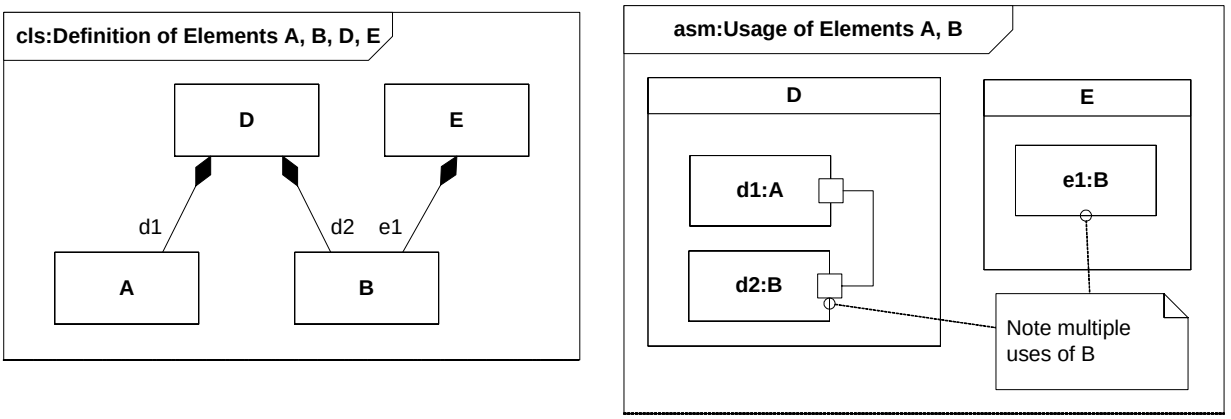


Figure C-6. Model Element Defintion expressed in Class Diagram, & Model Element Usage expressed in Assembly Diagram

Where diagrams express Use of model elements rather than their Description (as in Activity Diagrams and Assembly Diagrams), it is important to distinguish if the allocation depicted is simply allocation of usage, or if it implies allocation of definition.

Allocation relationships may be categorized based on the type of model elements that are allocated and may include:

1. Allocation of Usage (AoU): examples include Action to Part, Part to Part, ObjectNode to ItemProperty.
2. Allocation of Definition (AoD): examples include Activity to Assembly, Assembly to Assembly, Class to Class.
3. Assymetric Allocation, Definition to Usage (AADtU): examples include Activity to Part, Software Assembly to Hardware Processor.
4. Assymetric Allocation, Usage to Definition (AAUtD): examples include Action to Assembly

Each of these categories of Allocation relationships have potential usefulness to systems engineers. See section 16.6.1 for further discussion of how these categories of allocation may be employed. SysML will accommodate each of these categories, both in metamodel and in diagram notation. Specific attention was paid to making diagram elements distinctive for AoU, AoD, AADtU and AAUtD in each diagram type.The following table depicts various ways AoD, AoU, AADtU, and AAUtD might be employed.

Table 5 *Recommended employment of AoD, AoU, AADtU, AAUtD*

Allocation of Definition (AoD)	**Allocation of Usage (AoU)**	**Assymetric Allocation Definition to Usage (AADtU)**	**Assymetric Allocation Usage to Definition (AAUtD)**
E.g. Assembly to Assembly, or Activity to Assembly	E.g. Part to Part, Action to Part, Connector to Connector	E.g. Assembly to Part, Activity to Part	E.g. Part to Assembly, Action to Assembly,
Recommended employment (Table **22**): - Function to Assembly (2.1) - Assembly to Assembly (3.1)	Recommended employment (Table **22**): - ObjectFlow to ItemFlow (2.2.1) - Connector to ItemFlow (2.2.1) - ObjectNode to ItemProperty (2.2.1) - Connector to Connector/ Part (3.3) - Port to Port (3.4) - ItemFlow to ItemFlow (3.2) - SystemDeployment (Software Part to Hardware Part) (4)	Recommended employment (Table **22**) - TBD	Recommended employment (Table **22**): - (Software Assembly to Hardware Part) (4)

C.2.3 Examples

C.2.3.1 Diagrams for Allocation

Allocation relationships will typically be depicted on Class Diagrams, Assembly Diagrams, or Activity Diagrams. FunctionalAllocation and FlowAllocation may be depicted together using activity diagrams with activity partitions (aka swimlanes) and AlloctionReferences. It may also be shown in tabular views.

C.2.3.2 Subtypes of Allocation

Based on the tables above, the following subtypes of the allocation dependency may prove useful

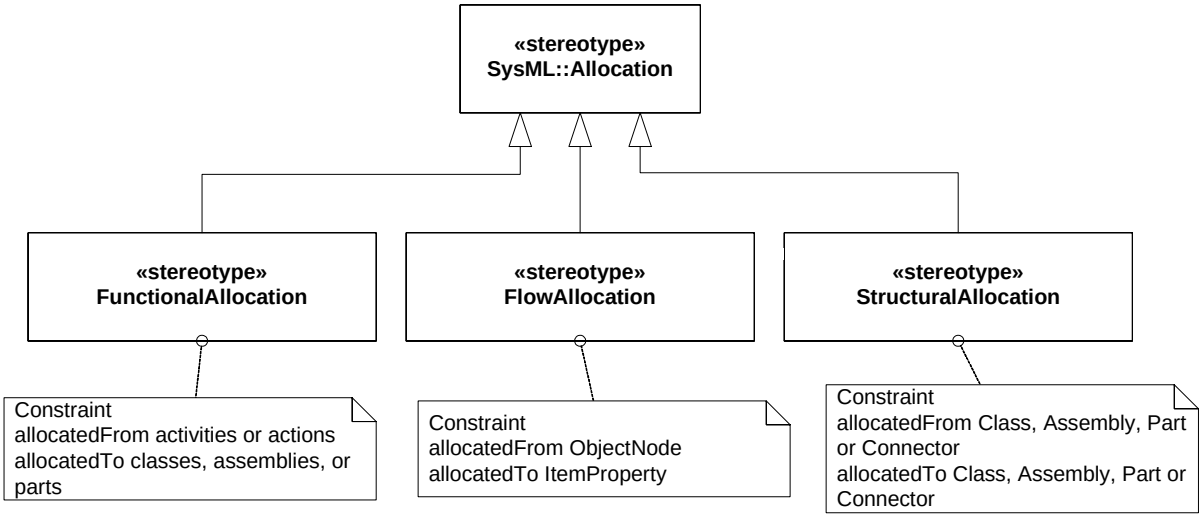


Figure C-7. Useful subtypes of the SysML::Allocation dependency

For each subtype of the SysML::Allocation dependency, the user model must include a corresponding subtype of the SysML::Allocated stereotype with a correstponding name, e.g. the FunctionalAllocation dependency must have a FunctionallyAllocated stereotype, which provides the population of /allocatedFrom and /allocatedTo properties for model elements which use the FunctionalAllocation dependency. It is possible for a given model element to be stereotyped by multiple derivative allocated stereotypes, if it participates in multiple types of Allocation dependencies. When depicted diagramatically, each type of allocation should appear in a separate compartment, as shown below.

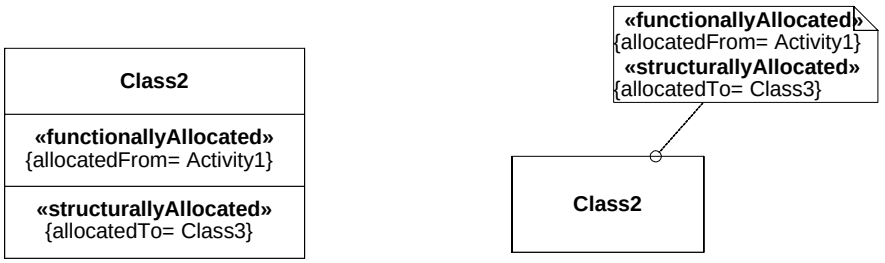


Figure C-8. Depiction of multiple allocation types for a single model element.

The user may further subtype allocation for special purposes. It is recommended that each subtype of allocation include the appropriate contstraints neccessary for that type.

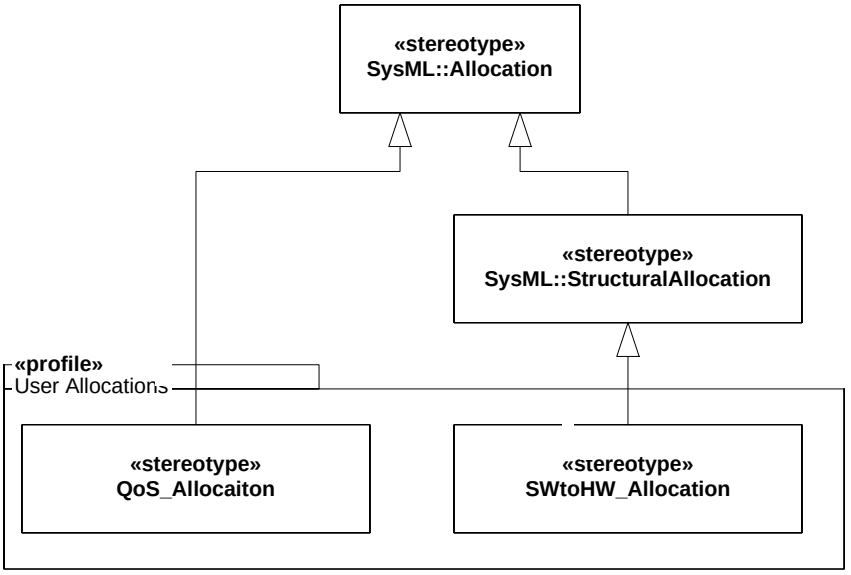


Figure C-9. Example of further subtypes of the Allocation dependency using a Profile

C.2.3.3 Requirement Allocation

The Allocation dependency does not refer to Requirements. See chapter 17 for Requirement dependencies.

C.2.3.4 Behavioral Allocation

Behavioral allocation relates purely behavioral model elements to structural model elements. Behavioral allocation is considered in three parts: allocation of Function (activity) to Assembly, allocation of ActivityEdge (object or control flow) to Connector, and corresponding allocation of activity Pin to assembly Port. Note that Control Flow allocation to Connector, and activity Pin allocation to assembly Port, are not defined in this specification and have been deferred until a future release.

Functional Allocation (Function to Assembly)

Example: consider the functions required to avoid wheel lockup when applying brakes in a car, and the system necessary to implement these functions:

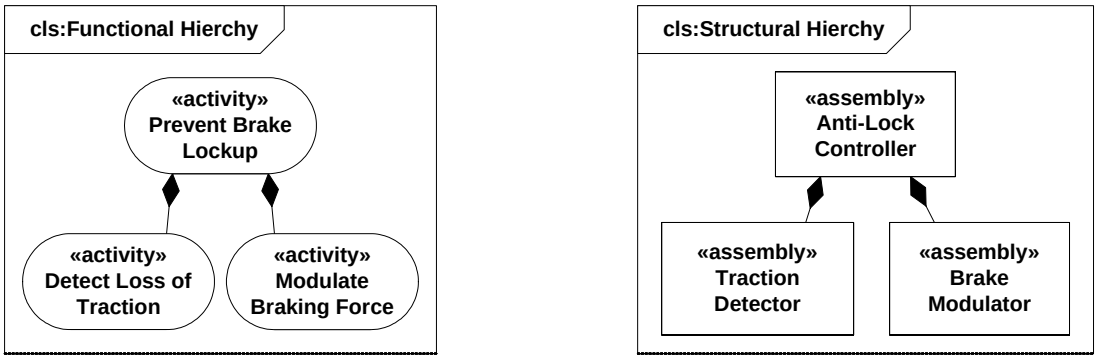


Figure C-10. Functional Hierchy & Structural Hierarchy

Consider how the fuctions should get allocated to the system assemblies:

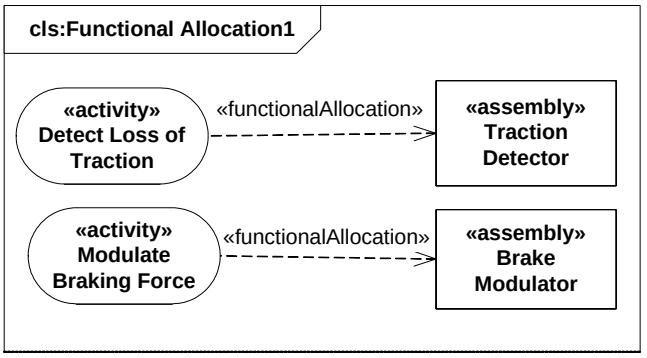


Figure C-11. Functional Allocation (AoD) on Class Diagram

Use of Allocation Compartments provides the following compact representation (Advanced Compliance):

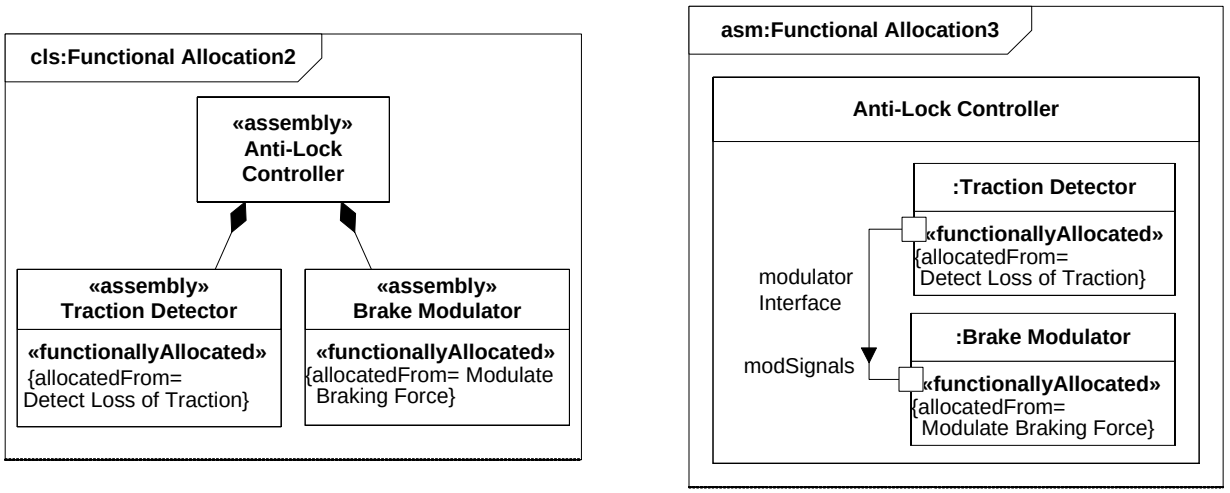


Figure C-12. Allocation Compartments on Class Diagram and Assembly Diagram

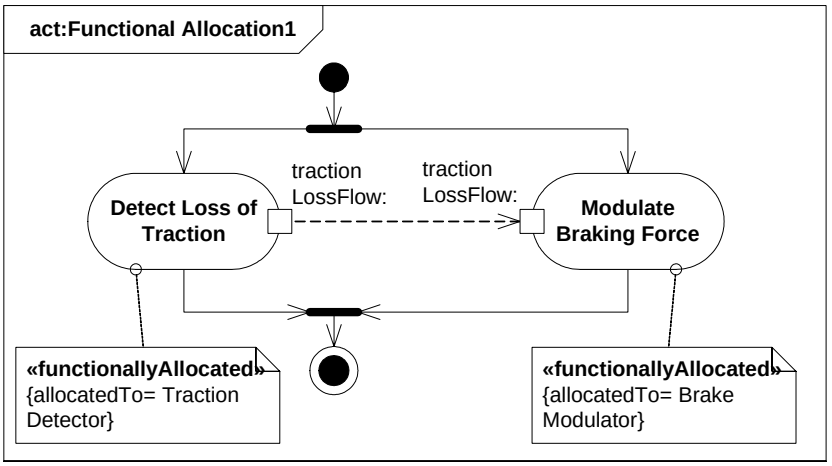


Figure C-13. Functional Allocation represented with Property Comments on Activity Diagram (AoD)

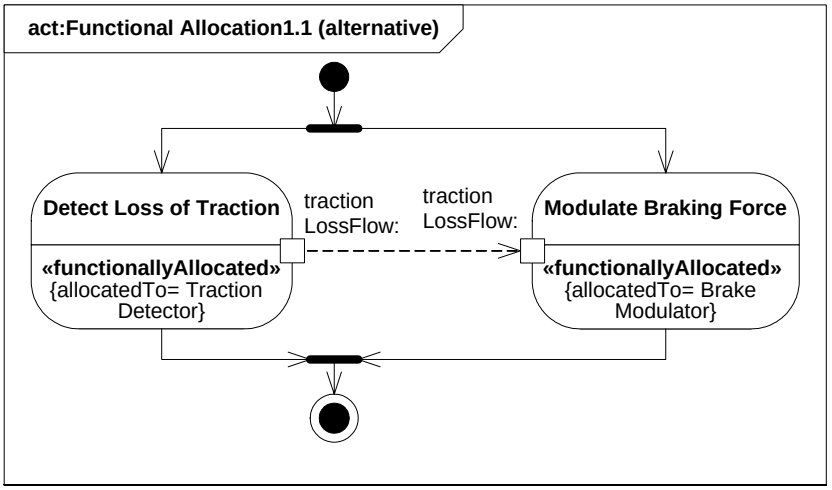


Figure C-14. Alternative representation of Functional Allocation using Property Compartments

ActivityPartition

An ActivityPartition (a.k.a. “swimlane”) in UML2 can be used to depict either 1) the ownedOperation relation between a CallOperationAction and a Class, or 2) the ownedBehavior relation between a CallBehaviorAction and a BehavioredClassifier.

System engineers may make use of 1) to serve the purpose of functional allocaitn, if the specific method used accommodates an Operation-based approach to describing functionality. The drawback of this approach is that it supports only Allocation of Definition, not Allocation of Usage. If specific usages of Activities or Classifiers need to be considered, this approach will be inadequate.

Systems engineers can not make use of 2) to serve as functional allocation, because ownedBehavior is an exclusive namespace relation, and a function (Activity) can be allocated to only one BehavioredClassifier. This does not accommodate reuse of functions.

To accommodate the full scope of functional allocation, the Allocation stereotype of Activity Partition has been proposed. An example of its use is provided below. The result of this diagram is that an Allocation dependency will exist between Detect Loss of Traction (client) and Traction Detector (supplier), and another will exist between Modulate Braking Force (client) and Brake Modulator (supplier).

