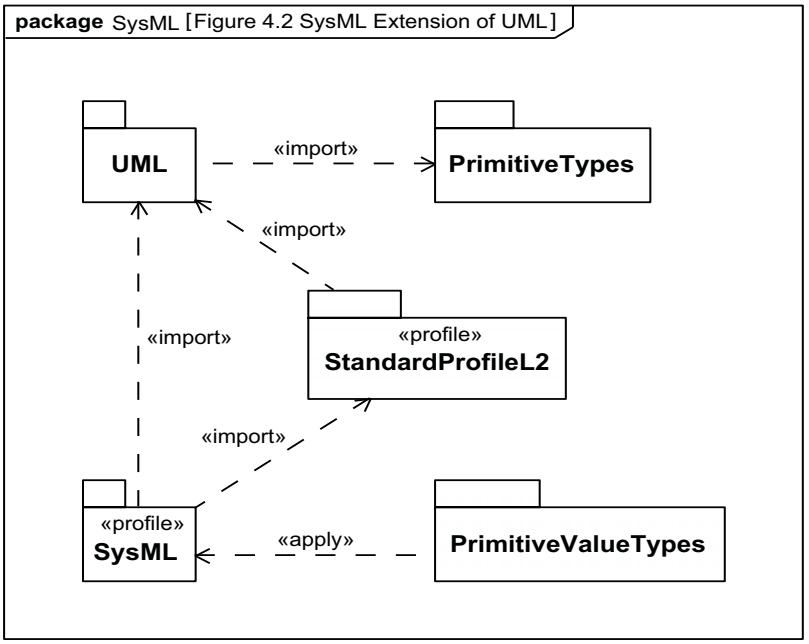


Figure 4.2 - SysML Extension of UML

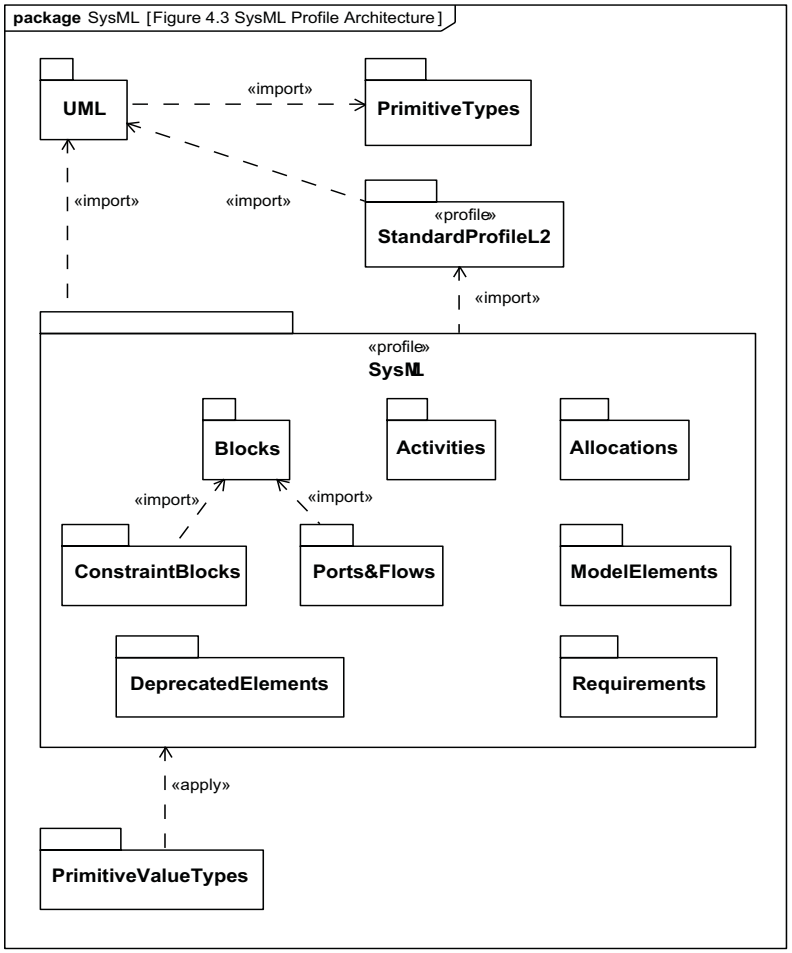


Figure 4.3 - SysML Package Structure

As previously stated, the design approach for SysML is to reuse a subset of UML and create extensions to support the specific concepts needed to satisfy the requirements in the UML for SE RFP. The SysML package structure shown in Figure 4.3 contains a set of packages that correspond to concept areas in SysML that have been extended.

The SysML packages extend UML as follows:

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

4.3 Extension Mechanisms

This specification uses the following mechanisms to define the SysML extensions:

- UML stereotypes
- UML diagram extensions
- Model libraries

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

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

4.4 SysML Diagrams

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

5 Compliance

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

5.1 Compliance with UML Subset (UML4SysML)

The subset of UML required for SysML is specified in terms of a subset of metaclasses imported from the UML 2 metamodel. It is organized in terms of 3 levels of modeling capabilities:

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

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

5.1.1 Compliance Level Contents

The three compliance levels for SysML are declaratively specified in terms of the set of classifiers other than associations available at each compliance level (datatypes, metaclasses, and stereotypes), as shown in Table 5.1, Table 5.2, and Table 5.3.

Table 5.1 - Elements available in SysML Compliance Level 1

Classifier	Kind
UML::AggregationKind, UML::CallConcurrencyKind, UML::MessageKind, UML::MessageSort, UML::ObjectNodeOrderingKind, UML::ParameterDirectionKind, UML::ParameterEffectKind, UML::VisibilityKind	DataType
PrimitiveValueTypes::Boolean, PrimitiveValueTypes::Complex, PrimitiveValueTypes::Integer, PrimitiveValueTypes::Number, PrimitiveValueTypes::Real, PrimitiveValueTypes::String, SysML::Activities::ControlValue, SysML::Ports&Flows::FeatureDirection, SysML::Requirements::VerdictKind	DataType
PrimitiveTypes::Boolean, PrimitiveTypes::Integer, PrimitiveTypes::Real, PrimitiveTypes::String, PrimitiveTypes::UnlimitedNatural	PrimitiveType

Table 5.1 - Elements available in SysML Compliance Level 1

Classifier	Kind
UML::Abstraction, UML::Action, UML::ActionExecutionSpecification, UML::Activity, UML::ActivityEdge, UML::ActivityFinalNode, UML::ActivityGroup, ML::ActivityNode, UML::ActivityParameterNode, UML::Actor, UML::AnyReceiveEvent, UML::Association, UML::Behavior, UML::BehaviorExecutionSpecification, UML::BehavioralFeature, UML::BehavioredClassifier, UML::CallAction, UML::CallBehaviorAction, UML::CallEvent, UML::CallOperationAction, UML::ChangeEvent, UML::Class, UML::Classifier, UML::Comment, UML::ConnectableElement, UML::Constraint, UML::ControlFlow, UML::ControlNode, UML::DataType, UML::Dependency, UML::DeployedArtifact, UML::DeploymentTarget, UML::DestructionOccurrenceSpecification, UML::DirectedRelationship, UML::Element, UML::ElementImport, UML::EncapsulatedClassifier, UML::Enumeration, UML::EnumerationLiteral, UML::Event, UML::ExecutableNode, UML::ExecutionOccurrenceSpecification, UML::ExecutionSpecification, UML::Expression, UML::Extend, UML::ExtensionPoint, UML::Feature, UML::FinalNode, UML::FunctionBehavior, UML::GeneralOrdering, UML::Generalization, UML::Include, UML::InitialNode, UML::InputPin, UML::InstanceSpecification, UML::InstanceValue, UML::Interaction, UML::InteractionFragment, UML::Interface, UML::InterfaceRealization, UML::InvocationAction, UML::Lifeline, UML::LiteralBoolean, UML::LiteralInteger, UML::LiteralNull, UML::LiteralReal, UML::LiteralSpecification, UML::LiteralString, UML::LiteralUnlimitedNatural, UML::Message, UML::MessageEnd, ML::MessageEvent, UML::MessageOccurrenceSpecification, UML::MultiplicityElement, UML::NamedElement, UML::Namespace, UML::ObjectFlow, UML::ObjectNode, UML::OccurrenceSpecification, UML::OpaqueAction, UML::OpaqueBehavior, UML::OpaqueExpression, UML::Operation, UML::OutputPin, UML::Package, UML::PackageImport, UML::PackageMerge, UML::PackageableElement, UML::Parameter, UML::ParameterableElement, UML::Pin, UML::PrimitiveType, UML::Property, UML::Realization, UML::Reception, UML::RedefinableElement, UML::Relationship, UML::SendSignalAction, UML::Signal, UML::SignalEvent, UML::Slot, UML::StateInvariant, UML::StructuralFeature, UML::StructuredClassifier, UML::Substitution, UML::Type, UML::TypedElement, UML::Usage, UML::UseCase, UML::ValuePin, UML::ValueSpecification	Metaclass
SysML::Activities::Continuous, SysML::Activities::ControlOperator, SysML::Activities::Discrete, SysML::Activities::NoBuffer, SysML::Activities::Optional, SysML::Activities::Overwrite, SysML::Activities::Rate, SysML::Allocations::Allocate, SysML::Allocations::Allocated, SysML::Blocks::Block, SysML::Blocks::DistributedProperty, SysML::Blocks::ParticipantProperty, SysML::Blocks::PropertySpecificType, SysML::Blocks::QuantityKind, SysML::Blocks::Unit, SysML::Blocks::ValueType, SysML::ConstraintBlocks::ConstraintBlock, SysML::ConstraintBlocks::ConstraintProperty, SysML::ModelElements::Conform, SysML::ModelElements::Problem, SysML::ModelElements::Rationale, SysML::ModelElements::View, SysML::ModelElements::Viewpoint, SysML::Ports&Flows::ChangeStructuralFeatureEvent, SysML::Ports&Flows::DirectedFeature, SysML::Ports&Flows::FlowProperty, SysML::Ports&Flows::InterfaceBlock, SysML::Requirements::Copy, SysML::Requirements::DeriveReqt, SysML::Requirements::Requirement, SysML::Requirements::RequirementRelated, SysML::Requirements::Satisfy, SysML::Requirements::TestCase, SysML::Requirements::Verify, StandardProfileL2::Metaclass, StandardProfileL2::Trace	Stereotype