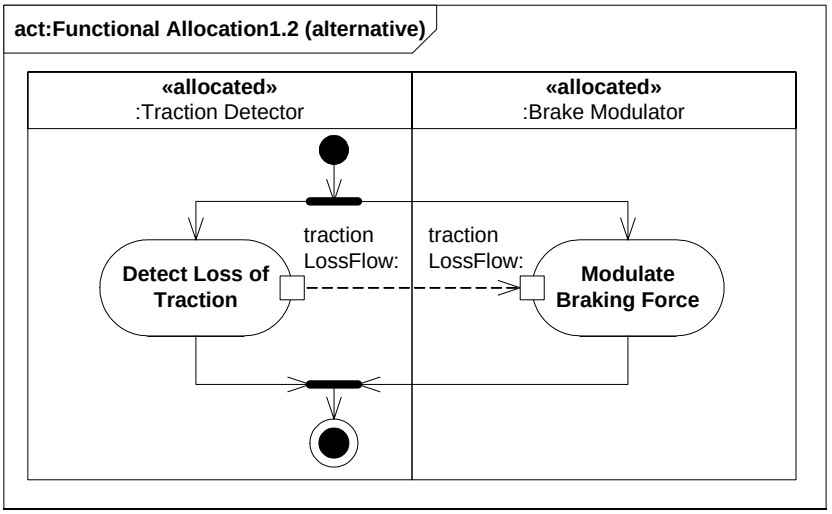


Figure C-15. Example of Optional User Representation: Functional Allocation represented via stereotyped Activity Partitions

Flow Allocation

Flow between activities can either be control or object flow. The following figures 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 depicted using a «realizingActivityEdge» dependency with ItemFlow.

UML2 provides concrete syntax for InformationFlows (supertype of SysML ItemFlows) to be explictly associated with ActivityEdges (representing ControlFlow or ObjectFlow) via the «realizingActivityEdge» dependency. UML2 also provides a concrete syntax for InformationFlows to be explicitly associated with Connectors via the «realizingConnector» dependency. Thus, through these ItemF.low related dependencies, ControlFlow and ObjectFlow may be related to Connectors. To this, SysML provides the additional capability of allocating ObjectNodes directly to ItemProperties (Classifiers having a UML2 «itemProperty» dependency with ItemFlow). It is also recommeded (but not required) that when an ObjectNode is allocated to an ItemProperty, both should be typed by a common Classifier having a «conveyedClassifier» dependency with the ItemFlow.

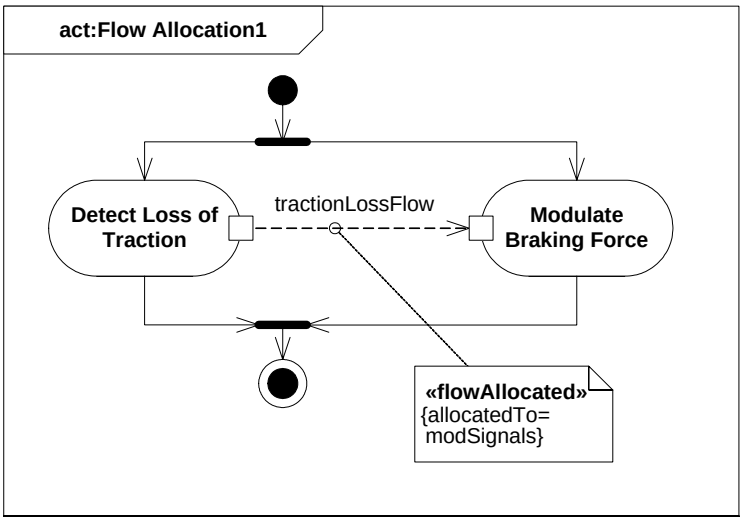


Figure C-16. Allocation of ObjectFlow to ItemFlow - Activity diagram

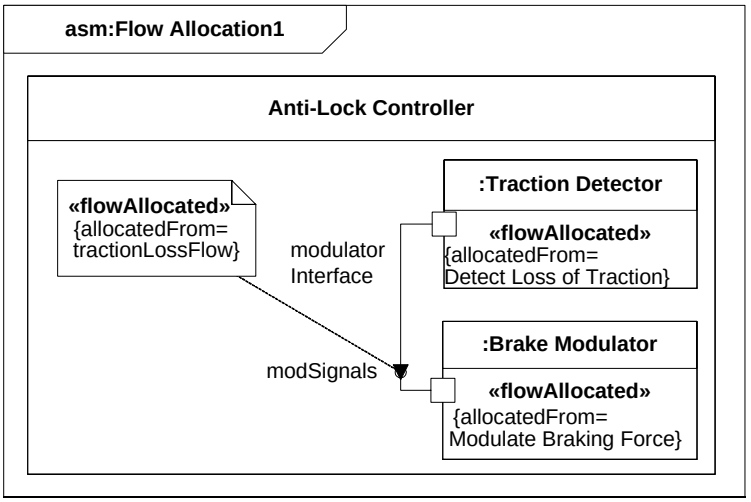


Figure C-17. AlloctionReference for ObjectFlow to ItemFlow - Assembly diagram

Note that allocation of ObjectFlow to Connector is an Allocation of Usage, and does NOT imply any relation between any defining class of ObjectFlows and any defining class of Connectors.

Allocation of ObjectNode to ItemProperty

The following figures illustrate an available mechanism for relating the objectNode from an activity diagram to the itemFlow on an Assembly diagram. SysML::itemFlow is discussed in Chapter 18 AuxiliaryConstructs.

Common typing of objectNode and itemFlow is a very valuable concept, and recognizes that it is possible to represent the same flow of information, energy, or matter in both activity and assembly models.

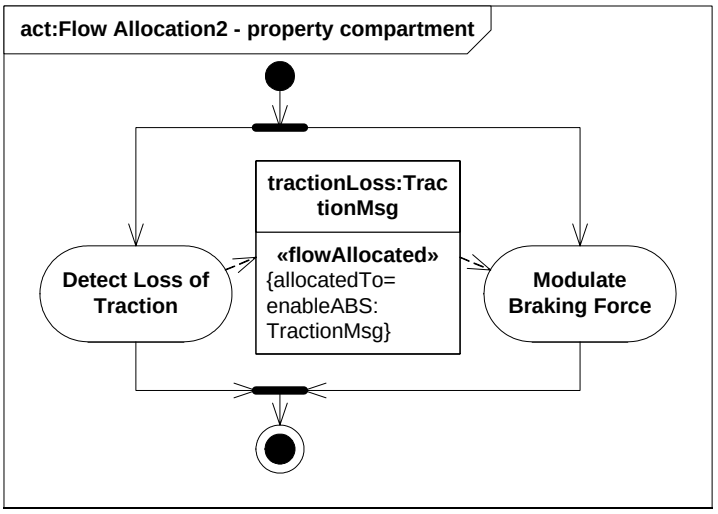


Figure C-18. ObjectNode allocatd to ItemProperty (AoU) - Activity diagram

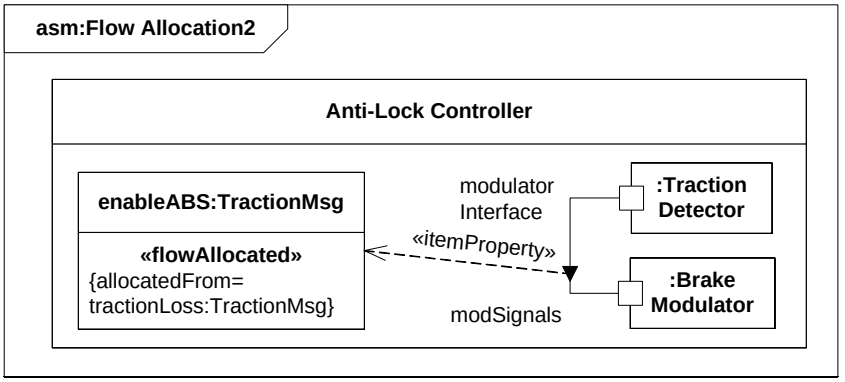


Figure C-19. ObjectNode allocated to ItemProperty (AoU) - Assembly diagram.

Pin to Port Allocation

Pin to Port allocation is not addressed in this release of SysML.

C.2.3.5 Assembly to Assembly Allocation

This document leaves it to the user to define the criteria for “logical” and “physical” structure. It is assumed that, in the general sense, both logical and physical representations will be depicted as assemblies, complete with ports, parts, and connectors. This section simply provides an example of how SysML supports allocation between these two kinds of structures.

The «allocatedTo» relation is used to relate logical assemblies to physcial assemblies, and logical connectors to physical connectors. Physical assemblies which have logical assemblies allocated to them may be annotated using AlloctionReferences, or AllocationCompartments. These are illustrated in the following figures.

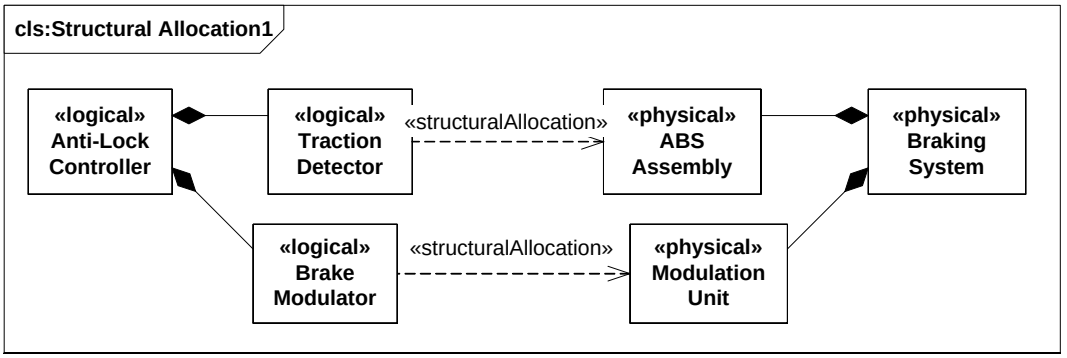


Figure C-20. Allocation of Assembly-to-Assembly on Class Diagram (Allocation of Definition)

Note that in this example, the abstract nature of the logical connector ":modulator interface", when being allocated to a concrete physical architecture, must be allocated to two physical connectors ":J57 Cktr" and ":J27 Cktr", as well as to the physical part ":Cable A".

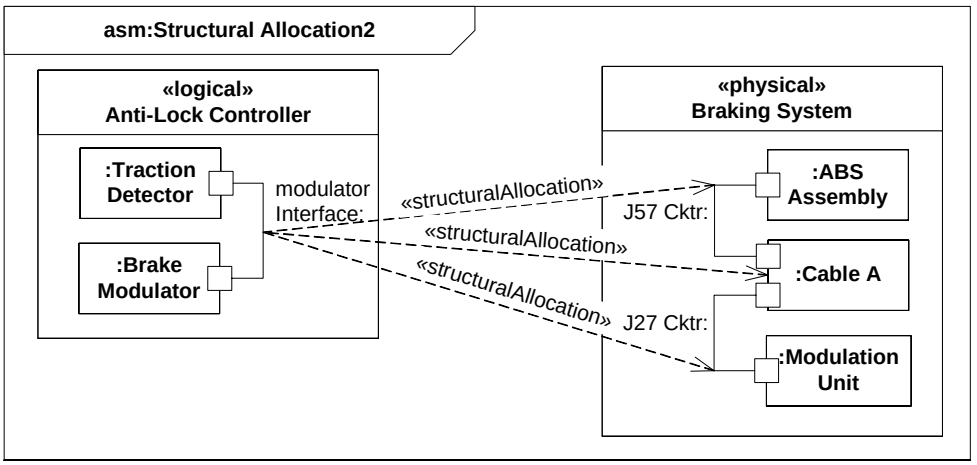


Figure C-21. Allocation of Connector-to-Connector/Assembly on Assembly Diagram (Allocation of Usage)

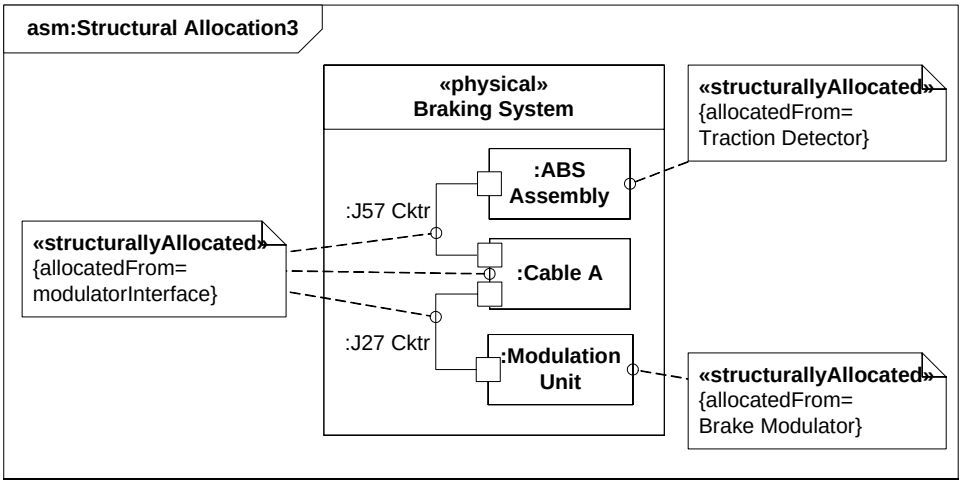


Figure C-22. Depiction of Assembly-to-Assembly and Connector Allocation using AllocationReferences on Assembly Diagram.

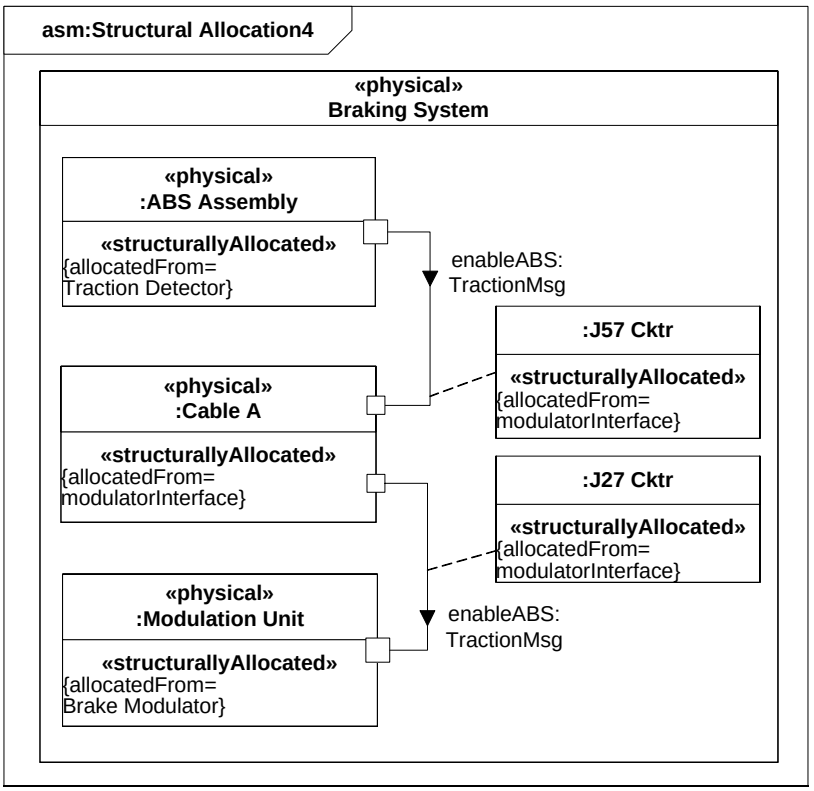


Figure C-23. Use of Allocation Compartments to Depict Assembly-to-Assembly Allocation, and Common Typing of Items to reconcile flows.

C.2.3.6 Tabular Representation of Allocation

Some typical allocation tables may include:

Functional Allocation Table from function (Activity) to structure or (Class or Assembly)

Flow Allocation Table from ActivityEdge to ItemFlow (realizingActivityEdge association), ObjectNode to ItemProperty, ItemProperty to ItemFlow (itemProperty association), and ItemFlow to Connector (realizingConnector association).

Structural Allocation Table from one structure (e.g. “logical” Assemblies, Parts, or Connectors) to another structure (e.g. “physical” Assemblies, Parts, or Connectors).

Deployment Allocation Table from deployment sources such as software to deployment targets such as hardware.

Table 6 Example Allocation Table

ID	Allocation Type (User Extension)	ElementType AllocatedFrom	ElementName AllocatedFrom	ElementType AllocatedTo	ElementName AllocatedTo
1	Functional Allocation	Activity	Detect Loss of Traction	Assembly	Traction Detector
2	Functional Allocation	Activity	Modulate Braking Force	Assembly	Brake Modulator
3	Flow Allocation	ObjectFlow	tractionLossFlow	Connector	modulator interface
4	Flow Allocation	ObjectFlow	tractionLossFlow	ItemFlow	modSignals
5	Flow Allocation	ObjectNode	tractionLoss:TractionMsg	ItemProperty	enableABS:TractionMsg
6	Structural Allocation	«logical» Assembly	Traction Detector	«physical» Assembly	ABS Assembly
7	Structural Allocation	«logical» Assembly	Brake Modulator	«physical» Assembly	Modulation Unit
8	Structural Allocation	«logical» Connector	modulator interface	«physical» Connector	J57 Cktr
8	Structural Allocation	«logical» Connector	modulator interface	«physical» Connector	J27 Cktr
8	Structural Allocation	«logical» Connector	modulator interface	«physical» Assembly	Cable A

C.3 Provided and Required Interfaces

C.3.1 Overview

This section describes an architecture modeling approach that supports the design of service request driven systems that use provided and required interfaces. The main characteristic of this approach is that the internode and intranode communication is based on message exchanges (service requests) rather than on the definition of data/control flows. The approach is UML-based

and uses the UML 2.0 notation for provided and required interfaces. In a modeling environment that supports both UML 2.0 and SysML, these elements can be used in combination with those of SysML, even though SysML currently does not include provided and required interfaces in its compliance requirements for SysML Assemblies.

C.3.2 Principles

In the outlined approach, the system structure is described by means of composite structure diagrams, using blocks (in SysML, assemblies and their parts) as basic structure elements and ports as “named connection points.”

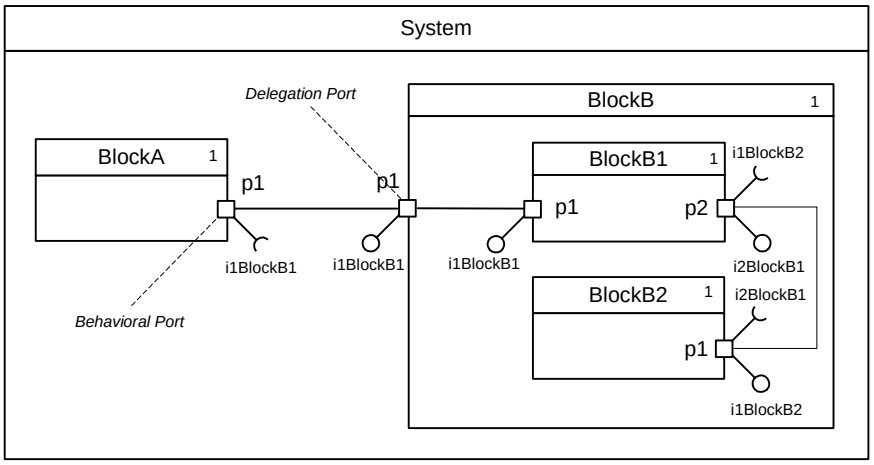


Figure C-24. Elements of a composite structure diagram.

There are two different kinds of ports (Figure C-24). Delegation or relay ports forward requests to other ports. Behavioral ports are parts of the block that actually implements the service.

Each port can have provided and required interfaces. A provided interface (denoted by a lollipop symbol) specifies a set of messages received at that port from elements outside the current block. A required interface (denoted by a socket symbol) specifies a set of message sent from that port to elements outside the current block. Thus, by characterizing an interface as required or provided the direction of the constituent messages at the port is defined. With regard to the naming of interfaces the following convention will be used in this document:

i<ServiceProviderInterfaceNumber><ServiceProvider>

The mechanics of the service request-driven system modeling approach is shown in Figure C-25. The message exchange between two blocks is visualized in the upper part of Figure C-25 by means of a sequence diagram. Service requests are sent as events, and the actual provisions of those services are shown as reflexive operations ("messages to self") at the lifeline of the receiving block. Both the service request messages and the associated service operations may have parameters. Typically, parameters are added at a later stage, when details with regard to the respective services are known.

The resulting structure diagram is shown below in a sequence diagram. In the operation compartment of each block, the received messages are listed as public (+) and the provided services are listed as private (-). Both blocks have a provided and required interfaces.

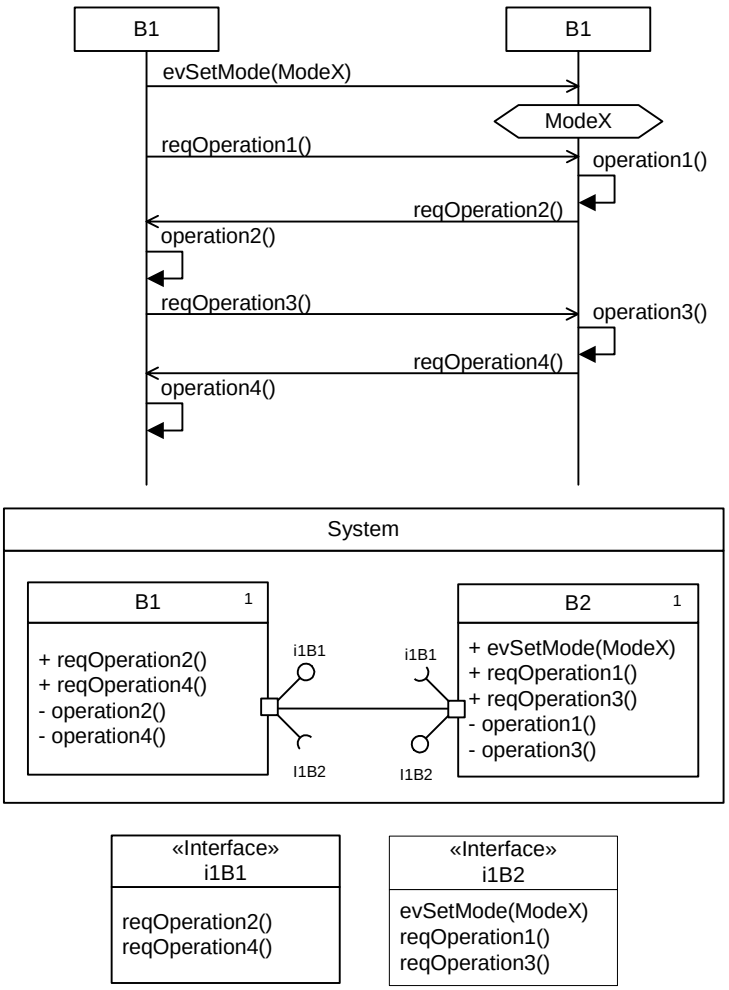


Figure C-25. Service Request-Driven System Modeling Approach.

C.3.3 Relation to DoDAF Views

The benefits of a service request-driven system modeling approach that uses provided and required interface becomes obvious when specifying complex, network-centric, system-of-systems architectures. Military-defense system specifications frequently must comply with the DoDAF (Department of Defense Architectural Framework) standard. Figure C-26 shows how

provided and required interfaces may be used to define a systems-of-systems architecture for a DoDAF-compliant specification.

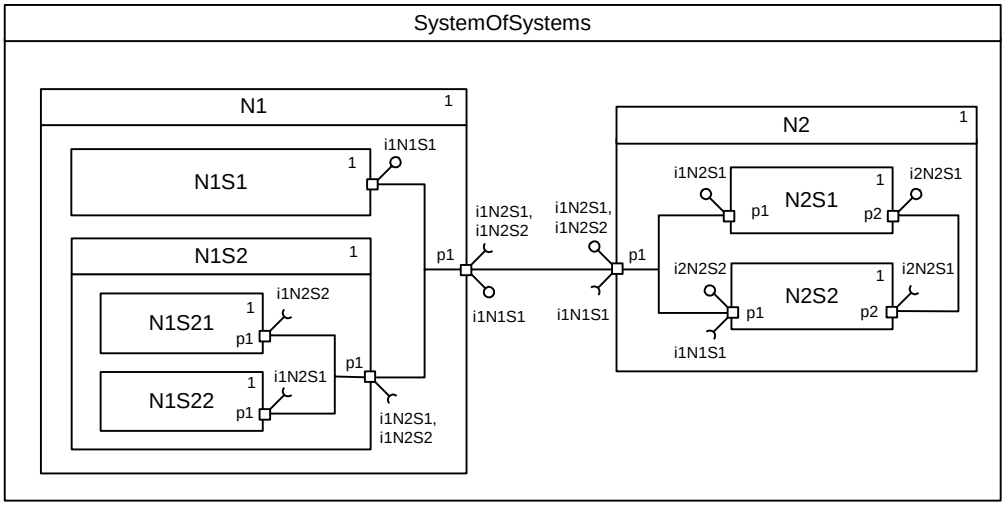


Figure C-26. Generic system-of-systems example.

The network specified consists of two nodes (N1, N2). Each node is decomposed into system components (N1 -> N1S1, N1S2; N2 -> N2S1, N2S2). Additionally, the system component N1S2 is further decomposed into the subcomponents N1S21 and N1S22. The different parts are linked via ports. Delegation (relay) ports are port p1 of system component N1S1, port p1 of node N1, and port p1 of node N2. All other ports are behavioral. For each port, the associated required and/or provided interfaces are shown. In this example, it is not relevant what the individual sent and received messages actually are.

The internodal operational information exchange matrix (DoDAF OV-3 view), as well as the internodal/intranodal, system-to-system interface description (DoDAF SV-1, SV-6 view) can be derived directly from the structure diagram. Typically, the information is documented by means of an N-Square (N2) diagram. In an N2 diagram the network nodes are arranged along the vertical and horizontal axis. The node interfaces then are listed in the intersection compartment of respective columns/rows. Table 7 and Table 8 show the N2 diagrams for the generic system-of-systems example.

Table 7. *Internodal Operational Information Exchange Matrix.*

Target / ***Source***	**N1**	**N2**
N1		**i1N2S1** **i1N2S2**
N2	**i1N1S1**	

Table 8. *Internodal/Intranodal System-to-System Interface Description.*

Source \ Target	N1S1	N1S21	N2S22	N2S1	N2S2
N1S1		—	—	—	—
N1S21	—		—	—	**i1N1S2**
N1S22	—	—		**i1N2S1**	—
N2S1	—	—	—		—
N2S2	**i1N1S1**	—	—	**i2N2S1**	

A more detailed description can be achieved by replacing the interface names in the N2 diagrams with the assigned messages and associated parameters. N2 diagrams can be generated from the data in the model repository.

Appendix D.Model Libraries

Editorial Comment: Guidelines for model library definitions are still being established.

D.1 Requirements Model Library

D.1.1 Requirement Taxonomies

Requirement taxononomies can be created by specializing SysML requirement stereotypes. Typical examples may include operational, behavioral, interface, control, performance, physical, storage, design constraints, and other specialized requirements for reliability, safety, etc..

Table 1 Example Requirements Taxonomy

Requirement subclass	Constraints
Operational	Satisfied by an activity, operation, or method.
Functional	Satisfied by an activity or method.
Interface	Satisfied by a connector, port and itemflow.
Performance	Satisfied by one or more performance properties and associated parametric relationship.
Activation/deactivation	Satisfied by a state machine, activity, or sequence diagram.
Storage	Satisfied by any part that represents a stored item.
Physical	Satisfied by one or more physical properties and associated parametric relationships or a geometric model.
Design constraint	Satisfied by any white box model element such as parts, ports, connectors, or implementations of black box behavior.
Specialized	Satisfied by any element modeling the specialized requirement.
Measure of Effectivness	Satisfied by class attribute reporting a simulated/measured performance of a system.

Specific constraints between subclasses of requirements and SysML model elements can be imposed by the stereotype. For example one can enforce that a behavioral requirement shall be satisfied by SysML activities or functions. Similar constraints can be imposed by the other subclasses of requirements.

D.1.2 Extending Requirement Attributes

The standard SysML Requirement stereotype specifies minimal attributes for flexibility reasons. We give below an example of stereotype extension for critical requirement. This stereotype provides an attributes for the criticality of the requirement.

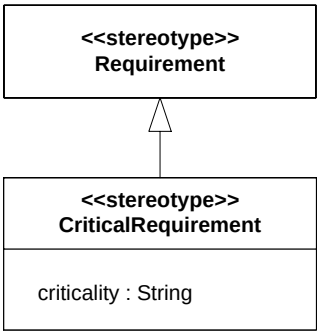


Figure D-1. Example of critical requirement stereotype.

D.1.3 Extending Requirement Relationships

In Figure D-2 we show some examples of relationships between requirements for the generic «satisfy» relationship and the «trace» relationship. These stereotype extensions are given as examples and are non-normative.

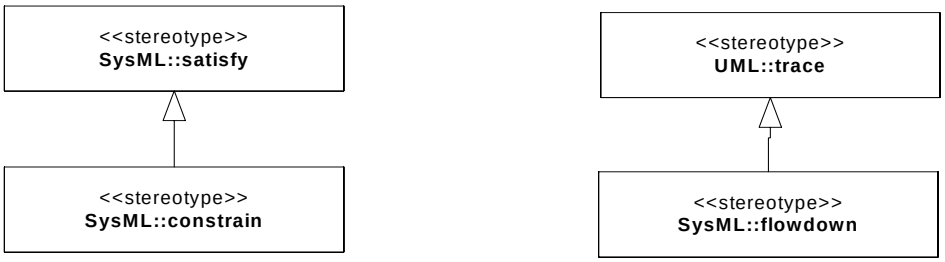


Figure D-2. Examples of relationships between requirements.

D.1.4 Alignment with UML Testing Profile

The UML Testing Profile provides a framework for modeling Test Cases within the extended framework of Test Suites. The scope of the testing profile goes beyond the ones of the UML for System Engineering RFP. Nevertheless, the example of the stereotype extension below provides a means for aligning the SysML TestCase stereotype with the one in the UML Testing Profile. Note that the enumeration UMLTestingProfile::Verdict is only given here for information about the set of possible tagged values for the attribute *verdict*.

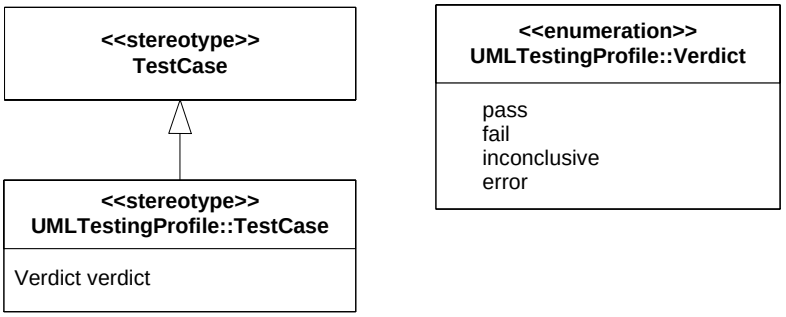


Figure D-3. Alignment of the SysML Test Case stereotype with the UML Testing Profile.

D.2 SI Units Model Library

The elements of the model library SIUnits instantiate the Unit and Dimension classes of the Quantities model library from the Auxiliary chapter.

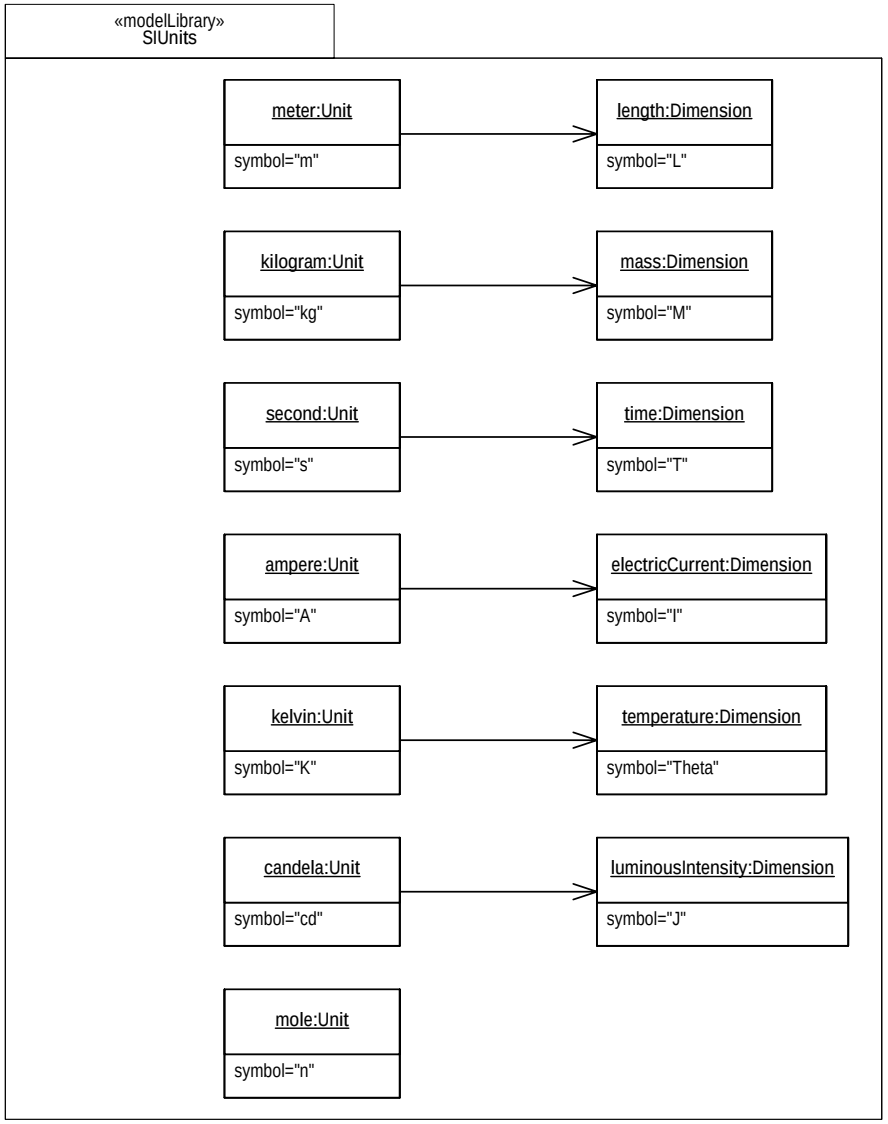


Figure D-4.