Table 5.2 - Elements available in SysML Compliance Level 2

Classifier	Kind
UML::ConnectorKind, UML::InteractionOperatorKind, UML::PseudostateKind, UML::TransitionKind	DataType

Table 5.2 - Elements available in SysML Compliance Level 2

Classifier	Kind
UML::ActionInputPin, UML::ActivityPartition, UML::AddStructuralFeatureValueAction, UML::AddVariableValueAction, UML::BroadcastSignalAction, UML::CentralBufferNode, UML::Clause, UML::ClearAssociationAction, UML::ClearStructuralFeatureAction, UML::ClearVariableAction, UML::CombinedFragment, UML::ConditionalNode, UML::ConnectionPointReference, UML::Connector, UML::ConnectorEnd (except for Constraint [3]), UML::ConsiderIgnoreFragment, UML::Continuation, UML::CreateLinkAction, UML::CreateObjectAction, UML::DecisionNode, UML::DestroyLinkAction, UML::DestroyObjectAction, UML::Duration, UML::DurationConstraint, UML::DurationInterval, UML::DurationObservation, UML::Extension, UML::ExtensionEnd, UML::FinalState, UML::FlowFinalNode, UML::ForkNode, UML::Gate, UML::Image, UML::InteractionConstraint, UML::InteractionOperand, UML::InteractionUse, UML::Interval, UML::IntervalConstraint, UML::JoinNode, UML::LinkAction, UML::LinkEndCreationData, UML::LinkEndData, UML::LinkEndDestructionData, UML::LoopNode, UML::MergeNode, UML::Observation, UML::PartDecomposition, UML::Port, UML::Profile, UML::ProfileApplication, UML::Pseudostate, UML::RaiseExceptionAction, UML::ReadLinkAction, UML::ReadSelfAction, UML::ReadStructuralFeatureAction, UML::ReadVariableAction, UML::Region, UML::RemoveStructuralFeatureValueAction, UML::RemoveVariableValueAction, UML::SendObjectAction, UML::SequenceNode, UML::State, UML::StateMachine, UML::Stereotype, UML::StructuralFeatureAction, UML::StructuredActivityNode, UML::TestIdentityAction, UML::TimeConstraint, UML::TimeEvent, UML::TimeExpression, UML::TimeInterval, UML::TimeObservation, UML::Transition, UML::ValueSpecificationAction, UML::Variable, UML::VariableAction, UML::Vertex, UML::WriteLinkAction, UML::WriteStructuralFeatureAction, UML::WriteVariableAction	Metaclass
SysML::Allocations::AllocateActivityPartition, SysML::Blocks::BindingConnector, SysML::Blocks::ConnectorProperty, SysML::Blocks::NestedConnectorEnd, SysML::Ports&Flows::FullPort, SysML::Ports&Flows::InvocationOnNestedPortAction, SysML::Ports&Flows::ProxyPort, SysML::Ports&Flows::TriggerOnNestedPort	Stereotype

Table 5.3 - Elements available in SysML Compliance Level 3

Classifier	Kind
UML::AcceptCallAction, UML::AcceptEventAction, UML::AssociationClass, UML::CreateLinkObjectAction, UML::DataStoreNode, UML::GeneralizationSet, UML::InformationFlow, UML::InformationItem, UML::InterruptibleActivityRegion, UML::Model, UML::ParameterSet, UML::ReadExtentAction, UML::ReadIsClassifiedObjectAction, UML::ReadLinkObjectEndAction, UML::ReclassifyObjectAction, UML::ReduceAction, UML::ReplyAction, UML::StartClassifierBehaviorAction, UML::StartObjectBehaviorAction,UML::UnmarshallAction	Metaclass
SysML::Activities::Probability, SysML::Ports&Flows::AcceptChangeStructuralFeatureEventAction, SysML::Ports&Flows::ItemFlow	Stereotype

5.2 Compliance with SysML Extensions

In addition to UML, further units of compliance for SysML are the subpackages of the SysML profile, except for DeprecatedElements, which is not a compliance unit. These packages are given by Clause 4, “Language Architecture.”

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

Table 5.4 - SysML package dependence on SysML compliance levels

SysML Package	SysML Compliance Level
Activities (without Probability)	Level 2
Activities (with Probability)	Level 3
Allocations	Level 2
Blocks	Level 2
ConstraintBlocks	Level 2
ModelElements	Level 1
Ports&Flows (without ItemFlow)	Level 2
Ports&Flows (with ItemFlow)	Level 3
Requirements	Level 1

5.3 Meaning of Compliance

An implementation of SysML must comply with both the subset of UML4SysML and the SysML extensions as described above. The meaning of compliance in SysML is based on the UML definition of compliance, excluding diagram interchange. (Note that diagram interchange is different from model interchange that is included in SysML—refer to XMI in Annex E, “Model Interchange.”)

Compliance can be defined in terms of the following:

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

Compliance for a given level can be expressed as:

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

The fullest compliance response is “YES,” which indicates full realization of *all language units/stereotypes* that are defined for that compliance level. This also implies full realization of all language units/stereotypes in all the levels below that level. “Full realization” for a language unit at a given level means supporting the *complete set of modeling concepts* defined for that language unit *at that level*. A compliance response of “PARTIAL” indicates partial realization and requires a feature support statement detailing which concepts are supported. These statements should reference either the

language unit and metaclass, or profile package and stereotype for abstract syntax, or a diagram element for concrete syntax (the diagram elements in SysML are given unique names to allow unambiguous references). Finally, a response of "NO" indicates that none of the language units/stereotypes in this compliance point is realized.

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

Table 5.5 - Example Compliance Statement

Compliance Summary		
Compliance level	**Abstract Syntax**	**Concrete Syntax**
SysML Compliance Level 1	YES	YES
SysML Compliance Level 2	PARTIAL	YES
SysML Compliance Level 3	NO	NO
Activities (without Probability)	YES	YES
Activities (with Probability)	NO	NO
Allocations	PARTIAL	PARTIAL
Blocks	YES	YES
ConstraintBlocks	YES	YES
ModelElements	YES	YES
Ports&Flows (without ItemFlow)	YES	YES
Ports&Flows (with ItemFlow)	NO	NO
Requirements	YES	YES

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

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

Table 5.6 - Example feature support statement

Feature Support Statement				
Compliance Level/	**Detail**	**Abstract Syntax**	**Concrete Syntax**	**Semantics**
SysML Compliance Level 2	StateMachines::BehaviorStateMachines	Note (1)	Note(1)	Note (2)
SysML::Blocks	Block	YES	Note (3)	

NOTE(s):

1. States and state machines are limited to a single region. Shallow history pseudostates are not supported.
2. Only FIFO queueing allowed in event pool.
3. Does not include Blocks::StructuredCompartment notation.

6 Language Formalism

6.1 Levels of Formalism

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

6.2 Clause Structure

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

6.2.1 Overview

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

6.2.2 Diagram Elements

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

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

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

6.2.3 UML Extensions

This sub clause specifies the SysML extensions to UML in terms of diagram extensions and semantic extensions. Diagram extensions are included when the concrete syntax uses notation other than the standard stereotype notation as defined in the Profiles & Model Libraries clause. Semantic extensions consist of stereotype and model library extensions. Stereotype extensions always include the abstract syntax that identifies which metaclasses a stereotype extends. Each stereotype includes a general description with a definition and semantics, along with stereotype properties (attributes), and constraints. Each constraint consists of a textual description and may be followed by a formal constraint expressed in Object Constraint Language (OCL). If there is any ambiguity between the two, the OCL statement of the constraint takes precedence. The model libraries are defined as subclasses of existing metaclasses.

6.2.4 Usage Examples

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

6.3 Conventions and Typography

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

- When referring to stereotypes, metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are used.
- No visibilities are presented in the diagrams, since all elements are public.
- If a sub clause is not applicable, it is not included, except for the top-level sub clauses outlined in “Clause Structure” on page 19.
- Stereotype, metaclass, and metassociation names: initial embedded capitals are used (e.g., “ModelElement,” “ElementReference”).
- Boolean metaattribute names: always start with “is” (e.g., “isComposite”).
- Enumeration types: always end with “Kind” (e.g., “DependencyKind”).

Part II - Structural Constructs

This Part defines the static and structural constructs used in SysML structure diagrams, including the package diagram, block definition diagram, internal block diagram, and parametric diagram. This Part includes the following clauses:

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

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

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

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

7 Model Elements

7.1 Overview

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

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

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

SysML has extended the concept of view and viewpoint from UML to be consistent with the *IEEE 1471 standard*. In particular, a viewpoint is a specification of rules for constructing a view to address a set of stakeholder concerns, and the view is intended to represent the system from this viewpoint. This enables stakeholders to specify aspects of the system model that are important to them from their viewpoint, and then represent those aspects of the system in a specific view. Typical examples may include an operational, manufacturing, or security view/viewpoint.