D.3 Distributions Model Library

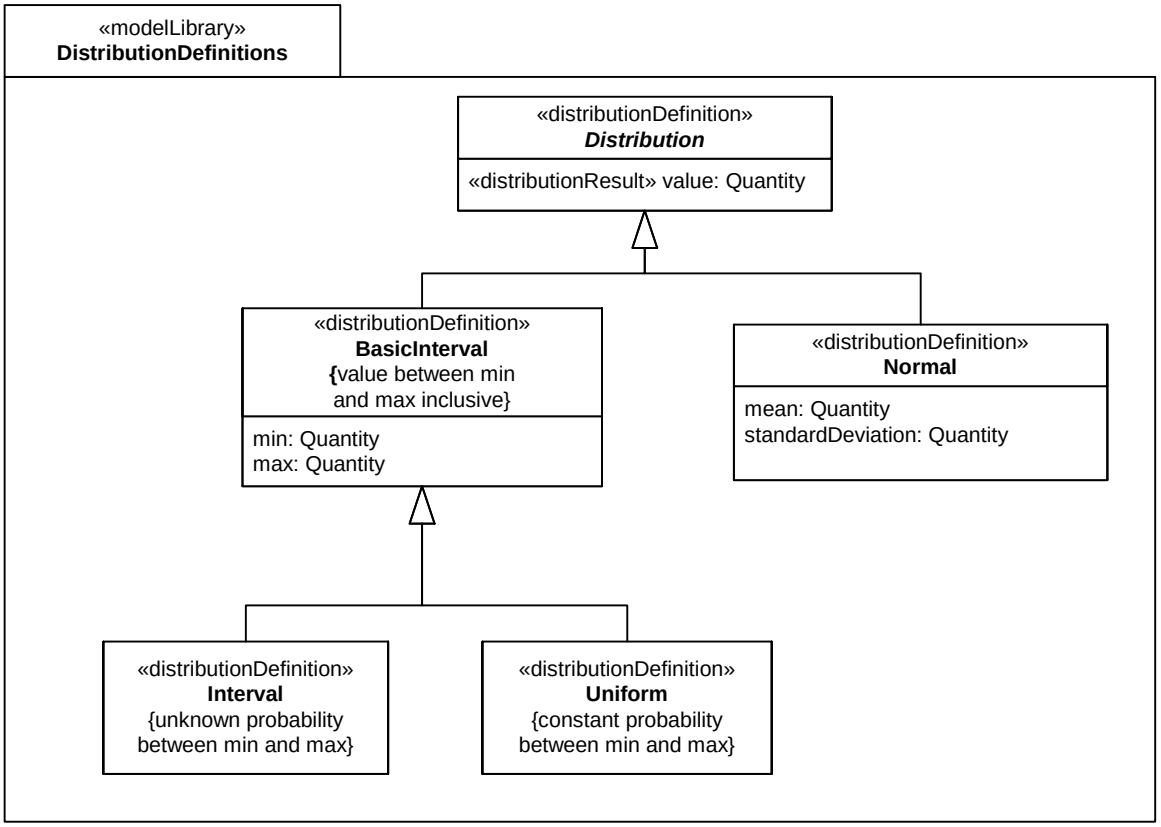


Figure D-5. Standard distributions defined in model library.

D.3.1 Distribution

Description

Abstract base class for distribution definitions.

Diagram extensions

The value of a concrete distribution definition is shown as follows:

> *'('<name of distribution definition>')''('<property name>'='<property value>*
> *[',' <property name>'='<property value>]*')'*

It is allowed to hide properties.

Appendix E. Requirements Traceability

This appendix describes the requirements tracability matrix (RTM). The RTM 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::

- 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.
- d) Adescription of how SysML addresses the requirement. Note: In some cases, there may be other SysML solutions to satsify 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 chapter in the SysML specification. Note: The reference to a chapter may require reference to a corresponding chapter in the UML specification. For example, when the classes chapter is referenced, this may include a combination of the SysML classes chapter and the UML classes chapter.

Editorial Comment: The concrete syntax that supports the solution can be found in the Diagram Element tables of the chapter referenced in f) above. Representative examples can be found in the same chapter referenced, as well as in the Sample Problem appendix.

Table 1 Requirement Traceability matrix

UML for SE Req't #	Requirement name	Compl (Y/N, Partial)	Requirement Satsifaction	Metaclass Extension	SysML Diagram Chapter	Version
6.5	Mandatory Requirements					
6.5.1	Structure		Structure diagrams include class and assembly diagrams		Structure	
6.5.1.1	System hierarchy	Y	Class aggregation/composition and parts in assembly diagrams are the primary mechanisms for depicting hierarchy.	UML::Class, UML::Association, SysML::Assembly, UML::Property	Class, Assembly	1.0
	a. Subsystem (logical or physical)	Y	Typically represented by a set of logical or physical parts in an assembly diagram that realize one or more system operations. The corresponding sequence diagram and swim lane diagram can represent a hybrid of structure and behavior.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0
	b. Hardware (i.e. electrical, mechanical, optical)	Y	Represented by a class, assembly, or part.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0
	c. Software	Y	Represented by a class, assembly or part or a UML component.	UML::Class, UML::Component, SysML::Assembly, UML::Property	Class, Assembly	1.0
	d. Data	Y	Represented by a class, assembly, or part. Refer to input/output requirements in 6.5.2.1.1 and 6.5.2.5 for data flows.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0
	e. Manual procedure	Y	Represented by a class, assembly, or part. Can also be represented by the standard UML stereotype <<document>>.	UML::Class, SysML::Assembly, UML::Property, UML::Document	Class, Assembly	1.0
	f. User/person	Y	Represented by a class, assembly, or part. External users are also represented as actors in a use case diagram.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0
	g. Facility	Y	Represented by a class, assembly, or part.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0

	h. Natural object	Y	Represented by a class, assembly, or part.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0
	i. Node	Y	Represented by a package containing a set of packageable elements based on some partitioning criteria.	UML::Package	Class	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 class or assembly or a stereotype of a classifier. Also, represented as actors in use cases.	UML::Class, SysML::Assembly, UML::Property, UML::Classifier	Class, Assembly, Use Case	1.0
6.5.1.3	System interconnection		Assembly diagram shows connections using parts, ports, and connectors. Class diagram with associations can depict communication paths.	UML::Class, SysML::Assembly	Class, Assembly	1.0
6.5.1.3.1	Port	Y	A port is a part that is on the boundary of another part or assembly that supports interactions with its environment.	UML::Port, UML::Property	Assembly	1.0
6.5.1.3.2	System boundary	Partial	The set of ports that represent the interaction points for an assembly or part.	UML::Port, UML::Property	Assembly	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). A connector can be decomposed to connect nested ports.	UML::Association, UML::Connector, SysML::NestedConnectorEnd UML::Port	Class, Assembly	1.0

6.5.1.4	Deployment of components to nodes	Y	A subtype of an allocation relationship called deployment allocation between named elements that can correspond to different types of components. This a more abstract form of deployment than UML deployment. Note: Refer to allocation relationship.	SysML::Allocation, SysML::Allocated, UML::NamedElement	Allocation, Special Usages Appendix	1.0
	a.	Y	Software part or assembly deployed to a hardware part or assembly (processor or storage device).	SysML::Allocation, SysML::Assembly, UML::Property	Allocation, Special Usages Appendix	1.0
	b.	Y	Generalized deployment relationship between a deployed element and its host.	SysML::Allocation, SysML::Assembly, UML::Property	Allocation, Special Usages Appendix	1.0
	c	Y	Deployed element and host can be decomposed using assemblies and parts.	SysML::Assembly, UML::Property	Allocation	1.0
6.5.2	Behavior		Behavior diagrams include activity, interaction (sequence, timing), and state machine diagrams. Communication diagrams and interaction overview 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.	Behavior Diagrams	Behavior	
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	Activity	

6.5.2.1.1	Input/Output	Y	Inputs and outputs can be represented as object nodes flowing between activity nodes in an activity diagram, parameters of activities, and as itemProperties that flow between parts in an assembly diagram. The itemProperties are associated with an ItemFlow.	UML::Class used as the type of UML::Parameter, UML::ObjectNode, or UML::Property, SysML::ItemFlow	Activity, Auxiliary	1.0
	a	Y	Object nodes, parameters and item properties are typed by classifiers that can have properties.	UML::Class, UML::ObjectNode, UML::Parameter, UML::Property	Activity, Auxiliary	1.0
	b	Y	The classifiers that represent the things that flow (type of object node, parameter, and itemProperty) can be decomposed and specialized.	UML::Classifier	Activity, Auxiliary, Class,	1.0
	c	Y	"ItemFlows" associate the things that flow with the connectors that bind the ports or the associations between classes. The parameters and object nodes are bound to the corresponding activities and actions.	SysML::ItemFlow	Auxiliary, Assembly, Class	1.0
6.5.2.1.2	System store	Partial	Stored items can be represented as parts of an assembly or class, and also represented in an activity diagram as object nodes or central buffer nodes.	UML::Classifier, UML::Property, UML::ObjectNode UML::CentralBufferNode	Class, Assembly; Activity, Auxiliary,	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::DataStoreNode	Activity	1.0
	b	Y	A stored item can be the same type of classifier as an input or output in both an assembly diagram and an activity diagram. The classifier supports different roles (store vs. flow).	UML::Classifier	Class, Assembly, Activity	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	Activity, Sequence, State Machine	1.0
	a	Y	Behaviors and the associated parameters are named (i.e. name of activity and activity parameter node).	UML::Behavior	Activity, Sequence, State Machine	1.0
	b	Y	The action semantics define different types of actions that include CreateObject, DestroyObject, ReadStructuralFeature (monitor), and WriteStructurealFeature (update). A SysML NullTransformation has also been added. A CallBehavior action is a generalized action that can call any behavior (activity, interaction, state).	UML::CreateObjectAction, UML::DeleteObjectAction, the various object modification actions in UML, monitoring with UML::AcceptEventAction, and SysML::NullTransformation.	Activity, Sequence, State Machine	1.0
	c	Y	The object nodes (pins) bind input and output parameters to actions.	UML::ObjectNode, UML::Pin	Activity	1.0
	d	Y	The queing semantics for object nodes are specified. The default queing is FIFO, but other forms of queing includeing LIFO, ordered, and unordered as defined by the enumeration for ObjectNodeKind.	UML::Behavior, SysML::InputPin, SysML::ObjectNode	Activity	1.0

	e	Partial	A resource stereotype applied to a constraint has a property that refers to the class that represents the resource. Resource constraints to support an execution can also 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::Resource-Constraint, SysML::Parametric-Constraint	Activity, Parametric	1.0
	f	Y	Refer to c	UML::ObjectFlow, UML::Pin	Activity	1.0
	g.	Y	An activity can be decomposed into lower level actions that invoke other activities.	UML::Activity, UML::CallBehavior-Action, UML::Activi-tyParameterNode, UML::ObjectFlow, UML:: Pin	Activity, Class	1.0
	h.	Y	An action has control inputs that can enable the exectuion of a function, and a control value input from a control operator that can both enable or disable an exectuion of a function. An execution of a function can also be terminated when it is enclosed in an interruptile region. Alternatively, state machine diagrams can be used to enable or disable execution upon transition events.	UML::Action, UML::Interrupt-ibleActivityRegion, SysML::Con-trolValue, UML::State	Activity, State Machine	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.	UML::Activity, UML::Action	Activity, Sequence, State Machine	1.0

	j	Y	A continous or discrete rate stereotype can be applied to inputs and outputs. Inputs and outputs are discrete by default. A 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 ouputs while executing if the inputs and outputs are also streaming.	SysML::Rate SysML::Continous, SysML::Discrete UML::Parameter (isStream=Value)	Activity, State Machine	1.0
	k	Partial	Different actions can invoke concurrent executions of the same generalized behavior. Actions can have multiplicity.	UML::Behavior, UML::Action	Activity, Class	1.0
6.5.2.2	Function activation/deactivation				Activity, Sequence, State Machine	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	Activity, Sequence, State Machine	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.isControl=true for control queuing	Activity	1.0
	b	Y	Control inputs are discrete valued inputs that can enable or disable an activity node.	SysML::ControlValue	Activity	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	Activity	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::InterruptibleActivityRegion, SysML::ControlValue, UML::State	Activity, State Machine	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::InterruptibleActivityRegion, SysML ControlValue, UML::TimeExpression, UML::State	Activity, State Machine	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	Activity	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	Activity	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::ControlOperator, SysML::ControlValue	Activity	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::ControlOperator, SysML::ControlValue, UML::Parameter	Activity	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::ControlOperator, SysML::ControlValue	Activity	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::InteractionOperator	Activity, Sequence	1.0
	c1	Y	Decision nodes in activity diagrams support selection. The "alt" Interaction Operator supports selection in sequence diagrams.	UML::DecisionNode, UML::InteractionOperator.Alt	Activity, Sequence	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::InteractionOperator.par	Activity, Sequence	1.0
	c3	Y	A join defines the “anding” of multiple flows together resulting in a single output flow.	UML::Join	Activity	1.0
	c4	Y	Merge supports multiple input flows resulting in a single output flow.	UML::Merge	Activity	1.0
	c5	Y	Decision and loop nodes support iteration and looping. The “loop” Interaction Operator supports loops in sequence diagrams.	UML::DecisionNode, UML::LoopNode, InteractionOperator.loop	Activity, Sequence	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.		Activity, Sequence, State Machine	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::AcceptEventAction including UML::TimeTrigger, UML::EventOccurence in Interactions,Note: Failure event can be result in variouts types of actions that terminate an Interruptible Region in Activiites, etc.	Activity, Sequence, State Machine	1.0
	b	Y	Refer to a) above	UML::ActivityEdge, UML::Trigger	Activity, Sequence, State Machine, Timing	1.0
	c	Y	Conditions can be specified as constraints that define guards to control execution of behaviors.	UML::Constraint (guard)	Activity, Sequence, State Machine, Timing	1.0
6.5.2.3	Function-based behavior	Y	Activity diagrams provide the capability to model function based behavior.	UML:: Activity	Activity	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 Machine	1.0

	a	Y	State	UML::State	State Machine	1.0
	b	Y	Simple state	UML::State, isSimple=True	State Machine	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 Machine	1.0
	d	Y	Transitions between states which are triggered by events with guard conditions.	UML::Transition, UML::Trigger	State Machine	1.0
	e	Y	Transition within a composite state	UML::Transition (TransitionK-ind=Internal)	State Machine	1.0
	f	Y	Pseudo states include joins, forks and choice	UML::PseudoState	State Machine	1.0
	g	Y	Transitions between states which are triggered by events with guard conditions.	UML::Activity	State Machine	1.0
	h	Y	Entry, exit, doActivities are performed upon entry or exit from a state or while in a state.	UML::Activity	State Machine	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 Machine	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 Machine	1.0
	k	Y	Send and receive signals can be sent via actions to interact with other objects.	UML::SendSignalAction	State Machine	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 Machine	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). A timing diagram can be used to model the time associated with the occurence of events, such as state changes, or changes in property values.	UML::SimpleTime package	Activity, Interaction (Timing Diagram), State Machine	1.0
6.5.2.5	Allocation of behavior to systems	Y	An allocation relationship provides a genearlized capability to allocate one model element to another including the capability to allocate behavior to system structure (assemblies).	SysML::Allocation, SysML::Allocated, UML::NamedElement	Allocation, Special Usages Appendix	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 assembly. The SysML Allocation relationship can be used to explicitly allocate activities to assemblies. Alternatively, activity partitions (swim lanes) can be used to allocate the action and/or activity to a part and/or assembly.	UML::BehavioredClassifier and UML::Behavior (owned behavior) - Refer to UML Common Behaviors, SysML::Allocation, SysML::AllocatedPartition	Allocation, Special Usages Appendix	1.0
	b	Partial	An object node in an activity diagram can be allocated to an item that flows in an assembly 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.	UML::Class (type of ObjectNode to type of ItemProperty), UML::ObjectNode, UML::Property	Allocation, Special Usages Appendix, Auxiliary (Item Flows)	1.0
6.5.3	Property					
6.5.3.1	Property type	Y	Primitive types provide the capability to model the different types of properties.	UML:: PrimitiveType, SysML::PrimitiveType	Auxiliary	1.0

	a	Y	Primitive type.	UML::Integer	Auxiliary	
	b	Y	Primitive type.	UML::Boolean	Auxiliary	
	c	Y	Primitive type.	UML::Enumeration	Auxiliary	
	d	Y	Primitive type.	UML::String	Auxiliary	
	e	Y	Primitive type.	SysML::Real	Auxiliary	
	f	Y	Data type.	UML::Complex	Auxiliary	
	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	Property can be typed by a quantity which has a value.	UML::Property, SysML::Quantity	Parametric, Auxiliary	1.0
	b	Y	A quantity can include units for its values. The units can relate a value to a dimension suchas length.	SysML::Unit	Auxiliary, Model Library	1.0
	c	Y	A distributed quantity is a quantity with an associated probability distribution on its values.	SysML::DistributedQuantity, SysML::DistributionDefinition, SysML::DistributionResult	Auxiliary, Model Library	1.0
	d	Y	Source data can be defined via an attached comment to the property.	UML::Comment	UML Class	1.0
	e	Y	Reference data can be via an attached comment to the property.	UML::Comment	UML Class	1.0
6.5.3.3	Property association		A property can be a feature of any classifier.			1.0
	a	Y	Assemblies, parts of assemblies, or items that flow (classifiers) can have (or reference) properties.	UML::Class, SysML::Assembly, UML::Property	Class, Assembly	1.0
	b	Y	A function (activity) can have properties since it is a class	UML::Activity	Class, Activity	1.0

	c	N	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.			
	d	Y	A property can be related to other properties through a parametric constraint.	SysML::ParamConstraint, UML::Property	Parametric	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 may be continuous or discrete. Clocks will have a property that can be connected, via a property binding to a parametric constraint usage. 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, peformance, and time profile.	UML::Property, SysML::ParamConstraint, UML::SimpleTimePackage	Parametric, Interactions (Timing)	1.0
6.5.3.5	Parametric model	Y	The parametric diagram supports modeling of parameters.	SysML::ParamConstraint, UML::Property, UML::Connector, SysML::NestedConnectorEnd UML::Port		1.0
	a	Y	Parametric constraints specify the mathematical relationships among properties.	SysML::ParamConstraint	Parametric	1.0
	b	Partial	Mathematical and logical expressions can be defined in SysML in a reference language, but there is not interpreter built into SysML. The range of values can be specified via quantities and probability distributions per 6.5.3.2a-c.	SysML::ParamConstraint	Parametric	1.0

	c	Y	The reference language for interpreting the parametric constraint can be included as an attached comment or in the compartment along with the expression.	SysML::ParamConstraint UML::Comment	Parametric	1.0
6.5.3.6	Probe	Partial	A probe can be an assembly that types a part that is connected to a port. 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.	UML::Class, SysML::Assembly UML::Property	Assembly	1.0
6.5.4	Requirement			Requirement	Y	
6.5.4.1	Requirement specification	Y	A requirement is a stertotype of a class in SysML. The various subtypes of requirement are specified as subclasses of the the requirement stereotype and can include specific propeties and constraints on what model elements can satisfy the subclass of requirement. A sample set of subclasses of requirements are included in the Model Library Appendix.	SysML::Requirement	Requirement, Model Library	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::Quantity	Requirement	1.0
	Note 2	Partial	There is no explicit subclass of requirement as a stakeholder need, but a requirement can be named or subclassed as “stakeholder need”.	SysML::Requirement	Requirement, Model Library	1.0
	Note 3	Partial	User defined requirements can be added via subclasses to specify any type of life cycle requirement of interest to the modeler.	SysML::Requirement	Requirement, Model Library	1.0
	a	Y	Operational requirement	SysML::OperationalRequirement	Requirement, Model Library	1.0
	b	Y	Functional requirement	SysML::Functional-Requirement	Requirement, Model Library	1.0

	c	Y	Interface requirement	SysML::InterfaceRequirement	Requirement, Model Library	1.0
	d	Y	Performance requirement	SysML::PerformanceRequirement	Requirement, Model Library	1.0
	e	Y	Activation/Deactivation (Control) requirement	SysML::Activation/DeactivationRequirement	Requirement, Model Library	1.0
	f	Y	Storage requirement	SysML::StorageRequirement	Requirement, Model Library	1.0
	g	Y	Physical requirement	SysML::PhysicalRequirement	Requirement, Model Library	1.0
	h	Y	Design constraint	SysML::DesignConstraint	Requirement, Model Library	1.0
	i	Y	Specialized requirement	SysML::SpecializedRequirement	Requirement, Model Library	1.0
	j	Y	Measure of effectiveness	SysML::MeasureOfEffectiveness	Requirement, Model Library	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.reqProperty	Requirement, Model Library	1.0
6.5.4.3	Requirement relationships	Y	The requirement relationships include trace and their specialization for derive, satsify,and verify. The trace relationship can be futher specialized. Containment is the relationship to decompose requirements.	UML::Trace	Requirement	1.0
	a	Y	A derive relationship,which is a stereotype of a trace, relates a derived (target) requirement to a source requirement.	SysML::Derive	Requirement, Model Library	1.0

	b	Y	A requirement satisfy relationship, which is a stereotype of a trace, relates the model elements (i.e. the design) that satisfy the requirement.	SysML::Satisfy	Requirement, Model Library	1.0
	c	Y	Goals, capabilities, or usages of systems are often expressed using use cases. Subgoals can be represented using the include or composition relationship between use cases. Requirements can be nested into lower level requirements using the containment relationship or trace relationship.	UML:UseCase, UML::Include, SysML::Requirement, UML::Trace	Requirement, Use Case	1.0
6.5.4.4	Problem	N				1.x
6.5.4.5	Problem association	N				1.x
6.5.4.6	Problem cause	N				1.x
6.5.5	Verification	Partial	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]			
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	Requirement	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. Alternativly, a reference to a TestCase using SysML:Rationale may be attached to a satisfy relationship.	SysML::Verify SysML::Rationale	Requirement	1.0
	c	Y	A trace relationship between the requirement being validated and the higer level requirement or need may have a Rationale attached that references the validation method(s).	UML::Trace, SysML:Derive SysML::Rationale	Requirement	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	Requirement	1.0
	Note 2	Partial	Validation methods are user defined. SysML::Rationale can reference any of the user defined methods.	SysML::Rationale	Requirement	1.0