7.2 Diagram Elements

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

Table 7.1 - Graphical nodes defined by ModelElements package

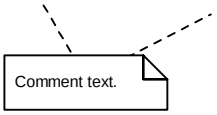
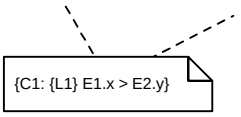
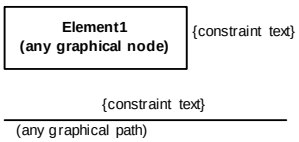
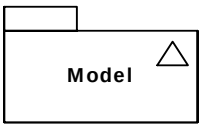
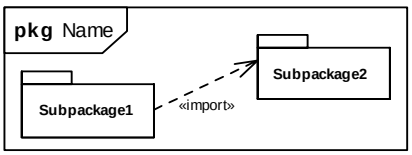
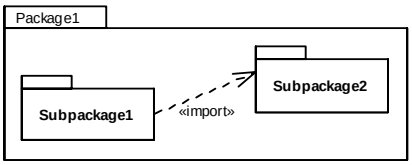
Element Name	Concrete Syntax Example	Abstract Syntax Reference
Comment		**UML4SysML::Comment**
ConstraintNote		**UML4SysML::Constraint**
ConstraintTextualNote		**UML4SysML::Constraint**
Model		**UML4SysML::Model**
PackageDiagram		**UML4SysML::Package**
PackageWith NameInTab		**UML4SysML::Package**

Table 7.1 - Graphical nodes defined by ModelElements package

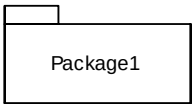
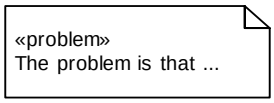
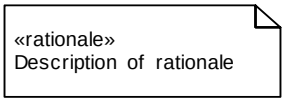
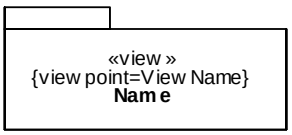
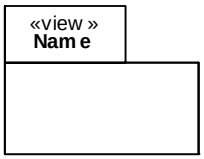
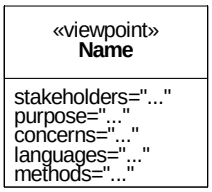
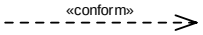
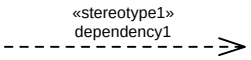
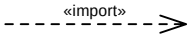
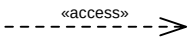
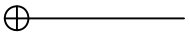
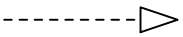
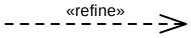
Element Name	Concrete Syntax Example	Abstract Syntax Reference
PackageWith NameInside		**UML4SysML::Package**
Problem		**SysML::ModelElements::Problem**
Rationale		**SysML::ModelElements::Rationale**
ViewWith NameInside		**SysML::ModelElements::View**
ViewWith NameInTab		**SysML::ModelElements::View**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

Table 7.2 - Graphical paths defined by ModelElements package

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

7.3 UML Extensions

7.3.1 Diagram Extensions

7.3.1.1 UML Diagram Elements not Included in SysML

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

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

7.3.2 Stereotypes

Package ModelElements

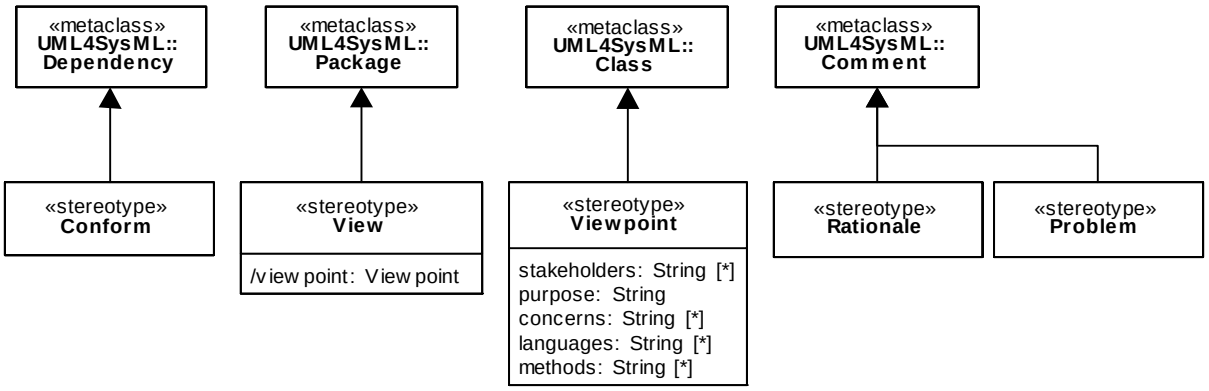


Figure 7.1 - Stereotypes defined in package ModelElements

7.3.2.1 Conform

Description

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

Constraints

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

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

7.3.2.2 Problem

Description

A Problem documents a deficiency, limitation, or failure of one or more model elements to satisfy a requirement or need, or other undesired outcome. It may be used to capture problems identified during analysis, design, verification, or manufacture and associate the problem with the relevant model elements. Problem is a stereotype of comment and may be attached to any other model element in the same manner as a comment.

7.3.2.3 Rationale

Description

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

7.3.2.4 View

Description

A View is a representation of a whole system or subsystem from the perspective of a single viewpoint. Views are allowed to import other elements including other packages and other views that conform to the viewpoint.

Attributes

- /viewpoint: Viewpoint
  The viewpoint for this View, derived from the supplier of the «conform» dependency whose client is this View.

Constraints

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

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

7.3.2.5 Viewpoint

Description

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

Attributes

- stakeholders: String [*]
  Set of stakeholders.
- purpose: String
  The purpose addresses the stakeholder concerns.

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

Constraints

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

[2] The property ownedOperation must be empty.

[3] The property ownedAttribute must be empty.

7.4 Usage Examples

See Figure C.27 in Annex C for a View/Viewpoint example.

Figure 7.2 shows examples of Rationale and Problem elements.

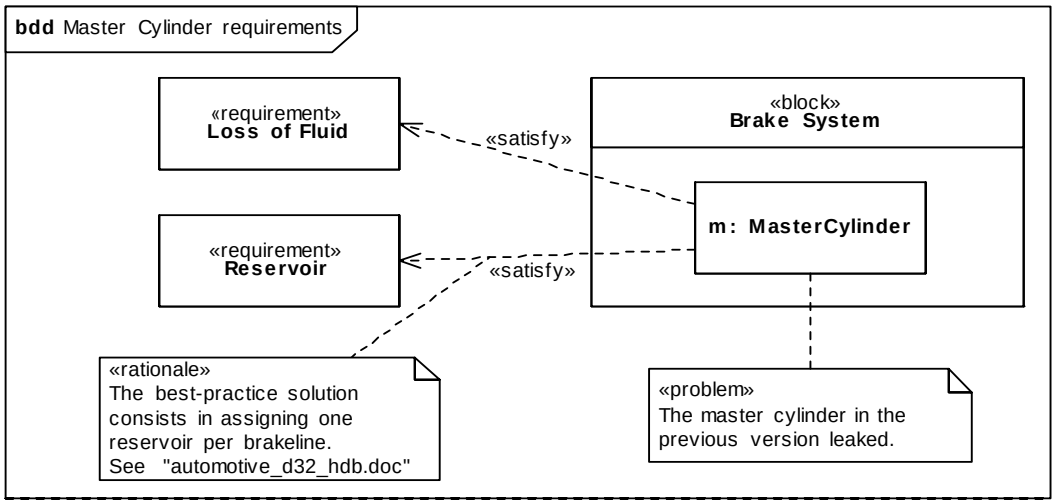


Figure 7.2 - Rationale and Problem examples

8 Blocks

8.1 Overview

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

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

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

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

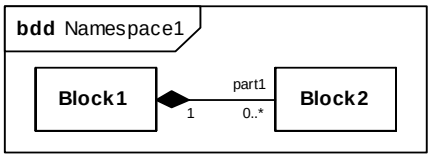
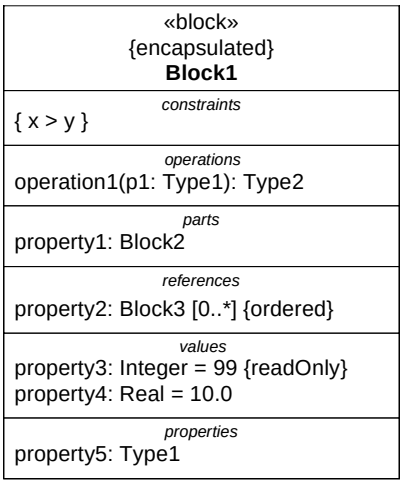
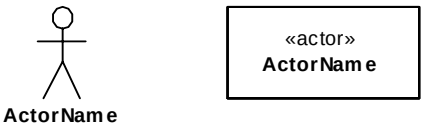
Blocks may also specify operations or other features that describe the behavior of a system. Except for operations, this clause deals strictly with the definition of properties to describe the state of a system at any given point in time, including relations between elements that define its structure.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

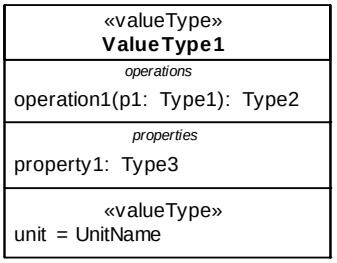
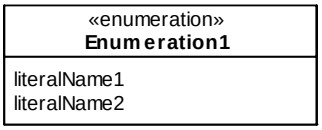
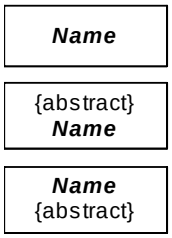
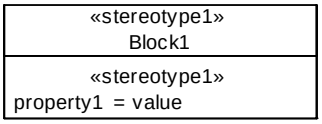
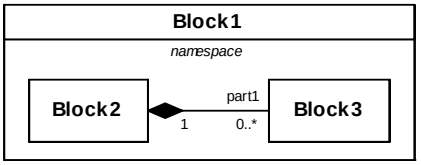
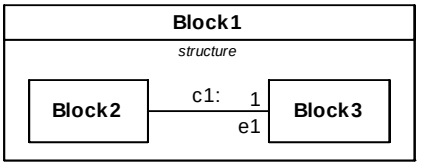
SysML blocks are based on UML classes as extended by UML composite structures. Some capabilities available for UML classes, such as more specialized forms of associations, have been excluded from SysML blocks to simplify the language. SysML blocks always include an ability to define internal connectors,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

8.2 Diagram Elements

8.2.1 Block Definition Diagram

Table 8.1 - Graphical nodes defined in Block Definition diagrams

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

Element Name	Concrete Syntax Example	Abstract syntax Reference
ValueType		**SysML::Blocks::ValueType**
Enumeration		**UML4SysML::Enumeration**
AbstractDefinition		**UML4SysML::Classifier with isAbstract equal true**
StereotypeProperty Compartment		**UML4SysML::Stereotype**
Namespace Compartment		**SysML::Blocks::Block**
Structure Compartment		**SysML::Blocks::Block**

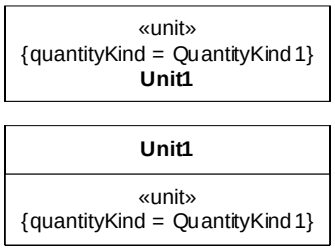
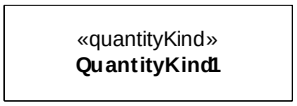
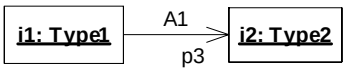
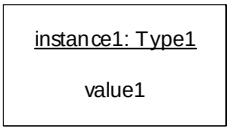
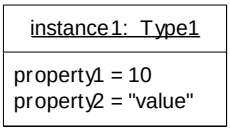
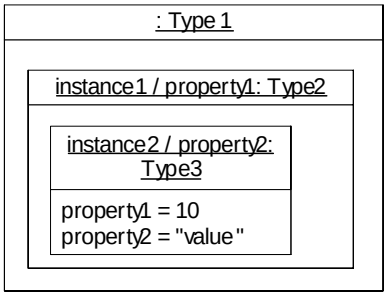
Element Name	Concrete Syntax Example	Abstract syntax Reference
Unit		**SysML::Blocks::Unit**
QuantityKind		**SysML::Blocks::QuantityKind**
InstanceSpecification		**UML4SysML:: InstanceSpecification**
InstanceSpecification		**UML4SysML:: InstanceSpecification**
InstanceSpecification		**UML4SysML:: InstanceSpecification**
InstanceSpecification		**UML4SysML: InstanceSpecification**

Table 8.2 - Graphical paths defined by in Block Definition diagrams

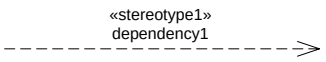
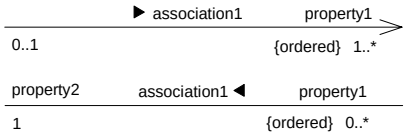
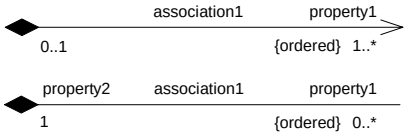
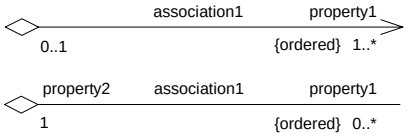
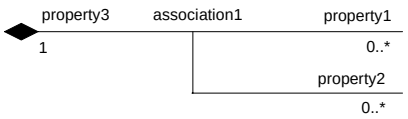
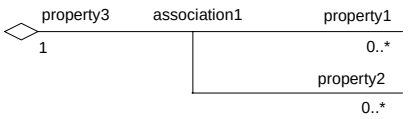
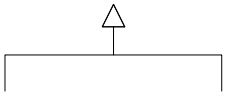
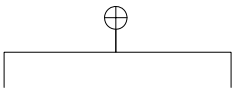
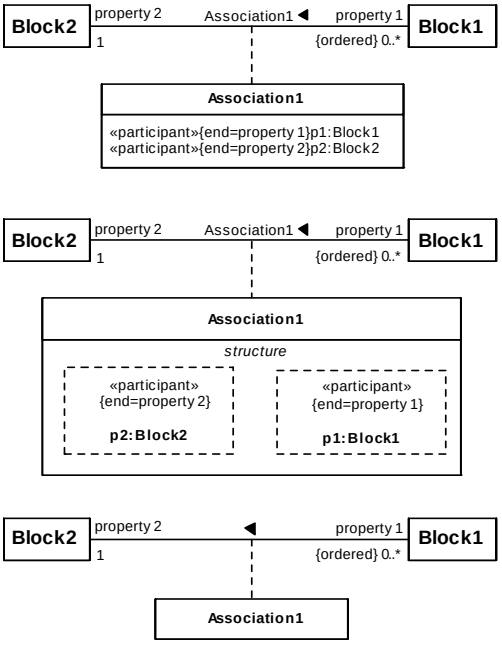
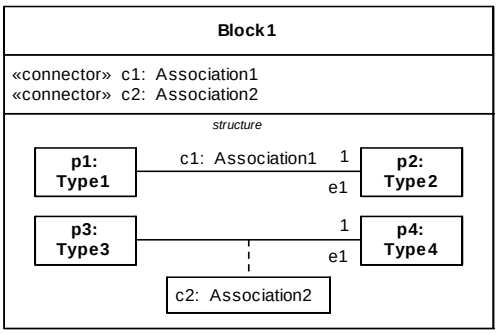
Element Name	Concrete Syntax Example	Abstract syntax Reference
Dependency		**UML4SysML::Dependency**
ReferenceAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = none**
PartAssociation		**UML4SysML::Association and UML4SysML::Property with aggregationKind = composite**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

Table 8.2 - Graphical paths defined by in Block Definition diagrams

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

8.2.2 Internal Block Diagram

Table 8.3 - Graphical nodes defined in Internal Block diagrams

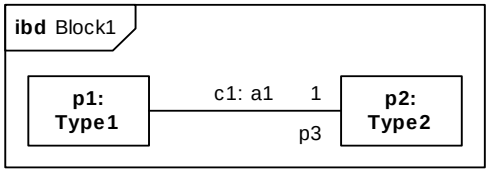
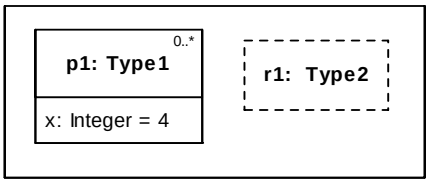
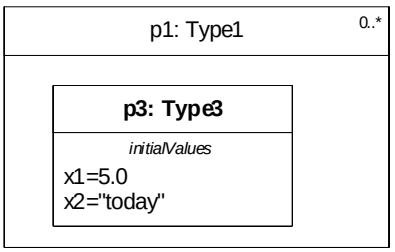
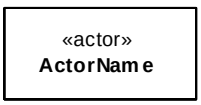
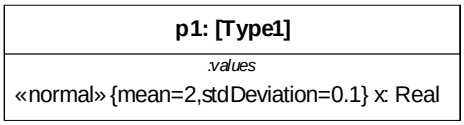
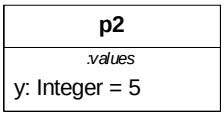
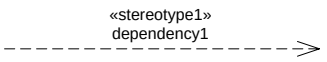
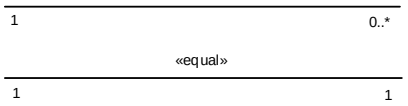
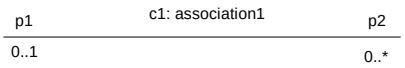
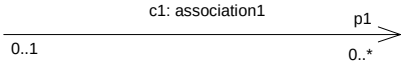
Element Name	Concrete Syntax Example	Abstract Syntax Reference
InternalBlockDiagram		**SysML::Blocks::Block**
Property	 	**UML4SysML::Property**
ActorPart	 	**SysML::Blocks::PartProperty typed by UML4SysML::Actor**
PropertySpecificType	 	**SysML::Blocks::PropertySpecifcType**

Table 8.4 - Graphical paths defined in Internal Block diagrams

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

8.3 UML Extensions

8.3.1 Diagram Extensions

8.3.1.1 Block Definition Diagram

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

8.3.1.1.1 Block and ValueType Definitions

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

8.3.1.1.2 Default «block» stereotype on unlabeled box

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

8.3.1.1.3 Labeled compartments

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

8.3.1.1.4 Constraints compartment

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

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

8.3.1.1.5 Namespace compartment

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

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

8.3.1.1.6 Structure compartment

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

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

8.3.1.1.7 Unit and QuantityKind definitions

Unit and QuantityKind elements are defined using a rectangular box notation similar to a block, in which only the “unit” or “quantityKind” stereotype keyword, the name of the Unit or QuantityKind, and optionally the “quantityKind” property value of a Unit may appear. Even though the base metaclass of Unit and QuantityKind is InstanceSpecification, the name

of a QuantityKind or Unit is not underlined, and no other graphical elements of a UML InstanceSpecification may be shown. The optional “quantityKind” property of a Unit and the “quantityKind” and/or “unit” properties of a ValueType are specified using standard stereotype property notations, which must refer by name to a QuantityKind or Unit which has already been defined separately and which is available for reference in the local namespace. A sample set of predefined quantity kinds and units is given in Annex C, Figure C.2.

8.3.1.1.8 Default multiplicities

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

8.3.1.1.9 Property-specific type

Enclosing the type name of a property in square brackets specifies that the type is a local specialization of the referenced type, which may be overridden to specify additional values or other customizations that are unique to the property. Redefined or added features of the newly defined type may be shown in compartments for the property on an internal block diagram. If no type name appears between the square brackets, the property-specific type is defined provided by its own declarations, without specializing any existing type.

8.3.1.2 Internal Block Diagram

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

8.3.1.2.1 Property types

Four general categories of properties of blocks are recognized in SysML: parts, references, value properties, and constraint properties. (See “Block” on page 44, for definitions of these property types.) A part or value property is always shown on an internal block diagram with a solid-outline box. A reference property is shown by a dashed-outline box, consistent with UML. Ports are special cases of properties, and have a variety of notations as defined in Clause 9, “Ports and Flows.” Constraint properties and their parameters also have their own notations as defined in Clause 10, “Constraint Blocks.”

8.3.1.2.2 Block reference in diagram frame

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

8.3.1.2.3 Compartments on internal properties

SysML permits any property shown on an internal block diagram to also show compartments within the property box. These compartments may be given standard or user-customized labels just as on block definitions. All features shown within these compartments must match those of the block or value type that types the property. For a property-specific type, these compartments may be used to specify redefined or additional features of the locally defined type. An unlabeled compartment on an internal property box is by default a structure compartment.