6.5.5.2	Test case	Partial	A test case refers to the method for verifying a requirement. Note: The testing profile associates a test case with a behavior that can include the specific method and associated input stimulus and response.	SysML::TestCase	Requirement, Behavior (Activity, Sequence, State Machine	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 requirement			1.x
	Note 3	Partial	Test cases can be composed of other test cases, like any other named element.	SysML::TestCase	Requirement	1.0
6.5.5.3	Verification result	Partial	The result of a SysML::TestCase may be expressed through its Verdict attrbute	SysML::TestCase.Verdict	Requirement	1.0
6.5.5.4	Requirement verification	Partial	A parametric constraint may be used to relate the required value with the verification result.	SysML::ParamConstraint;SysML::TestCase.Verdict, SysML::Rationale	Requirement	1.0
6.5.5.5	Verification procedure	Partial	A rationale can be associated with the test case or the satisfy relationship between a requirement and a design, and reference a verification procedure. Note: The testing profile will associate a behavior with a test case which can be implemented by a specific procedure.	SysML::TestCase, SysML::Rationale	Sequence	1.x
	Note					
6.5.5.6	Verification system	Partial	A verification system can be modeled as any other system (assembly) 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::Assembly		1.x
6.5.6	Other					
6.5.6.1	General relationships	Y	The UML general relationships support the following requirements.			
	a	Y	An association relationship.	UML::Association	Class	1.0

	b	Y	A package contains packageable elements and can represent collections of elements.	UML::Package, UML::PackageableElement	Class	1.0
	c	Partial	Classes can be decomposed using aggregation/composition. Assemblies are classes that are decomposed into parts (refer to Reqt 6.5.1.1). The completeness of the decomposition is not explicitly represented.	UML::Association (aggregation, composition), UML::Property	Class, Assembly	1.0
	d	Y	A dependency relationship.	UML::Dependency	Class	1.0
	e	Y	Generalization/specialization relationship. Generalization sets provide the means to partition specializations to support further categorization.	UML::Generalization, UML::GeneralizationSet	Class	1.0
	f	Y	Instantiation is modeled using Instance Specifications to uniquely identify a classifier.	UML::InstanceSpecification, UML::InstanceValue	Class	1.0
6.5.6.2	Model views	Partial	A view is a stereotype of model that represents the model from a particular viewpoint. The viewpoint is a perspective of a set of stakehodlers and their concerns. Both the view and the viewpoint are represented in SysML. The view identifies the set of model elements that address the viewpoint, and the viewpoint specifies the stakeholders and their concerns.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	Auxiliary	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 requriements of this RFP. In addition, a formal 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::DiagramUsage	Diagram Appendix	1.0
6.5.6.4	System role	Patial	A part in an assembly represents the role for a classifier in the context of the enclosing assembly. It defines the relationship between an instance of the classifier that types the part and an instance of the assembly that encloses the part. This is a primary mechanism for providing a unique context for a part of a whole (enclosing assembly). The part may use only a subset of the behavior and properties of the class that types the part. However, the specific mechanism for representing the subset has not been defined.	UML::Property	Assembly	1.0
6.6	Optional Requirements					
6.6..1	Topology					
	a	N				1.x

	b	Partial	A class diagram can be used to model relationships between classes.	UML::Class, UML::Association, UML::Dependency, UML::Generalization	Class	1.0
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	Auxiliary	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	Specific constructs for criteria, weighting, and alternatives are planned for a future version of SysML to support modeling of trade studies. 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.	SysML::ParamConstraint	Parametric	1.x
	a	N				1.x
	b	Partial	Criteria can be modeled as properties typed by quantites.	UML::Property, SysML::Quantity		1.x
	c	Partial	Measures of effectiveness can be modeled as a subclass of requirement. The requirement can include properties that are typed by quantities. The parametric constraint can represent the optimization function which constraints the properties.	SysML::MeasureofEffectivenes, UML::Property, SysML::Quantity, SysML::ParamConstraint		1.x
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					
	a	Y	The action semantics provide the capability for creating and destroying objects.	UML::CreateObjectAction, UML::DestroyObjectAction	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	Activity, Sequence, State Machine	1.0
6.6..8	Integration with domain-specific models	Partial	SysML is a general purpose language that can integrate with other types of domain specific models. This is accomplished in part by the alignment with the AP-233 data interchange standard. In addition, the parametric diagram is intended to provide a capability to integrate with domain specific engineering analysis models.		Parametric, AP-233 Appendix	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	1.x
6.6..10	Management Model	N				

Appendix F. ISO AP233 Alignment

Editorial Comment: The architectural alignment of SysML and AP233 is an ongoing activity since AP233 is not finalized yet. Therefore this chapter reflects the current status of the work. It gives background information on AP233, the alignment approach and a mapping of the requirements module.

F.1 Background

This appendix describes the alignment activities between SysML and ISO 10303 AP233. SysML as thoroughly described in this document is the standardized modeling language for systems engineering. SysML will support the whole product/project lifecycle from specification through to verification, validation and maintenance of the system. This involves engineers from various disciplines and their specific tools. As systems engineers are in charge to coordinate the systems development the data or model has also to be shared between the systems engineers and the domain engineers.

AP 233 is a data exchange protocol for systems engineering data based on ISO10303 STEP (Standard for the Exchange of Product Model Data). The basic idea of STEP to provide a tool neutral data exchange schema based on a tool-independent meta-model. The history of STEP ranges back to the early 80s.

As SysML is based on UML 2 it can make re-use of its infrastructure for data interchange. The obvious way to do pursue this is XMI (XML Meta-model Interchange). This is a very efficient way to exchange data between UML based tools. However the most benefit do have UML based tools as they do implement the same meta-model based on the specification. But for tools which meta-model is not based on the UML meta-model the integration on XMI is not impossible but more difficult. Therefore STEP is an interesting option

For a real tool interoperability on enterprise level it is very important that the SysML data exchange is not limited to XMI. In the frame of systems engineering ISO STEP AP233 plays an important role to help to provide interoperability within the systems engineering tools but also to have a mapping to the domain engineering discipline related data.

F.2 ISO 10303 STEP

STEP (ISO 10303) is a standard to describe, represent and exchange industrial data in a computer interpretable format. For different application domains for data exchange a neutral (i.e. tool independent) data model has to be defined. This is called AP (Application protocol). Currently around 40 different APs are defined. Among other the following application protocols are defined respectively under definition:

- AP203: Configuration controlled design
- AP210: Electronic assembly, interconnection and packaging design
- AP214: Core data for automotive mechanical design process
- AP233: Systems Engineering
- …

The definition of a protocol follows in a MDA like structure. It consists of the following elements:

- ARM: Application reference model which is a implementation independent model of the data
- AIM: Application implementation model is an implementation of the ARM using
- Mapping Table between the ARM and the AIM

- Implementation methods driven from EXPRESS

To support the modeling of ISO 10303 application protocols, ISO 10303 consists of different parts to support the definition and utilization of data models:

Descriptive methods: The modeling language for the data models is EXPRESS (part 12, respectively 14) which has a ASCII and a graphical notation (EXPRESS-X).

Implementation methods for the implementation of the data model. It consists of

- Part 21: Clear text encoding of the exchange structure (STEP file)
- Part 23: C++ language binding of the standard data access interface
- Part 24: C language binding of the standard data access interface
- Part 25: EXPRESS (modeling language) binding to OMG XMI
- Part 28: XML representation for EXPRESS-driven data
- Conformance testing methodology and framework

Integrated resources which are models common to the different application protocols and can be re-used for the implementation

For the integration of a tool to an ISO 10303 application protocol the tool vendor has to do the mapping between the tool-internal (proprietary) information models to the standardized data exchange model using an API (e.g. using a part 23 based C++ API). Because of the multi layer modeling approach the tool vendors do have many options in terms of API and data appearance.

F.3 ISO 10303 / AP233

AP233 is a data exchange protocol for systems engineering data based ISO 10303 and is a neutral data exchange schema for systems engineering data. AP233 is to support the whole system development life cycle ranging from requirements definition to system verification and validation. Within projects the system engineering activities tie together the different domain engineering disciplines one consistent view to the system.

The same applies to the data. Systems engineering data forms the core of a systems description and has to be linked to the remaining domain engineering data. The following UML diagram shows the AP233 and its relationship to other protocols. AP233 makes re-use of the STEP-PDM definitions, indicated by the dependency to STEP-PDM with the stereotype <<uses>>. The remaining packages represent other STEP application protocols. The dependencies can be read as “AP233 depends on the definition of …” which means if a protocol will be changed this would mean that AP233 could be affected too.

The following areas are covered in the AP233 data model:

- Requirements
- Functional architecture
- Physical architecture
- Verification/Validation
- Management
- Supporting Modules as work, person, properties

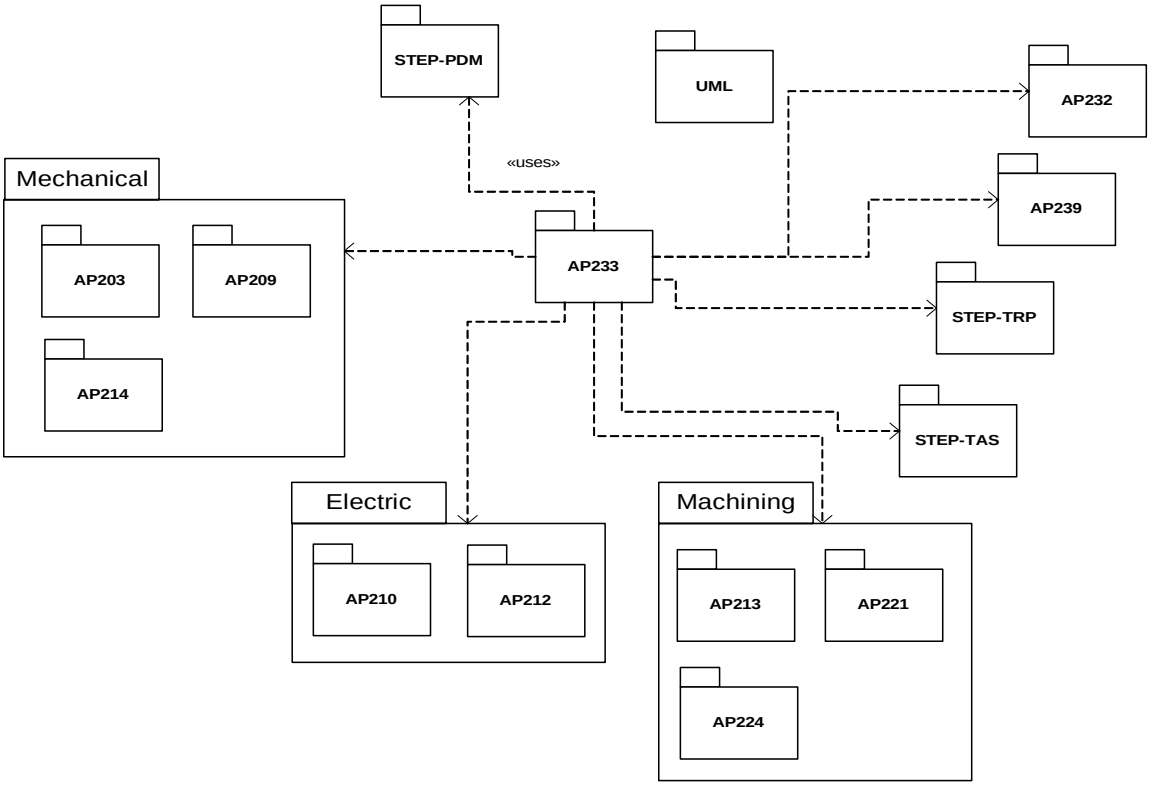


Figure F-1. AP233 and related protocols

The basic item in AP233 is a product. This is shown in the next diagram. Each package makes re-use of the definition of a product (in the PDM sense). A requirement in the requirement package is derived from Product (in the package Product) and inherits the properties of Product.

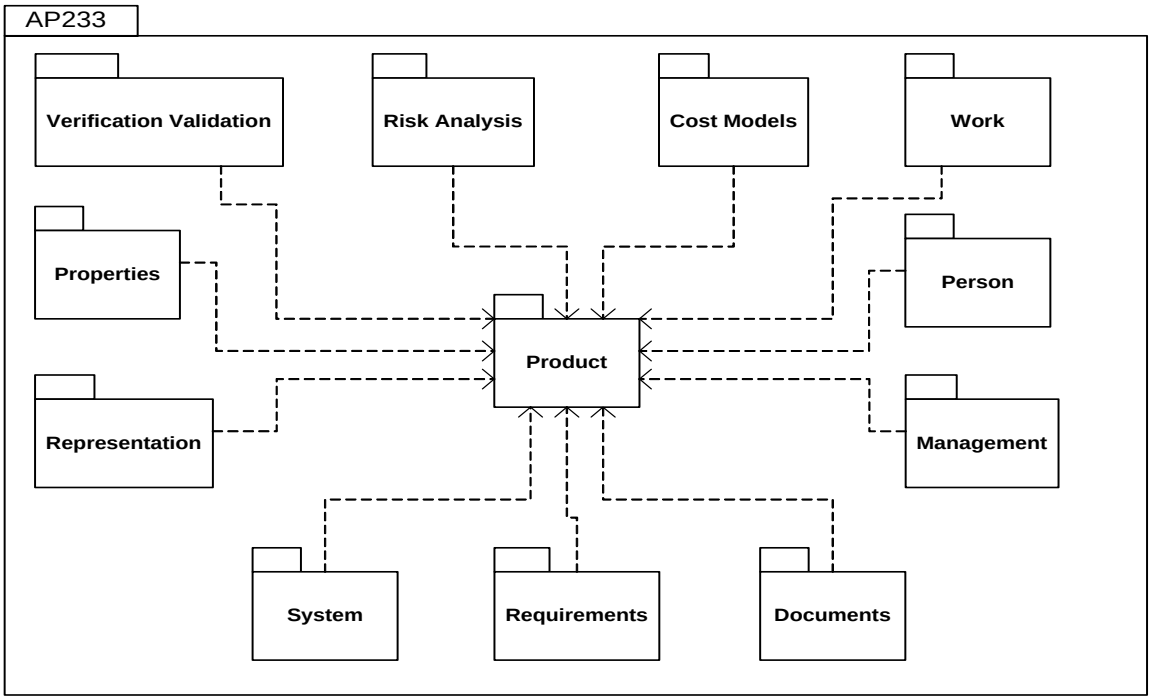


Figure F-2. AP233 Toplevel Architecture

AP233 is a follow-on activity of the European SEDRES (System Engineering and Exchange Standardisation) project which developed systems engineering data model based on the STEP technology. AP233 was launched based on the SEDRES results and started with a modularization of the data model to ease re-use of parts of other protocols.

The current status of AP233 is the following:

- Requirements module implemented
  - Text-based requirements
  - Property based requirements
  - Basic Structure module
  - Tracing between structure and requirements
- AP233 Demonstrator Tool: To ease the understanding, demonstration and utilization of AP233 a demonstrator tool is under development as part of the AP233 activities. It implements basic features for the definition of requirements (in different appearances like text, property and spread-sheet), a system break-down and traceability between requirements and systems. Beside that it has multiple interfaces to read and write the data, not only in STEP and XML but also interfaces to the Office world such as Work and Visio.
- Next Steps:
  - Structural Module
  - Behavioral Module
  - Risk Module

- Scheduling Module
- Rules Module
- Cost Module

The major stakeholder for the AP233 development is the Incose/MDSD (Model-Driven System Design).

F.4 Approach

From systems engineering perspective the upcoming SysML tools are just a subset of tools which are used throughout the life cycle for system development and maintainance. The challenge for any tool integration activity is to provide a mapping between the different meta models which are used to capture the tools data. As ISO10303 STEP and in particular AP233 (for systems engineering) provide a neutral data repository for tool integration, the SysML meta model has to be mapped to AP233.