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

8.3.1.2.4 Compartments on a diagram frame

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

8.3.1.2.5 Property path name

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

This notation is purely a notational shorthand for a property that could otherwise be shown within a structure of nested property boxes, with the names in the dotted string taken from the name that would appear at each level of nesting. In other words, the internal property shown with a path name in the left-hand side of Figure 8.1 is equivalent to the innermost nested box shown at the right.

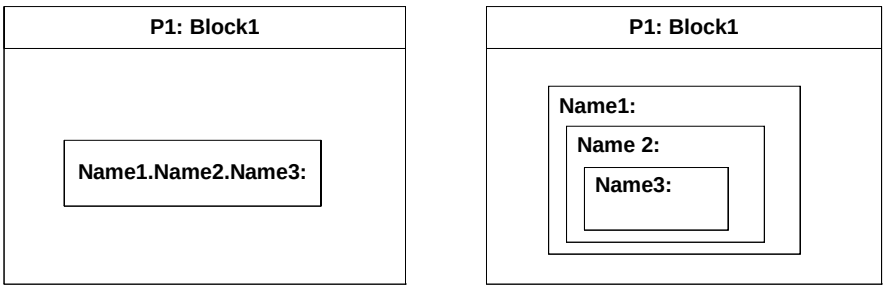


Figure 8.1 - Nested property reference

8.3.1.2.6 Nested connector end

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

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

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

8.3.1.2.7 Property-specific type

Enclosing the type name of an internal property in square brackets specifies that the type is a local specialization of the referenced type, which may be overridden to specify additional values or other customizations that are unique to the property. Redefined or added features of the newly defined type may be shown in compartments for the property. If the property name appears on its own, with no colon or type name, or if no type name appears between the square brackets, the property-specific type is entirely provided by its own declarations, without specializing any existing type.

8.3.1.2.8 Initial values compartment

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

Values are specified in an initialValues compartment by lines in the form <property-name> = <value-specification> or <property-name> : <type> = <value-specification>, each line of which specifies the initial value for one property owned either by the block that types the property or by any of its supertypes. This portion of concrete syntax is the same as may be shown for values within the UML instance specification notation, but this is the only element of UML InstanceSpecification notation that may be shown in an initial values compartment. See “Block” on page 44” for details of how values within initialValues compartments are represented in the SysML metamodel.

8.3.1.2.9 Default multiplicities

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

8.3.1.3 UML Diagram Elements not Included in SysML Block Definition Diagrams

The supported variety of notations for associations and association annotations has been reduced to simplify the burden of teaching, learning, and interpreting SysML diagrams for the systems engineering user. Notational and metamodel support for n-ary associations and qualified associations has been excluded from SysML. N-ary associations, shown in UML by a large open diamond with multiple branches, can be modeled by an intermediate block with no loss in expressive power. Qualified associations, shown in SysML by an open box at the end of an association path with a property name inside, are a specialized feature of UML that specifies how a property value can represent an identifier of an associated target. This capability, while useful for data modeling, does not seem essential to accomplish any of the SysML requirements for support of systems engineering. The use of navigation arrowheads on an association has been simplified by excluding the case of arrowheads on both ends, and requiring that such an association always be shown without arrowheads on either end. An “X” on a single end of an association to indicate that an end is not navigable has similarly been dropped, as has the use of a small filled dot at the end of an association to indicate that the end is owned by the associated classifier.

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

8.3.1.4 UML Diagram Elements not Included in SysML Internal Block Diagrams

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

8.3.2 Stereotypes

Package Blocks

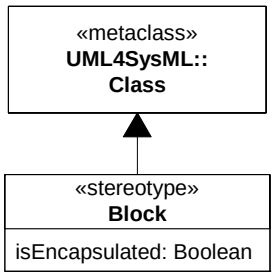


Figure 8.2 - Abstract syntax expressions for SysML blocks

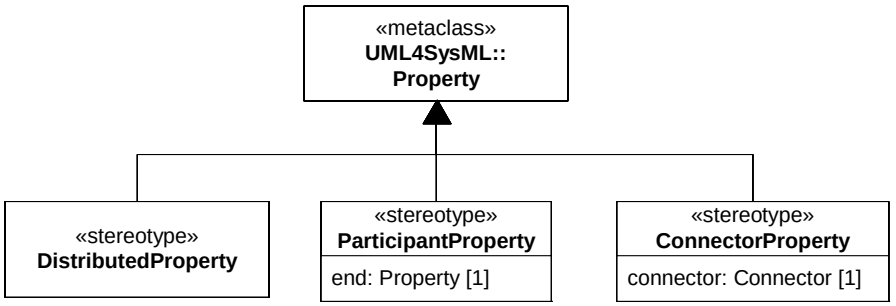


Figure 8.3 - Abstract syntax extensions for SysML properties

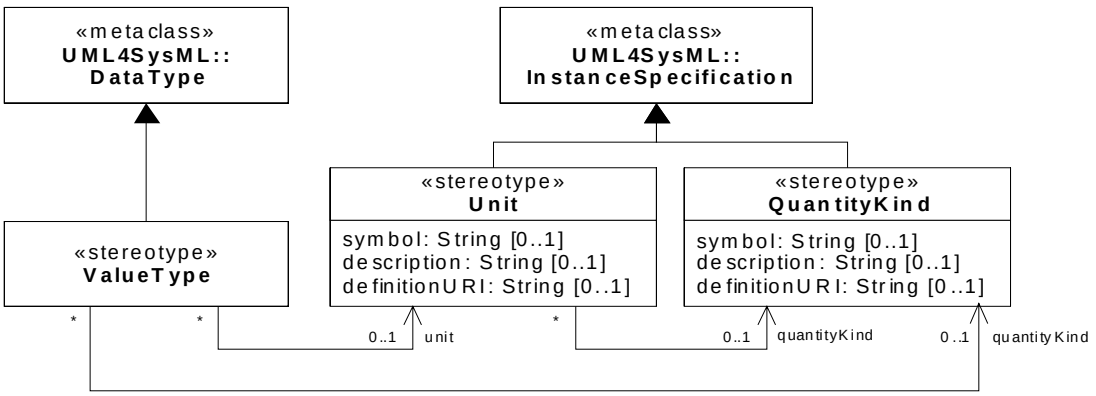


Figure 8.4 - Abstract syntax extensions for SysML value types

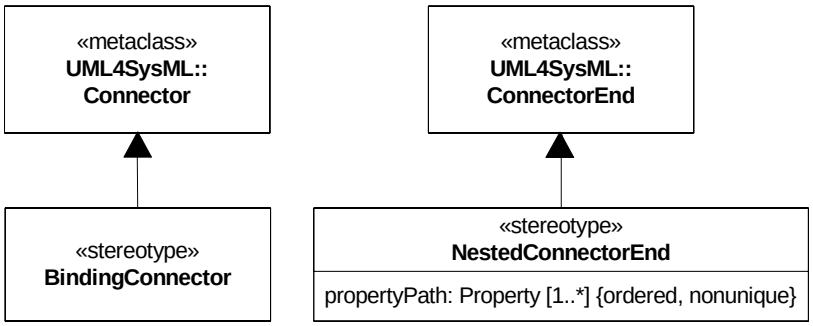


Figure 8.5 - Abstract syntax extensions for SysML connector ends

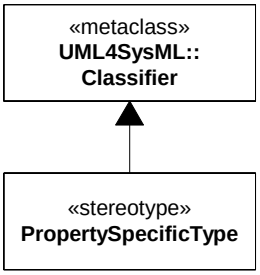


Figure 8.6 - Abstract syntax extensions for SysML property-specific types

8.3.2.1 Binding Connector

Description

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

Constraints

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

8.3.2.2 Block

Description

A Block is a modular unit that describes the structure of a system or element. It may include both structural and behavioral features, such as properties and operations, that represent the state of the system and behavior that the system may exhibit. Some of these properties may hold parts of a system, which can also be described by blocks that type the

properties. Properties without types do not restrict the instances that can be values of the properties, as if they had the most general type possible. A block may include a structure of connectors between its properties to indicate how its parts or other properties relate to one another.

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

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

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

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

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

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

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

In addition to the form of default value specifications that SysML supports on properties of a block (with an optional "=" <value-specification> string following the rest of a property definition), SysML supports an additional form of value specification for properties using initialValue compartments on an internal block diagram (see "Internal Block Diagram" on page 40). An entire tree of context-specific values can be specified on a containing block to carry values of nested properties as shown on an internal block diagram.

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

If a property belonging to a block has a specification of initial values for any of the properties belonging to its type, then the default value of that property must be a UML InstanceValue element. This element must reference a UML InstanceSpecification element created to hold the initial values of the individual properties within its usage context. The instance specification must be unnamed and owned by the same package that owns the outermost containing block for which the initial values are being specified.

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

Attributes

- isEncapsulated: Boolean [0..1]
  If true, then the block is treated as a black box; a part typed by this black box can only be connected via its ports or directly to its outer boundary. If false, or if a value is not present, then connections can be established to elements of its internal structure via deep-nested connector ends.

Constraints

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

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

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

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

[5] The following constraint under sub clause 9.3.6, "Connector" in the UML 2 Superstructure Specification is removed by SysML: "[3] The ConnectableElements attached as roles to each ConnectorEnd owned by a Connector must be roles of the Classifier that owned the Connector, or they must be ports of such roles."

[6] If a property owned by a SysML Block or SysML ValueType is typed by a SysML ValueType, then the aggregation attribute of the property must be “composite.”

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

[8] Any classifier that specializes a Block must also have the Block stereotype or one of its specializations applied.

[9] The following constraint under sub clause 9.3.7, “ConnectorEnd” in the UML 2 Superstructure Specification is removed by SysML: “[3] The property held in self.partWithPort must not be a Port.”

8.3.2.3 ConnectorProperty

Description

Connectors can be typed by association classes that are stereotyped by Block (association blocks, see “ParticipantProperty” on page 48). These connectors specify instances of the association block created within the instances of the block that owns the connector. The values of a connector property are instances of the association block created due to the connector referred to by the connector property.