In order to decouple the different meta-models from AP233 and SysML it has been decided to define a mapping model. This mapping model than is used to map the correspondent elements of AP233 and SysML to it. The mapping model is a high-level (independent) representation of the systems engineering concepts implemented in SysML and AP233. The mapping model is defined in UML.

The AP233 modeling is done using the STEP modeling language Express. Basically Express implements similar concepts as UML: classes(entities), attributes, associations and inheritance. In addition to that Express has some data modeling related modeling elemens currently not implemented in UML. But in order to have a common mapping platform it has been decided to perform the mapping in UML. For this it is necessary to convert the AP233 model to UML. In order to achieve common semantics for the AP233 model in UML a dedicated profile has been developed.

Appendix F-3 shows the relationship between the different models. At the bottom the mapping model can be seen which is used to bridge the AP233 and SysML meta models. On the right hand side the AP233 model (in UML) is an instantiation of a profiled UML, to capture and preserve Express semantics. The left hand side shows SysML as an extension of UML2.

UML1.x can be sufficiently used to capture Express based on a dedicated express profile, but will be replaced with UML2 anyway.

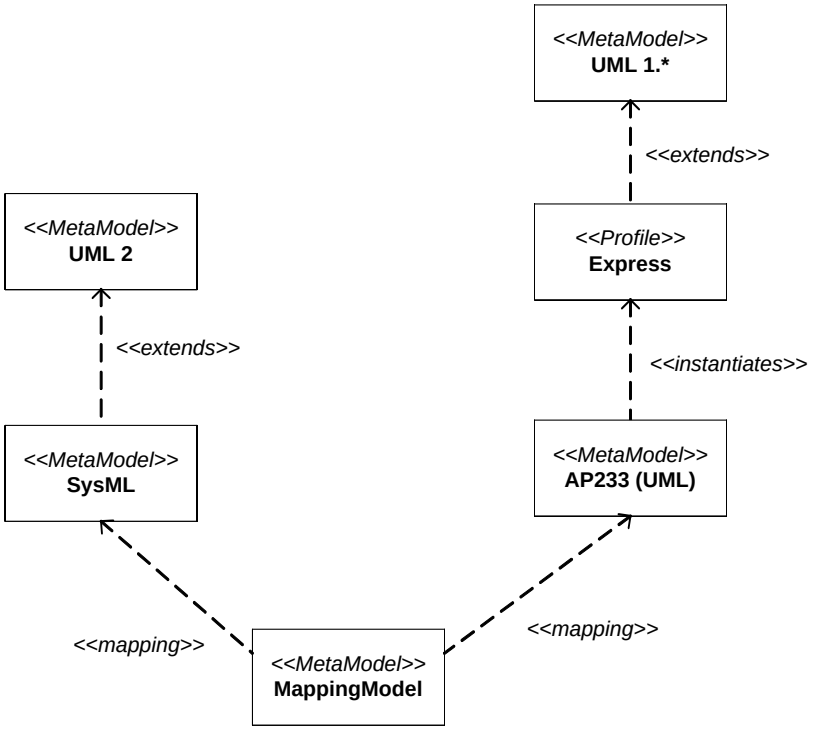


Figure F-3. Models in use for the SysML to AP233 alignment

F.4.1 Capturing Express models in UML

The development of the UML profile for Express is shown in the Appendix 11-19 . In the diagram shows only the ‘core’ subset of the modeling language Express. The basic element is an entity which can be compared to a class in UML. It may have attributes, attributes and associations. The attribut definition is similar to UML. Attributes are described with a name, type and multiplicity. Attributes may be optional. The ‘SubTypeOf’ class defines the inheritance relationship in Express.

Express has the concept of a type which can be either an enumeration, a basic data type (real, number, string,..), a select statement or a container class type (set, list). The select statement can be seen as a ‘one of’ relationship (e.g. a person can drive either a car or a bike at a given time).

Derived from this conceptual model is the UML profile. This is a manual step. The relationship between can be seen as the problem-model, the UML profile is an implementation of this problem in a UML tool using the standardized UML extensibility mechanisms. The UML profile for the Express modeling language looks as the following:

Appendix F-5shows the Express profile for UML manually derived from the Express meta model.

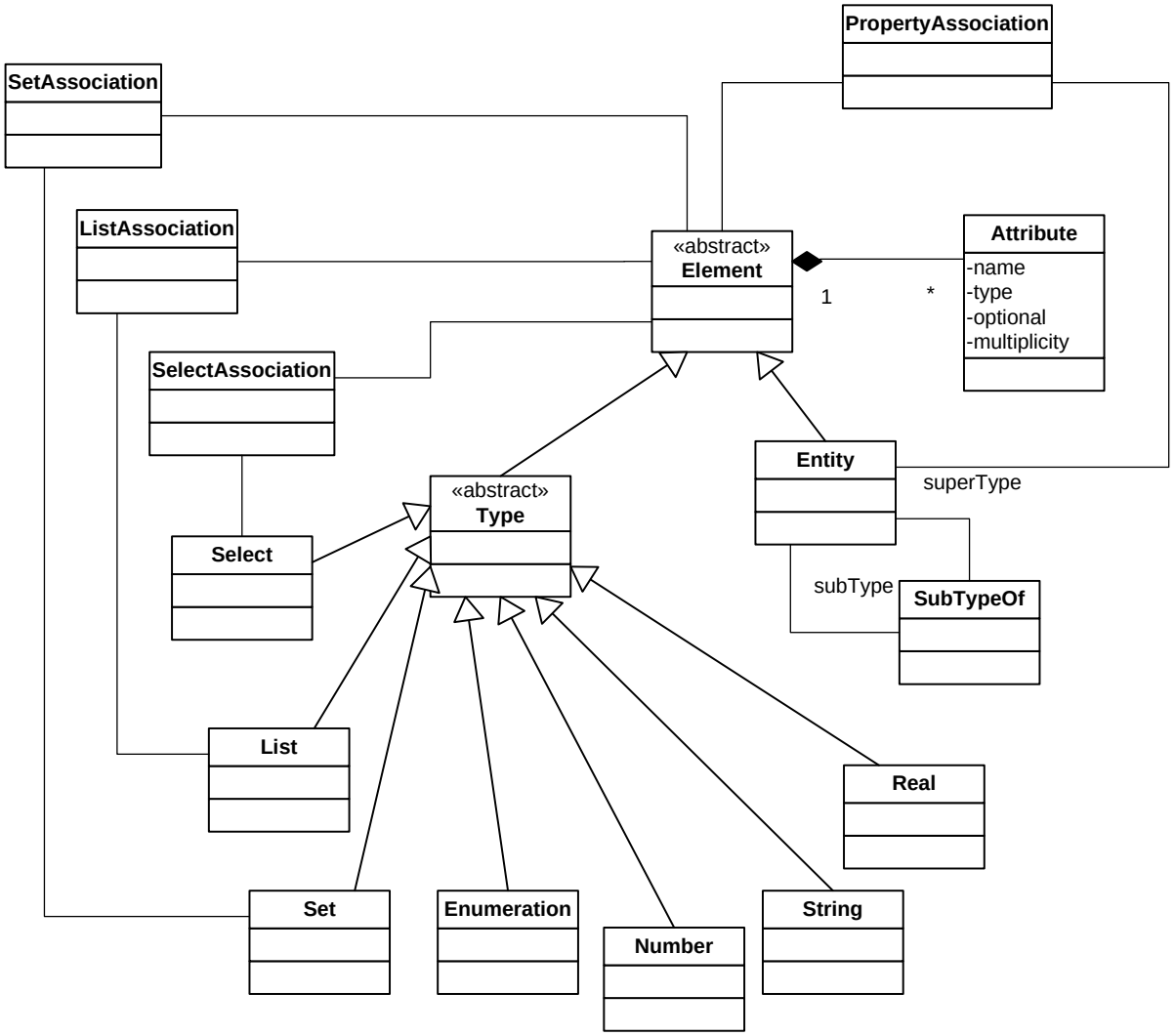


Figure F-4. Excerpt of the Express meta model

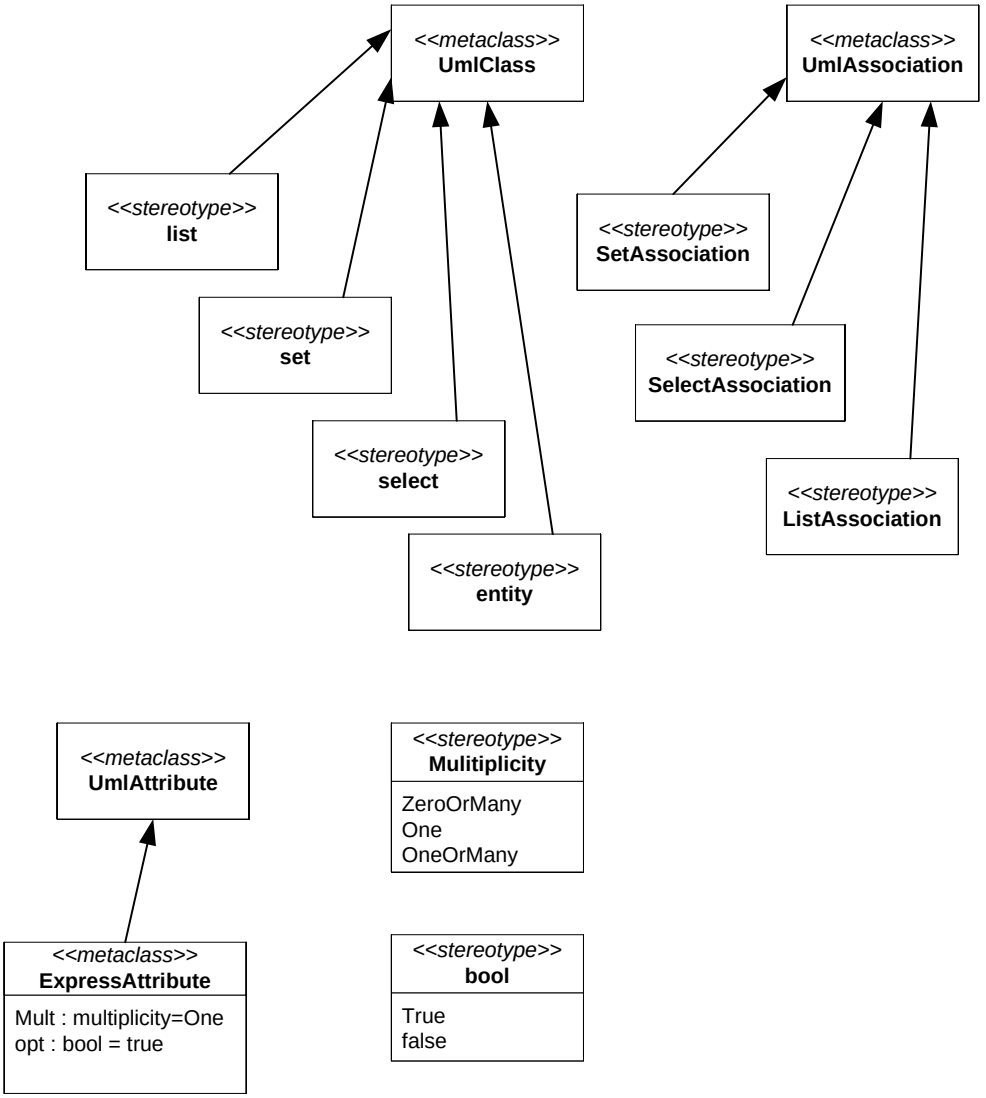


Figure F-5. UML Profile for Express

F.4.2 Converting Express models to UML

The AP233 UML model will be automatically derived from the AP233 Express model. The process is shown in Appendix F-6. Input for the conversion process is the AP233 express model. This will be parsed be a dedicated express parser which takes the express model an produces an informal XML model capturing the express model. This can be used by a XSLT process which takes the XML file and generates a XMI file based on the Express profile for UML.

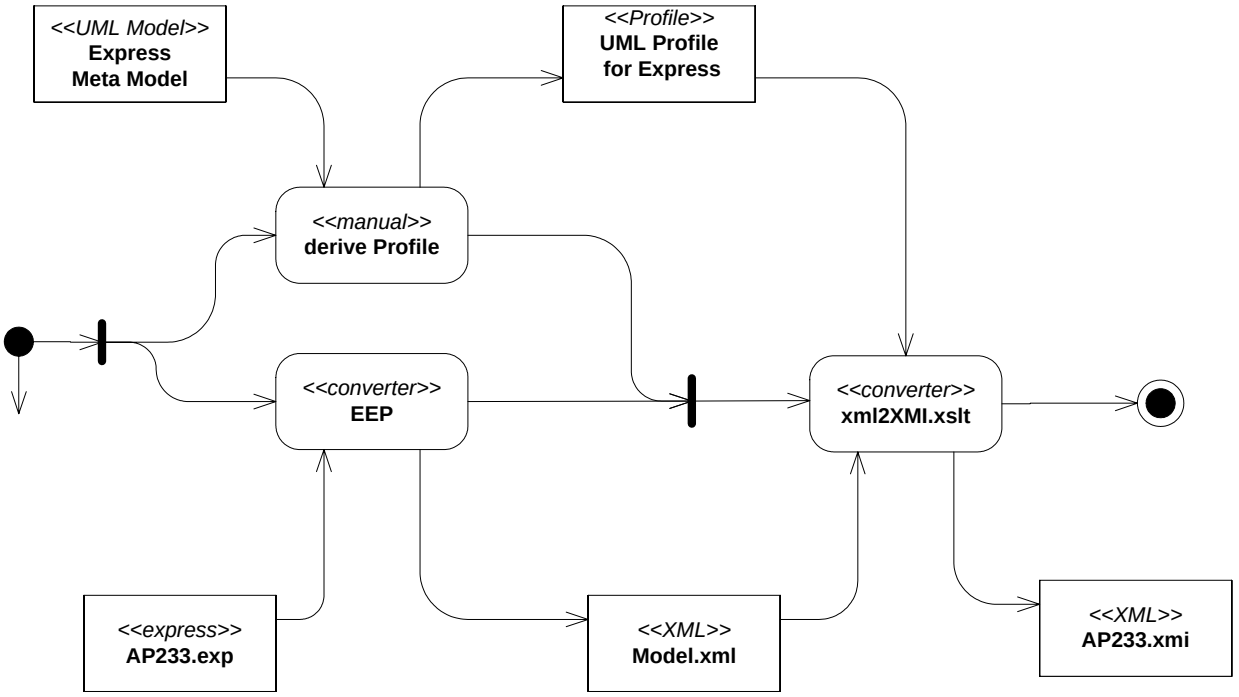


Figure F-6. Activities to derive the AP233 UML model from the Express model

F.5 Model Alignment

F.5.1 SysML Requirements Model

In order to demonstrate the alignment the following shows the requirement module as described in the dedicated section of this specification. It shows the basic elements used to describe requirements and relate them to other modeling elements such as functions, components or test caes. The fundamental element is a requirement with the according parameters (*Requirement*) . A test case (*testCase*) is used to demonstrate the success of the implementation of a given requirement. To check the completeness of the design the requirement is linked to a UML element which satisfies the requirement (*RequirementSatisfaction*). As this model is to define the modeling language it makes reuse of the UML provided modeling elements. For example the RequirementVerification is a UML dependency link.

The main purpose to recall the requirements module of SysML here is to explain the differences of the different models to justify the approach.

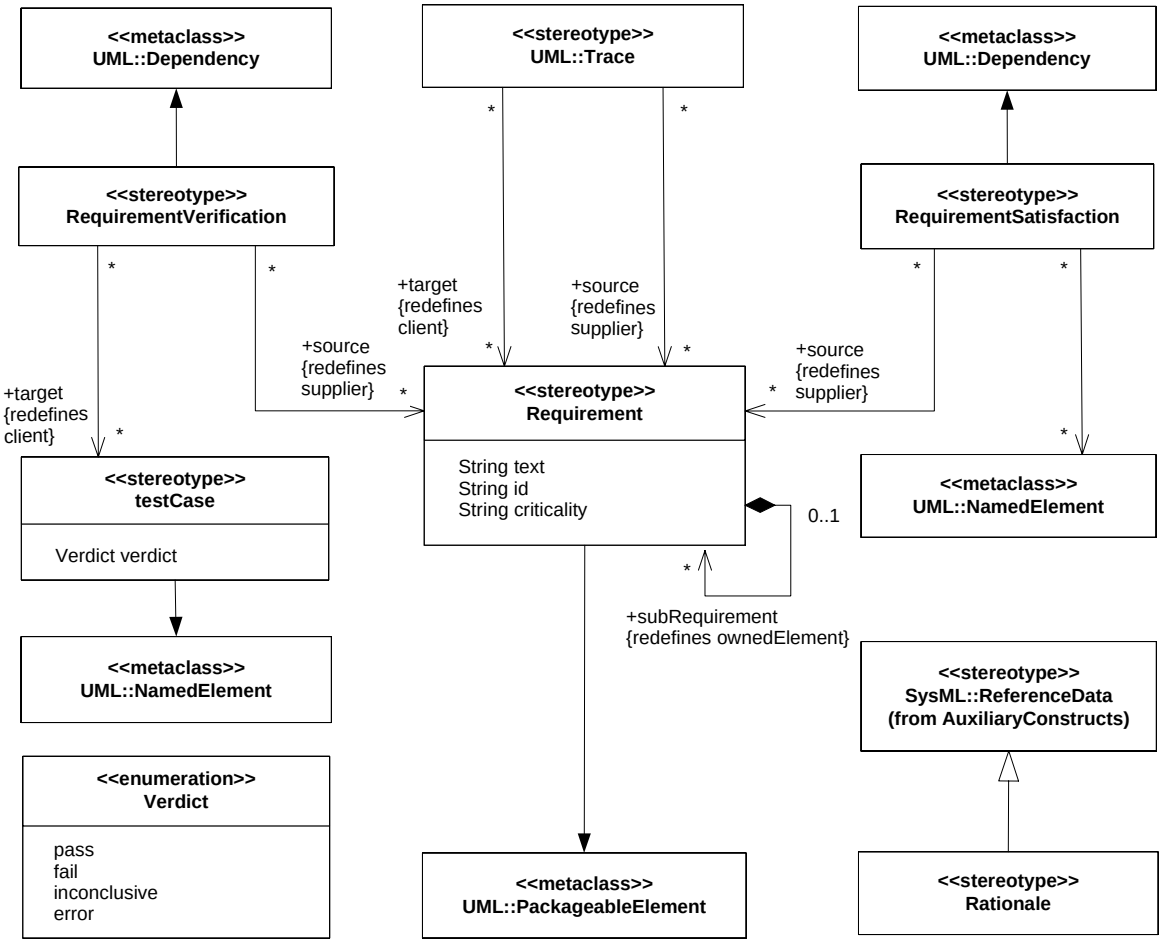


Figure F-7. SysML requirements model

F.5.2 AP233 Requirements Model

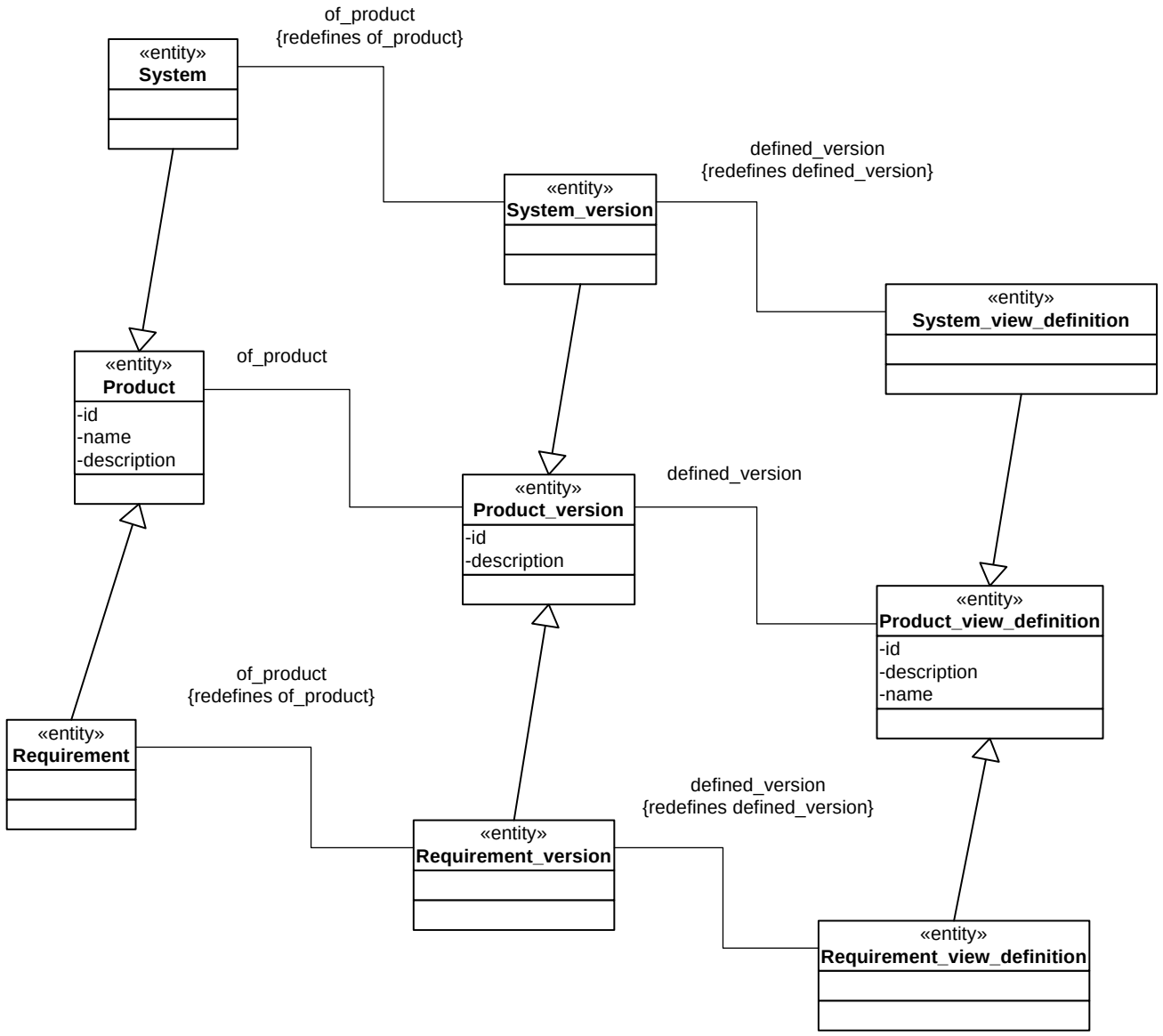


Figure F-8. Basic pattern for AP233

This chapter explains in brief the model of AP233 with a focus on the requirements module and the representation of requirements. The main purpose is to explain the AP233 model and to point out the differences between SysML and AP233 models. Appendix F-8 shows an excerpt of AP233, the basic PDM (Product Data Management) pattern, consisting of *Product*, *Product_version* and *Product_view_definition*. In AP233 the re-use has been done to enable the versioning of the data captured in an AP233 file and to ease the interfacing of PDM systems. In the definition of PDM everything which has to be produced is a product. Therfore, requirements, systems or documents are products.

In order to track the changes for each product along the life cycle configuration control has to be applied. The pattern *Product*, *Product_version* and *Product_view_definition* defines the following:

- *Product*: Defines the identity of a product with an unique identifier, a name and a description
- *Product_version*: Captures the different versions of a product, each version is described by an identifier and a name
- *Product_view_defintion*: Defines the different views in which the product appears, e.g. a diagram or a table

This means that every product is represented in a tree-like manner in an AP233 file.

For each product a tree-like information is given in the AP233 file: For each product an instance of *Product* is the root. Each product may appear in different versions (*Product_version*) and each version may appear and be reference from different views (*Product_view_definitoni*).

Each product which shall be captured in AP233 this pattern has to be re-used. The re-use of this pattern will be done in AP233 via inheritance. Therefore a requirement is represented by the entities *Requirement*, *Requirement_version* and *Requirement_view_definition*. All of them are derived from the according product items. This means a

- *Requirment* 'is a' *Product*
- *Requirment_version* 'is a' *Product_version*'
- *Requirement_view_definition* 'is a' *Produc_view_definition*

The definition of a system is done accordingly which is shown in the diagram. For other items such as connectors or ports this pattern has to be replicated too in the same way.

The assignment of properties to a product is shown in Appendix F-9. Proptiertes are additional descriptions or parameters which have to be attached to a product. Those properties can be for example descriptions, role or context definitions. For a SysML requirement the attributes as defined in Appendix F-7 (id, text and criticality) would be attached as property. As explained per above the *Product_view_definition* is used to attach the detailed information to the requirement.

The top of the diagram shows *Product_view_definition* and its derived classes *Requirement_view_definition* and *System_view_definition*. The *property_assigment_select* is used to attach the properties either to *Produc_view_definition* or to *Tracing_relationship*. *Element_property* describes the property being attached. The select statement *represented_item_select* contains just one choice, the *Element_property*. The class property representation describes the representation for the property attached. The class Representation gives detailed information on the representation occurrence of the property. It can be broken down hierarchically (*Represenation_relationship*) and has links to the different representation items (*Representation_item*).

The different representation items are further detailed in Appendix F-10. It shows the hierarchy of the different represenations: *Data_structure, Element_in_structure, String_representation_item, Binary_representation_item, Document_definition, Mathematical_representation_item, Property_value_with_unit, Descriptive_property_value* and *Plain_text_item*. It is possible to arrange the different representation items in data structures. A data structure consists of elements, an element of a data structure can be for example Binary_representation_item, Document_definition, Mathematical_representation_item, Property_value_with_unit or a Descriptive_property_value.

This represention module of AP233 described here is just a subset of the AP233 representation.

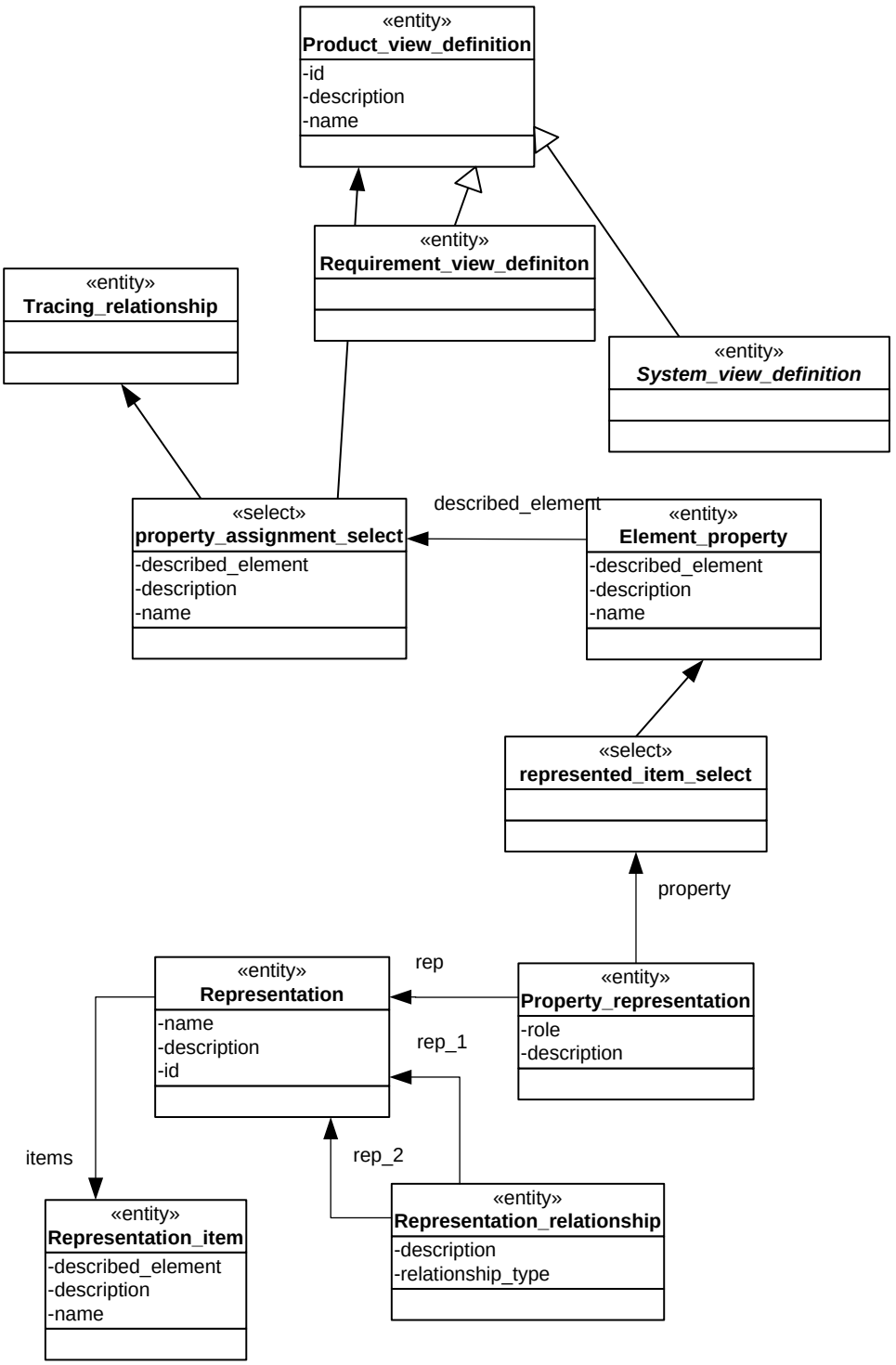


Figure F-9. Property assignment

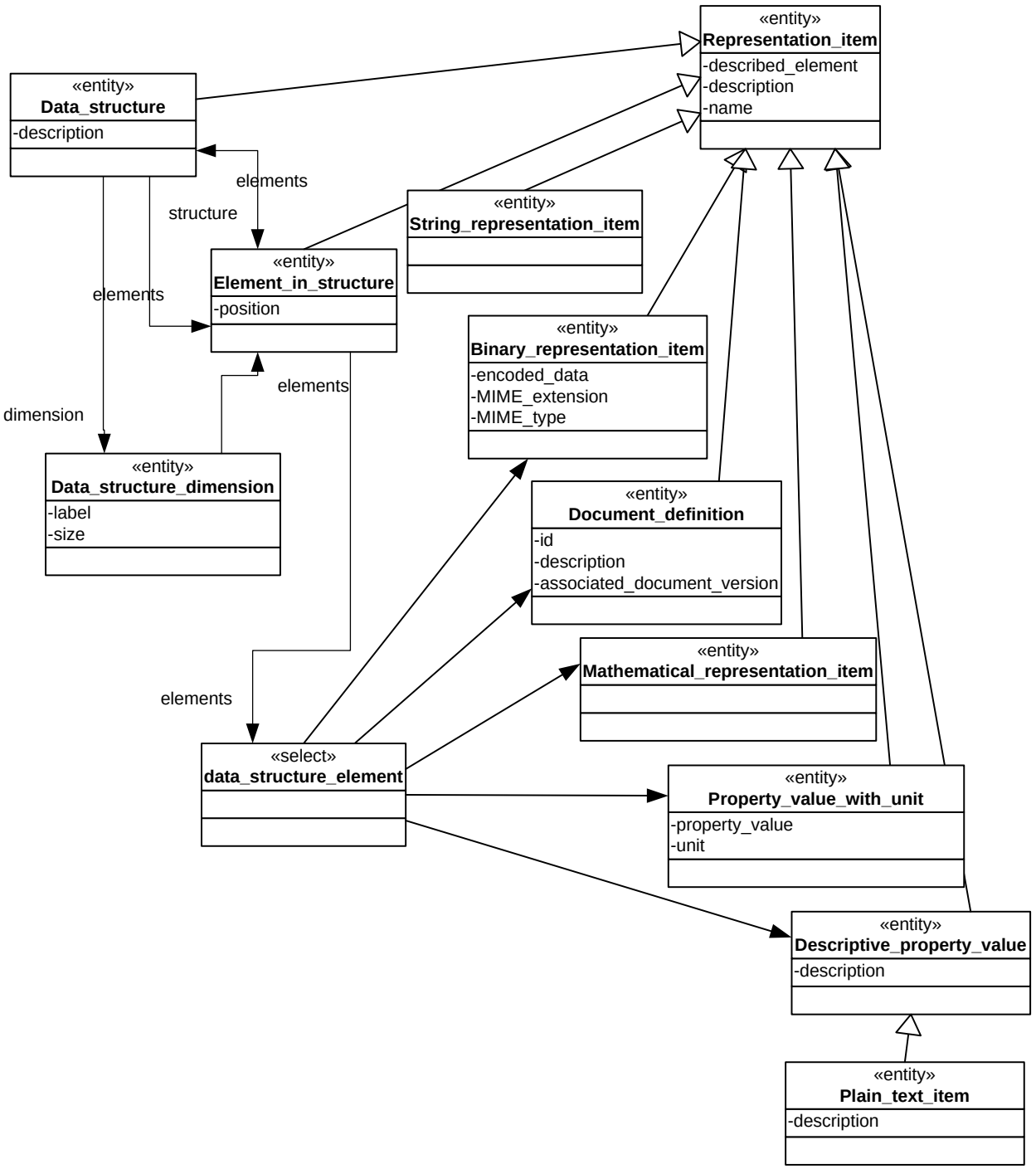


Figure F-10. Representation of properties in AP233

The hierachical breakdown structure of AP233 is shown in Appendix F-11. As explained per above the hierarchical breakdown structure will also be applied to the corresponding view definition items (e.g. *Requirement_view_definition*, *System_view_definition*, ...). The *View_definition_relationship* provides a hierarchial decomposition for *Product_view_definition* items. View_definition_relationship will be further refined for the items derived from *Product_view_definition*.

The class *Requirement_view_definition*, derived from *Product_view_definition* is used to represted the links to the requirements. As explained before it is used to attache the properties to the requirement. It is also used to define the hierarchical breakdown structure for requirements, called traceability links. The traceablility links are established by the item *Tracing_relationship*. It connects derived requirements on different hierarchical levels to each other.

System_view_definition represents items of the system hierarchy. *System_view_definitions* can be hierachically linked to each other (*System_view_definition_relationship*). Important to mention that *System_view_definition* itself is an abstract class and will be further detailled in the subsequent diagrams.

Allocations between requirements and system items are represented by *System_requirement_relationship*.

In diagram Appendix F-12 the different system breakdown hierarchies are shown. The class *System_design* represents system design items and is derived from *System_view_definition*. System design items can be seen as items on the specification (design) level. System design items can be hierarchically decomposed (*System_assembly_relationship*).

Concrete physical items are represented by System_occurrence (It can be seen as a class (*System_design*) and instance (*System_occurence*) releationship for object oriented systems). The items on design levels represent the ‘shall-by’ status of a system. The system occurence items do represent real existing items (‘with serial numbers’).

The physical assembly of a system follows the design items hierarchy, but is not necessarily the same, due to integration constraints. Therfore another breakdown is required to show how the system must be assembled, often called integration tree. Integration tree links are established by *System_occurrence_assembly* items. *System_occurrence* items have a link to *System_view_definition* items to show which part of the system they represent.

All of the items used to represent system hierarchy links (*System_occurrence_relationship*, *System_view_definition_relationship* and *System_assembly_relationship*) are derived from *View_definition_relationship*.