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

Attributes

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

Constraints

[1] ConnectorProperty may only be applied to properties of classes stereotyped by Block.

[2] The connector attribute of the applied stereotype must refer to a connector owned or inherited by a block owning the property on which the stereotype is applied.

[3] The aggregation of a property stereotyped by ConnectorProperty must be composite.

[4] The type of the connector referred to by a connector attribute must be an association class stereotyped by Block.

[5] A property stereotyped by ConnectorProperty must have the same name and type as the connector referred to by the connector attribute.

8.3.2.4 DistributedProperty

DistributedProperty is a stereotype of Property used to apply a probability distribution to the values of the property. Specific distributions should be defined as subclasses of the DistributedProperty stereotype with the operands of the distributions represented by properties of those stereotype subclasses. A sample set of probability distributions that could be applied to value properties is given in Annex D, “Distribution Extensions” on page 242.”

Constraints

[1] The DistributedProperty stereotype may be applied only to properties of classifiers stereotyped by Block or ValueType.

8.3.2.5 NestedConnectorEnd

Description

The NestedConnectorEnd stereotype of UML ConnectorEnd extends a UML ConnectorEnd so that the connected property may be identified by a multi-level path of accessible properties from the block that owns the connector.

Attributes

- propertyPath: Property [1..*] {ordered, nonunique}
  The propertyPath list of the NestedConnectorEnd stereotype must identify a path of containing properties that identify the connected property in the context of the block that owns the connector. The ordering of properties is from a property of the block that owns the connector, through a property of each intermediate block that types the preceding property, ending in a property that includes the nested property in its type. The property attached to the connector end is not included in the propertyPath list, but instead is held by the role property of the connector end. The same property might appear more than once because a block can own a property with the same block as a type, or another block that has the same property.

Constraints

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

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

[3] Within a connector end to which the NestedConnectorEnd stereotype has been applied, the role property of the connector end must be contained in the type of the property at the last position of the propertyPath list.

8.3.2.6 ParticipantProperty

Description

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

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

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

Attributes

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

Constraints

[1] ParticipantProperty may only be applied to properties of association classes stereotyped by Block.

[2] ParticipantProperty may not be applied to properties that are member ends of an association.

[3] The aggregation of a property stereotyped by ParticipantProperty must be none.

[4] The end attribute of the applied stereotype must refer to a member end of the association block owning the property on which the stereotype is applied.

[5] A property stereotyped by ParticipantProperty must have the same type as the property referred to by the end attribute.

[6] The property referred to by end must have a multiplicity of 1.

8.3.2.7 PropertySpecificType

The PropertySpecificType stereotype is automatically applied to the classifier that types a property with a property-specific type. This classifier can contain definitions of new or redefined features that extend the original classifier referenced by the property-specific type.

Classifiers with the PropertySpecificType stereotype are owned by the block that owns the property that has the property-specific type. A classifier with this stereotype must specialize at most a single classifier that was referenced as the starting classifier of the property-specific type. If there is no starting classifier (which occurs if no existing name is specified as the starting type of a property-specific type), then a classifier with the stereotype applied has no specialization relationship from any other classifier.

Constraints

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

[2] The name of a classifier to which a PropertySpecificType is applied must be missing. (The “name” attribute of the NamedElement metaclass must be empty.)

8.3.2.8 QuantityKind

A QuantityKind is a kind of quantity that may be stated by means of defined units. For example, the quantity kind of length may be measured by units of meters, kilometers, or feet. QuantityKind is defined as a stereotype of InstanceSpecification, but it uses this metaclass only to define supporting elements for ValueType definitions. (The reuse of InstanceSpecification to define another metaclass is similar to the EnumerationLiteral metaclass in UML.) The only valid use of a QuantityKind instance is to be referenced by the “quantityKind” property of a ValueType or Unit stereotype.

See the non-normative model library in Annex D, “Model Library for Quantities, Units, Dimensions, and Values (QUDV)” on page 222,” for an optional way to specify more comprehensive definitions of units and quantity kinds as part of systems of units and systems of quantities. The name of a QuantityKind, its definitionURI, or other means may be used to link individual quantity kinds to additional sources of documentation such as this optional model library.

Attributes

- symbol: String [0..1]
  Short symbolic name of the quantity kind.
- description: String [0..1]
  Textual description of the quantity kind.
- definitionURI: String [0..1]
  URI that references an external definition of the quantity kind.

8.3.2.9 Unit

A Unit is a quantity in terms of which the magnitudes of other quantities that have the same quantity kind can be stated. A unit often relies on precise and reproducible ways to measure the unit. For example, a unit of length such as meter may be specified as a multiple of a particular wavelength of light. A unit may also specify less stable or precise ways to express some value, such as a cost expressed in some currency, or a severity rating measured by a numerical scale.

Unit is defined as a stereotype of InstanceSpecification, but it uses this metaclass only to define supporting elements for ValueType definitions. (The reuse of InstanceSpecification to define another metaclass is similar to the EnumerationLiteral metaclass in UML.)

The only valid use of a Unit instance is to be referenced by the “unit” property of a ValueType stereotype.

See the non-normative model library in Annex D, “Model Library for Quantities, Units, Dimensions, and Values (QUDV)” on page 222,” for an optional way to specify more comprehensive definitions of units and quantity kinds as part of systems of units and systems of quantities. The name of a Unit, its definitionURI, or other means may be used to link individual units to additional sources of documentation such as this optional model library.

Attributes

- symbol: String [0..1]
  Short symbolic name of the unit.
- description: String [0..1]
  Textual description of the unit.
- definitionURI: String [0..1]
  URI that references an external definition of the unit.
- quantityKind: QuantityKind [0..1]
  A kind of quantity that may be stated by means of defined units, as identified by an instance of the QuantityKind stereotype.

8.3.2.10 ValueType

Description

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

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

SysML ValueType adds an ability to carry a unit of measure and quantity kind associated with the value. A quantity kind is a kind of quantity that may be stated in terms of defined units, but does not restrict the selection of a unit to state the value. A unit is a particular value in terms of which a quantity of the same quantity kind may be expressed.

A SysML ValueType may define its own properties and/or operations, just as for a UML DataType. See “Block” on page 44” for property classifications that SysML defines for either a Block or ValueType.

Attributes

- quantityKind: QuantityKind [0..1]
  A kind of quantity that may be stated by means of defined units, as identified by an instance of the QuantityKind stereotype. A value type may optionally specify a quantity kind without any unit. Such a value has no concrete representation, but may be used to express a value in an abstract form independent of any specific units.
- unit: Unit [0..1]
  A quantity in terms of which the magnitudes of other quantities that have the same quantity kind can be stated, as identified by an instance of the Unit stereotype.

Constraints

[1] Any classifier that specializes a ValueType must also have the ValueType stereotype applied.

8.3.3 Model Libraries

Package PrimitiveValueTypes

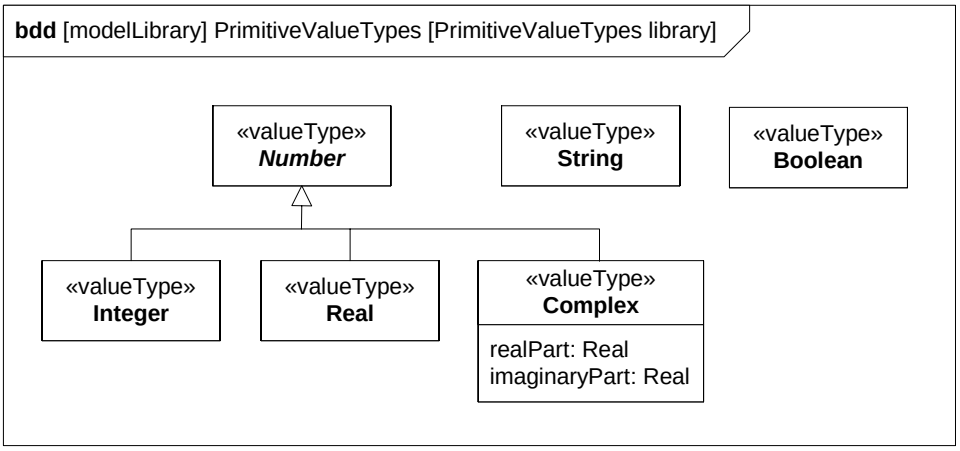


Figure 8.7 - Model library for primitive value types

8.3.3.1 Boolean

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

8.3.3.2 Complex

Description

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

Attributes

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

8.3.3.3 Integer

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

8.3.3.4 Number

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

8.3.3.5 Real

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

8.3.3.6 String

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

8.4 Usage Examples

8.4.1 Wheel Hub Assembly

In Figure 8.8 a block definition diagram shows the blocks that comprise elements of a Wheel. The block property LugBoltJoint.torque has a specialization of DistributedProperty applied to describe the uniform distribution of its values. Examples of such distributions can be found in “Model Library for Quantities, Units, Dimensions, and Values (QUDV)” on page 222.”

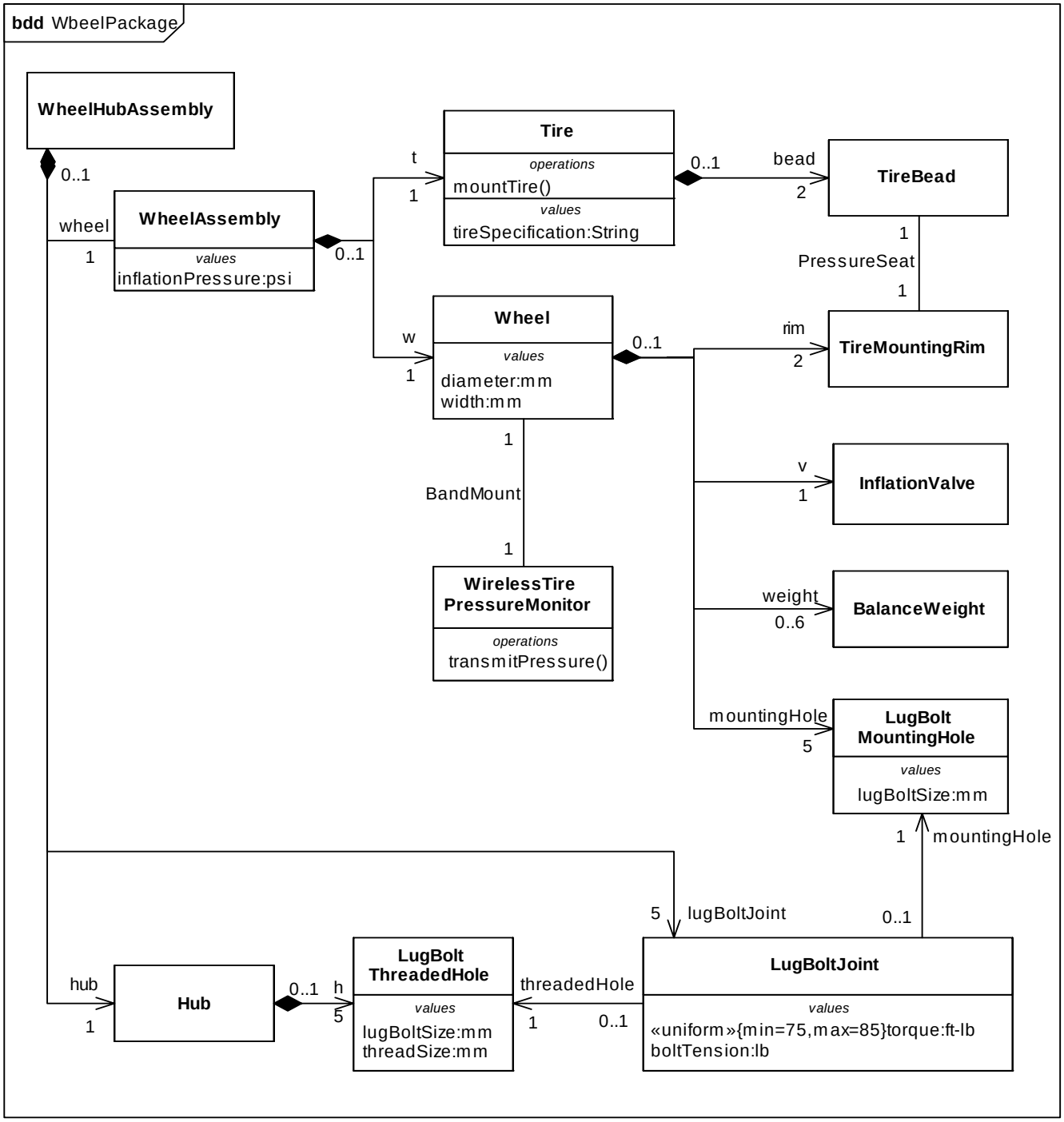


Figure 8.8 - Block diagram for the Wheel Package

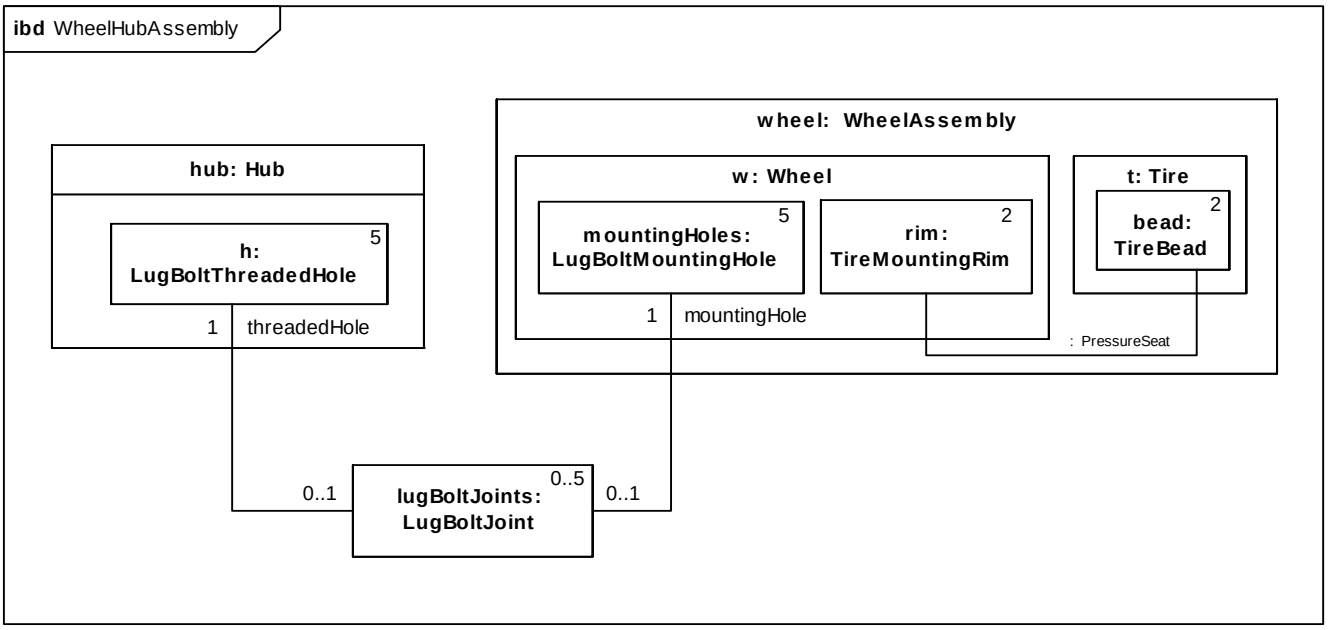


Figure 8.9 - Internal Block Diagram for WheelHubAssembly

In Figure 8.9 an internal block diagram (ibd) shows how the blocks defined in the Wheel package are used. This ibd is a partial view that focuses on particular parts of interest and omits others from the diagram, such as the “v” InflationValve and “weight” BalanceWeight, which are also parts of a Wheel.

8.4.2 Example Value Type Definitions

In Figure 8.10, several value types that use standard units of measure from the International System of Units (SI) are defined to be available in the Example Value Type Definitions package. The value types in this package could be imported into other contexts for typing properties of SysML Blocks. Because a SysML Unit can already identify a type of quantity, or QuantityKind, that the unit measures, a value type only needs to identify the unit to identify a quantity kind as well. The value types in this example refer to units that are assumed to be defined in an imported package, such as the Model Library defined in Annex D, “Model Library of SysML Quantity Kinds and Units for ISO 80000-1” on page 219.”

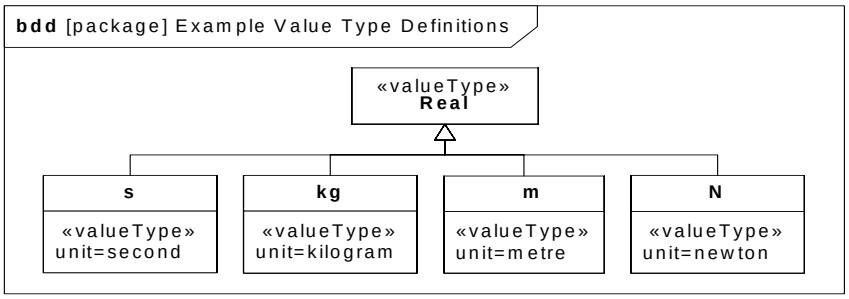


Figure 8.10 - Defining Value Types with units of measure from the International System of Units (SI)

8.4.3 Design Configuration for SUV EPA Fuel Economy Test

SysML internal block diagrams may be used to specify blocks with unique identification and property values. Figure C.38 in Annex C shows an example used to specify a unique vehicle with a vehicle identification number (VIN) and unique properties such as its weight, color, and horsepower. This concept is distinct from the UML concept of instance specifications in that it does not imply or assume any run-time semantic, and can also be applied to specify design configurations.

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

8.4.4 Water Delivery

Association blocks can be decomposed into connectors between properties of the associated blocks. These properties can be ports, as in the water delivery example in “Association and Port Decomposition” on page 76.

9 Ports and Flows

9.1 Overview

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

9.1.1 Ports

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

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

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

9.1.3 Proxy Ports and Full Ports

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

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

9.1.4 Item Flows

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

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

9.1.5 Deprecation of Flow Ports and Flow Specifications

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

9.2 Diagram Elements

9.2.1 Block Definition Diagram

Table 9.1 - Graphical nodes defined in Block Definition diagrams

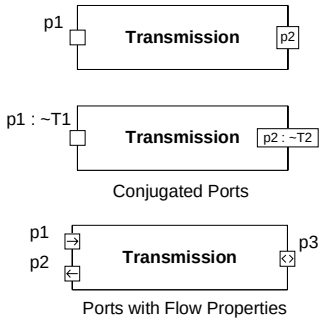
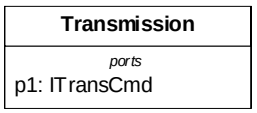
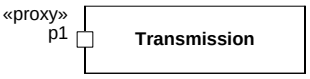
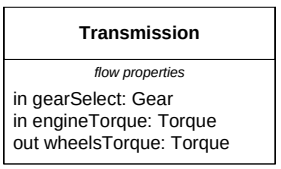
Node Name	Concrete Syntax	Abstract Syntax Reference
Port		**UML4SysML::Port**
Port (Compartment Notation)		**UML4SysML::Port**
Port (Nested)		**UML4SysML::Port**
ProxyPort		**SysML::Ports&Flows::ProxyPort**
FullPort		**SysML::Ports&Flows::FullPort**
FlowProperty		**SysML::Ports&Flows:: FlowProperty**