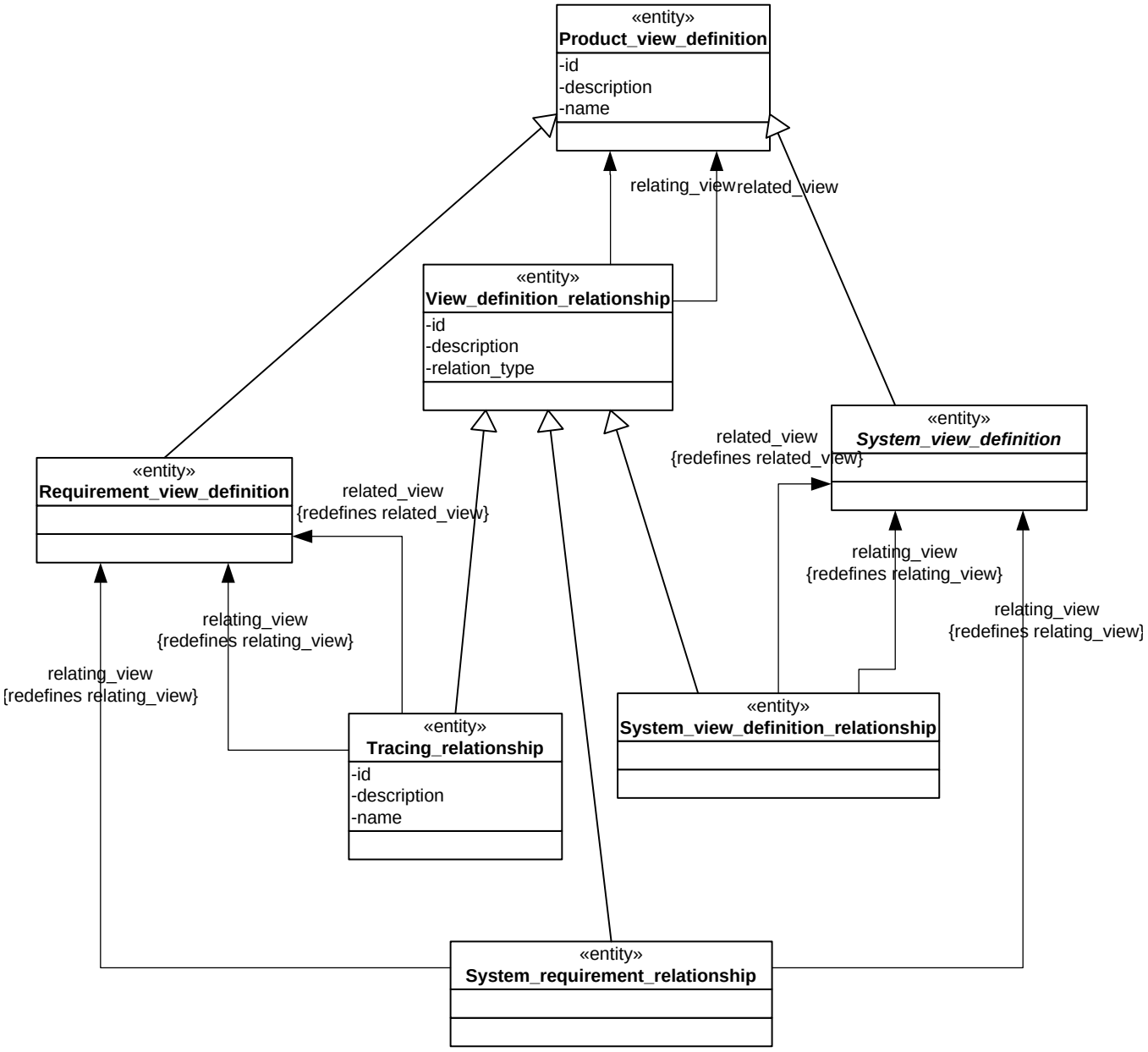


Figure F-11. Breakdown structure in AP233

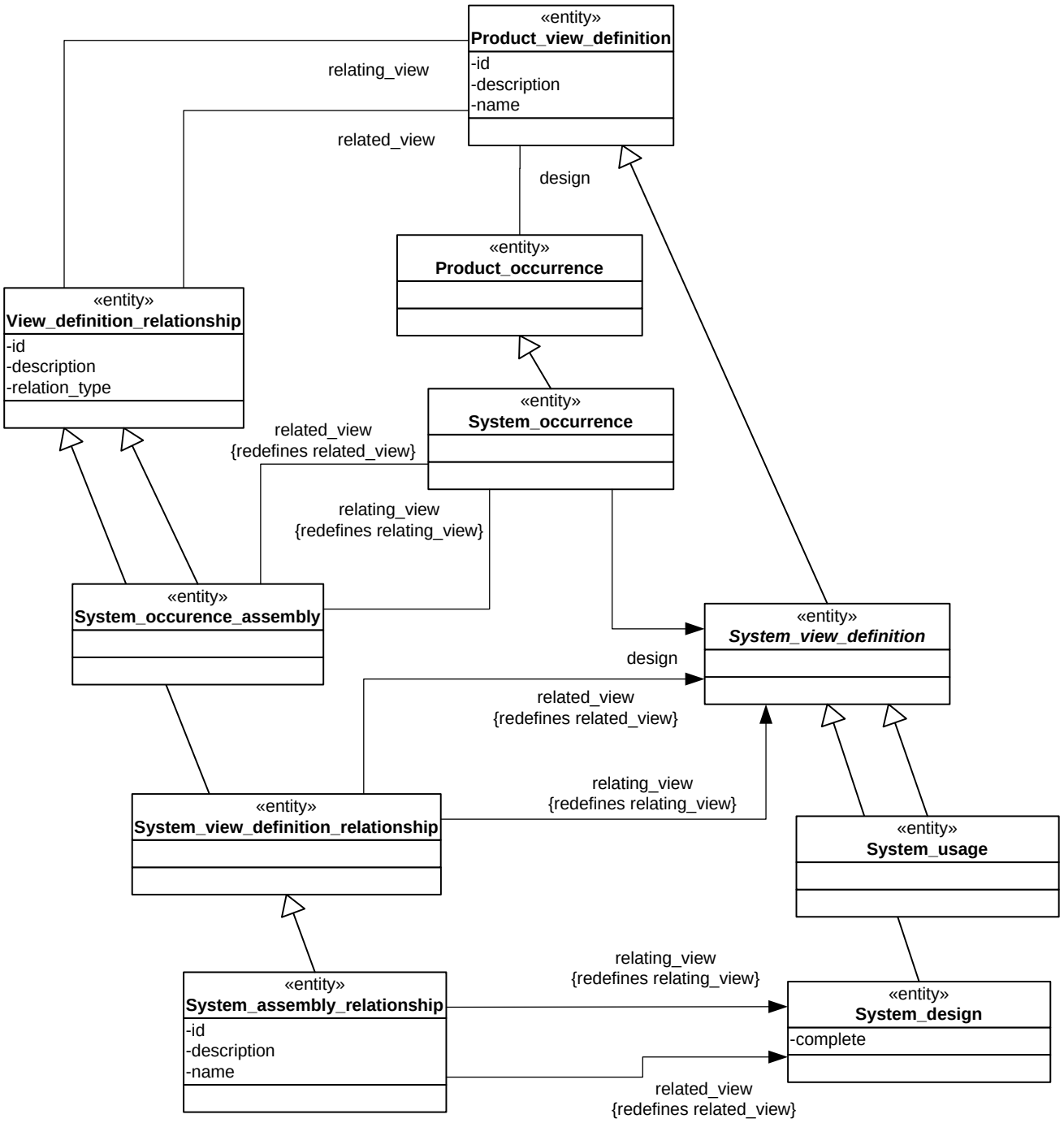


Figure F-12. System breakdown hierarchy

F.5.3 Mapping Module: Requirements

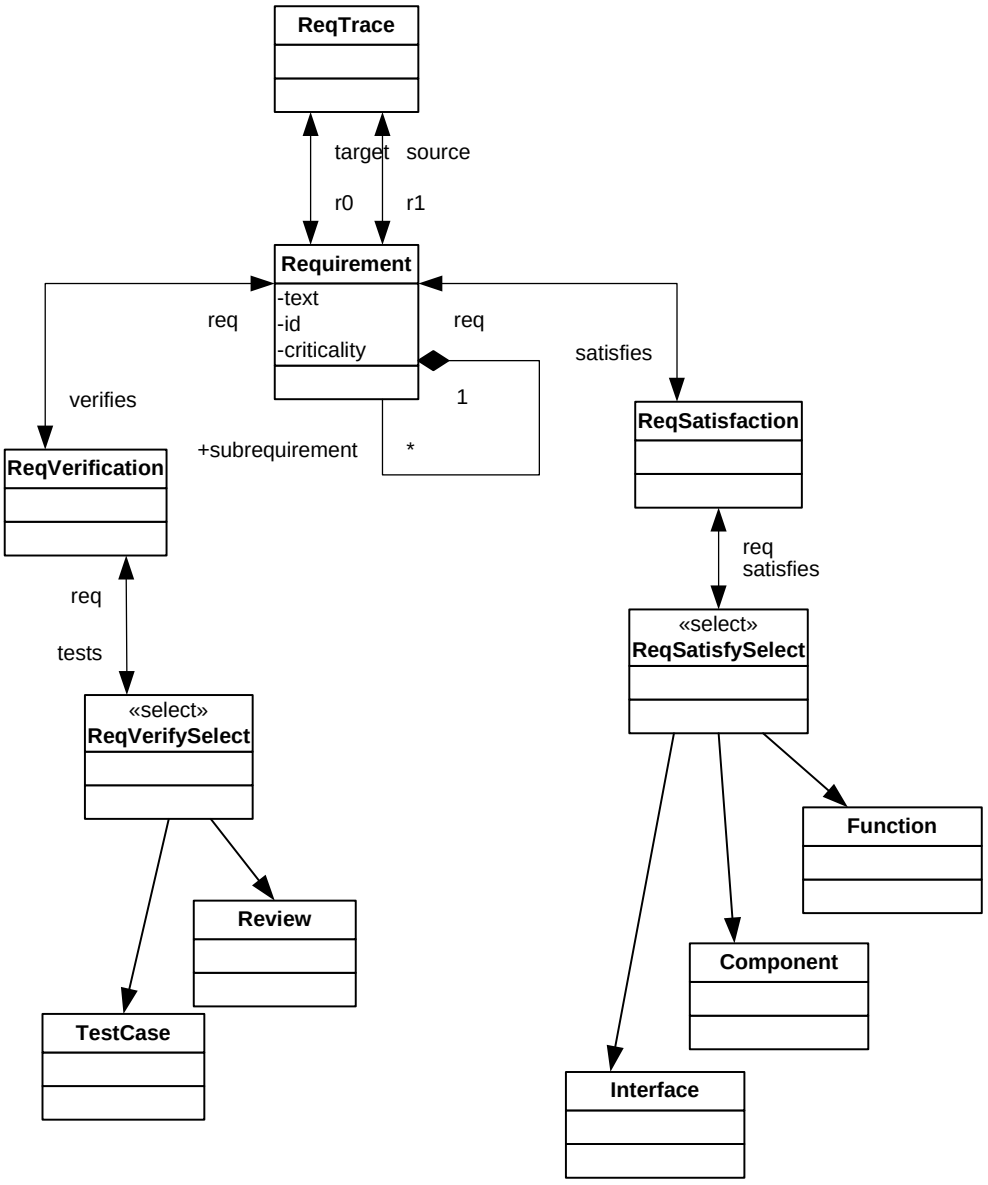


Figure F-13. Mapping Module Requirements

In Appendix F-13 the requirements part of the mapping model is shown. The main difference to the SysML and the AP233 model is the following: The mapping model doesn't address any specify data modeling items and makes not re-use of UML infra- or superstructure definition. Therfore the mapping model can be seen as an independent instance of the MOF. It serves as a kind of requirements model for system engineering conceptst.

Requirement hierarchies are indicated with *ReqTrace*, which connects requirements on different hierachical levels. A requirement can be linked to verification elements (*ReqVerification*) and system modeling elements (*ReqSatisfaction*). On the left hand side two examples for a requirement verifcation are shown (*TestCase*, *Review*). On the reight hand side different elements which can satisfy a requirements (*Function*, *Component*, *Interface*). Both, verification as well as system elements are just a deliberate selection and not exhaustive.

It is import to mention that, in order to ease the mapping, that the ‘select’ statement of Express has been reused. All of the different choices for example for the requirement satisfaction are listed explicitly, similiar to Express.

F.5.4 Mapping between AP233 and SysML

Table 1

#	SysML	Mapping Model	AP233
010	Requirement	Requirement	Requirement Requirement_version Requirement_view_definition
011	text	text	Plain_text_item
012	id	id	Requirement.id
013	criticality	criticality	Plain_text_item
020	Trace	ReqTrace	Tracing_relationship
021	target	r0	relating_view
022	source	r1	related_view
030	RequirementVerification	ReqVerification	
031	target	verifies	
032	source	req	
040	RequirementSatisfaction	ReqSatisfaction	System_requirement_relationship
041	target	satisfies	relating_view
042	source	req	related_view

F.6 Proof of Concept:

ISO10303 STEP application protocols provide a neutral data representation which can be used to collect the data from different tools and capture them in an independent data repository. For this STEP protocols have to provide a generic superset of data items used by the tools. This and some addional concepts required for efficient data modeling STEP application protocols are considered to be ‘complicated’ and difficult to integration into an existing infrastructure. Therefore nowadays

often short-handed XML solutions are preferred instead of integrating STEP.

In order to hide the complexity often a more abstract model (and a related API) is placed on top of STEP protocols. Those APIs do focus on user known concepts and are therefore easier to understand and integrated. But still using the power of STEP as the underlying data model. The mapping model, well mapped to AP233 is high-level model of system engineering concepts.

As both the AP233 and the mapping model are represented in UML they can be used to derive APIs applying the MDA approach to it. In order to obtain an open, robust framework two independent APIs will be generated. One on the AP233 native level and one on the mapping (conceptual) level. This approach also can be used to in cooperate other tools. In Appendix F-14 the different models and APIs are shown.

The resulting API architecture is shown in the Appendix F-15. It shows the different layer of abstraction from the bottom (in this case a STEP file in different representations) to the top different SE tools. The yellow layers show the two different APIs. As examples for tools we see on the top SysML tools, or UML tools which implements selected parts of SysML in terms of profiles and classic SE tools which have nothing in common (with respect to the meta-model) with UML based tools. The mapping model API provides an easy to integrate API based on the conceptual level. In order to integrate it an adaptor has to be written which converts the model (as instance of the meta model) into the representation of the mapping model. For SysML tools this converter can be written based on the mapping model. Based on the mapping between the mapping model and the AP233 model the adaptor can be automatically generated. Finally the AP233 API provides the conversion between the representation as instance of the model to the file representation.

Based on the APIs the mapping between AP233 and SysML can be verified and the success demonstrated. Finally the APIs will be made freely available for further dissemination of the approach.

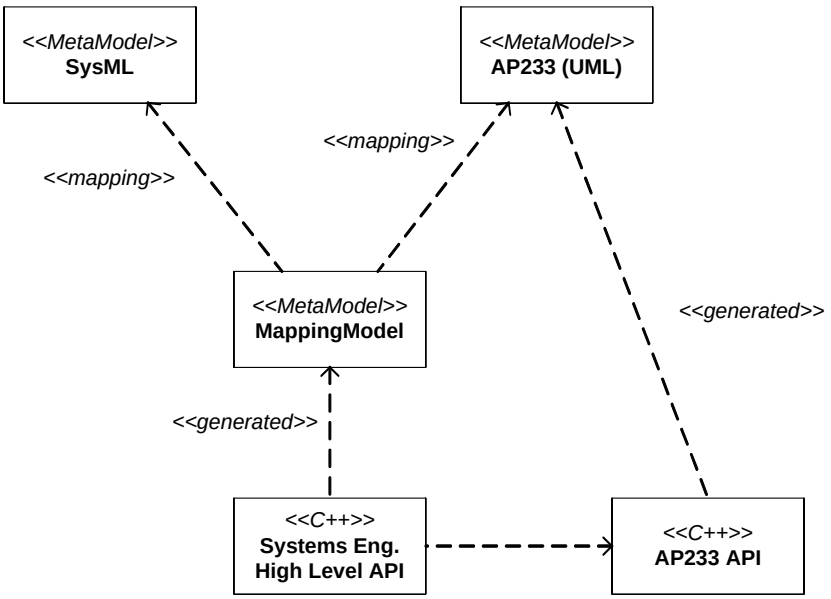


Figure F-14. Model-derived APIs

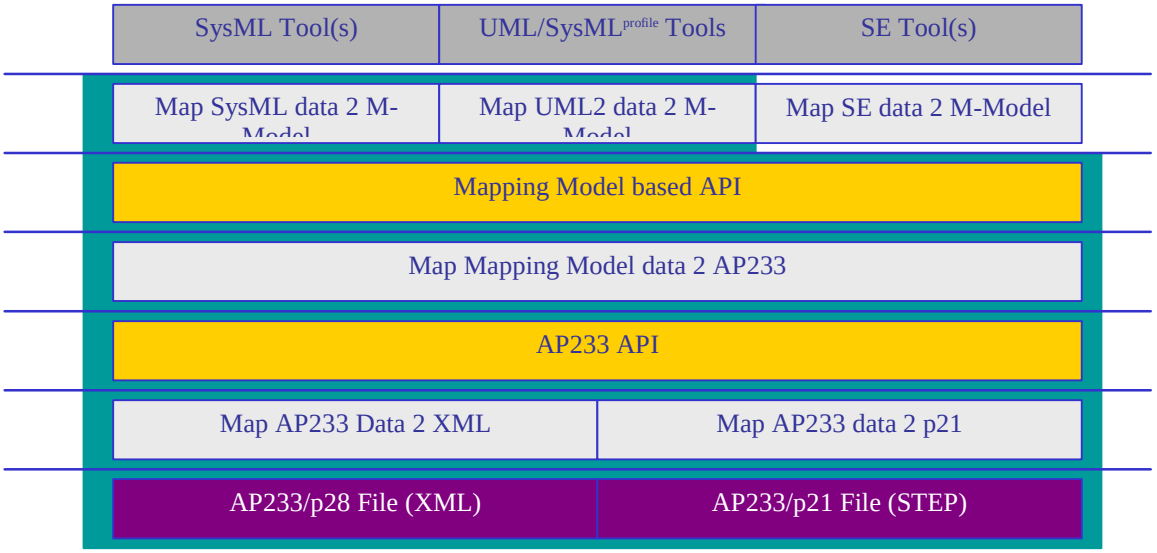


Figure F-15. API abstraction layers

Appendix G. OMG XMI Alignment

Editorial Comment: The architectural alignment of SysML and OMG MOF XMI 2.0 is an ongoing activity since OMG MOF XMI 2.0 is not finalized yet.

The XMI for serializing SysML as an instance of MOF 2.0 according to the rules specified by the proposed MOF2 XMI specification will be made available in a separate document soon after the MOF2 XMI specification is finalized.