Table 9.1 - Graphical nodes defined in Block Definition diagrams

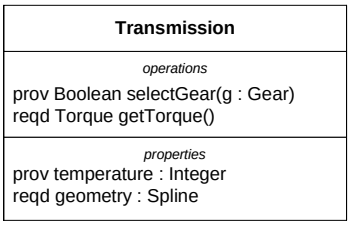
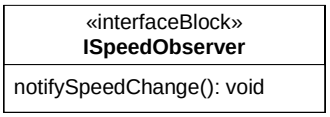
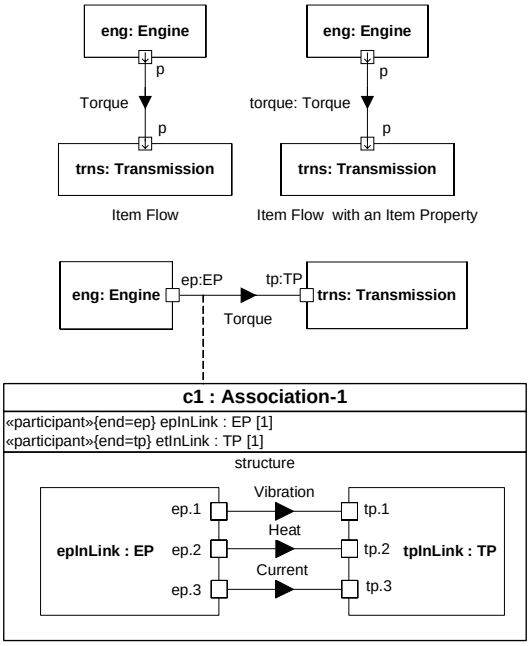
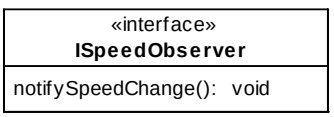
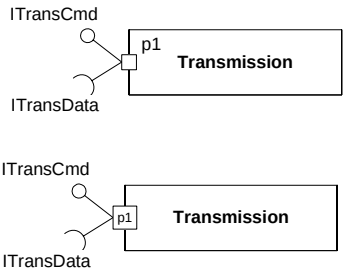
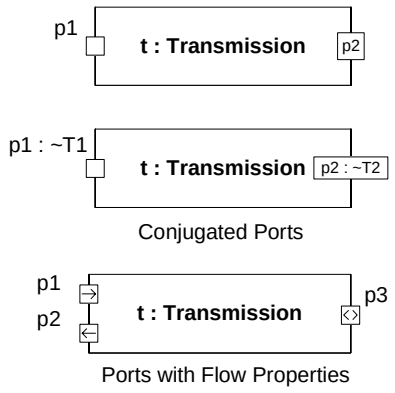
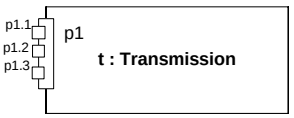
Node Name	Concrete Syntax	Abstract Syntax Reference
Required and Provided Features		**SysML::Ports&Flows:: DirectedFeature**
InterfaceBlock		**SysML::Ports&Flows:: InterfaceBlock**
Item Flow		**SysML::Ports&Flows::ItemFlow**
Interface		**UML4SysML::Interfaces:: Interface**

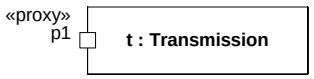
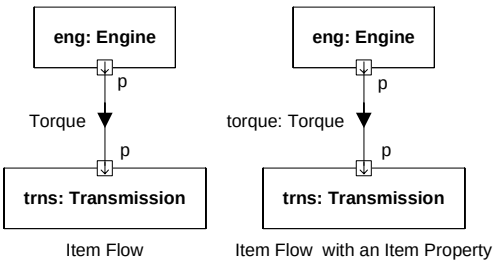
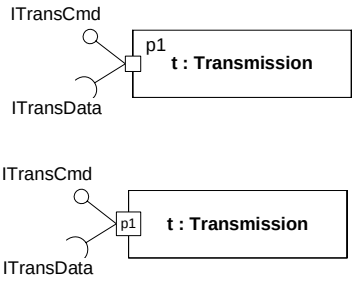
Table 9.1 - Graphical nodes defined in Block Definition diagrams

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

9.2.2 Internal Block Diagram

Table 9.2 - Graphical nodes defined in Internal Block diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Port		**UML4SysML::Port**
Port (Nested)		**UML4SysML::Port**

ProxyPort		**SysML::Ports&Flows::ProxyPort**
FullPort		**SysML::Ports&Flows::FullPort**
ItemFlow		**SysML::Ports&Flows::ItemFlow**
Required and Provided Interfaces		**UML4SysML::Interface**

9.3 UML Extensions

9.3.1 Diagram Extensions

9.3.1.1 DirectedFeature

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

9.3.1.2 FlowProperty

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

9.3.1.3 FullPort

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

9.3.1.4 InvocationOnNestedPortAction

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

9.3.1.5 ItemFlow

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

9.3.1.6 Port

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

Port rectangles can have port rectangles overlapping their boundaries, to notate a port type that has ports (nested ports). The fill color of port rectangles is white and the line and text colors are black.

Ports with types that have flow properties all in the same direction, either all in or all out, can have an arrow inside them indicating the direction of the properties with respect to the owning block. (See "FlowProperty" on page 69 for definition of owning block of proxy ports in this case.) This includes the direction of flow properties on nested ports, and if the port is full and its type is unencapsulated, ports on parts of the port, recursively. The arrows are perpendicular to the boundary lines they overlap. Arrows in conjugated ports are reversed from the flow property direction. Ports with types that have flow properties in different directions or flow properties that are all in both directions, including have two open arrow heads inside them facing away from each other (<>). This includes the directions of nested and contained flow properties as described above for one-way arrows. Ports appearing in block compartments can have their direction appear textually before the port name as "in," "out," or "inout" determined in the same way as the arrow direction.

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

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

9.3.1.7 ProxyPort

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

9.3.1.8 TriggerOnNestedPort

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

9.3.2 Stereotypes

Package Ports&Flows

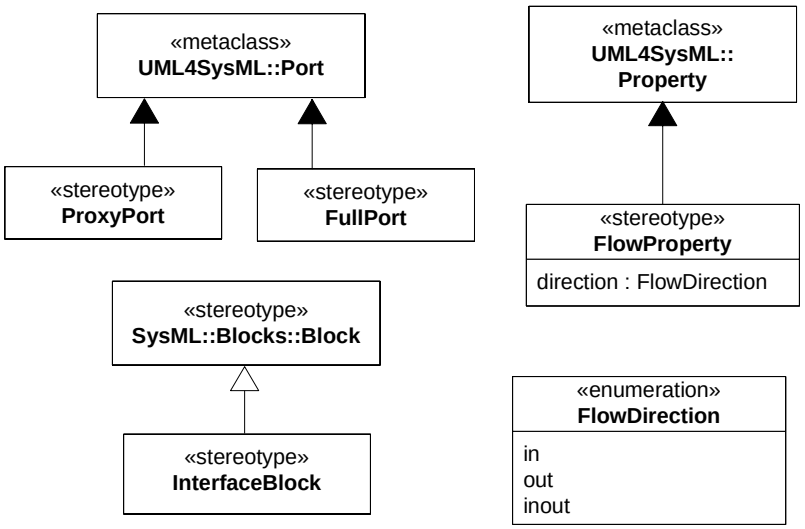


Figure 9.1 - Port Stereotypes

Package Ports&Flows

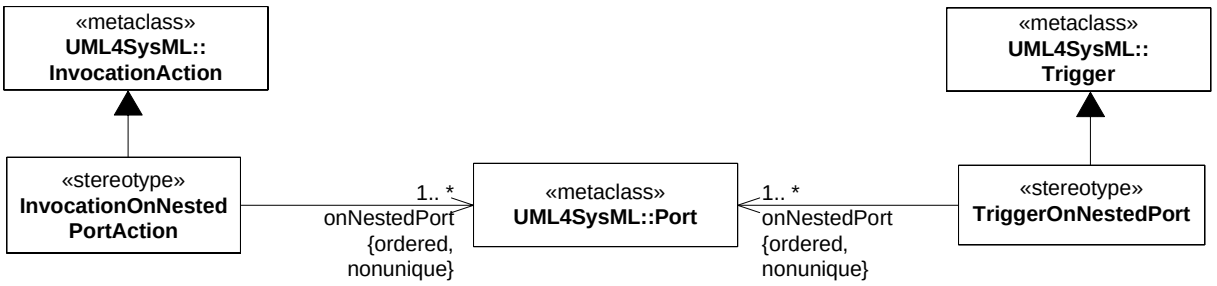


Figure 9.2 - Stereotypes for Actions on Nested Ports

Package Ports&Flows

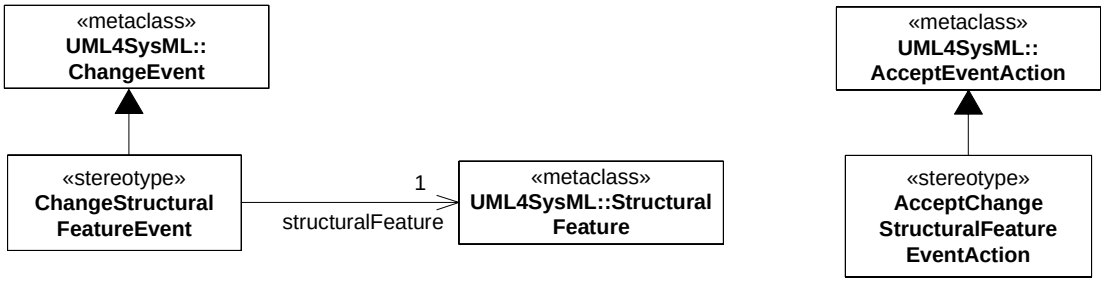


Figure 9.3 - Stereotypes for Property Value Change Events

Package Ports&Flows

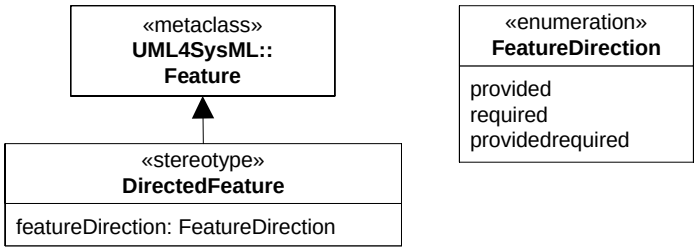


Figure 9.4 - Provided and Required Features

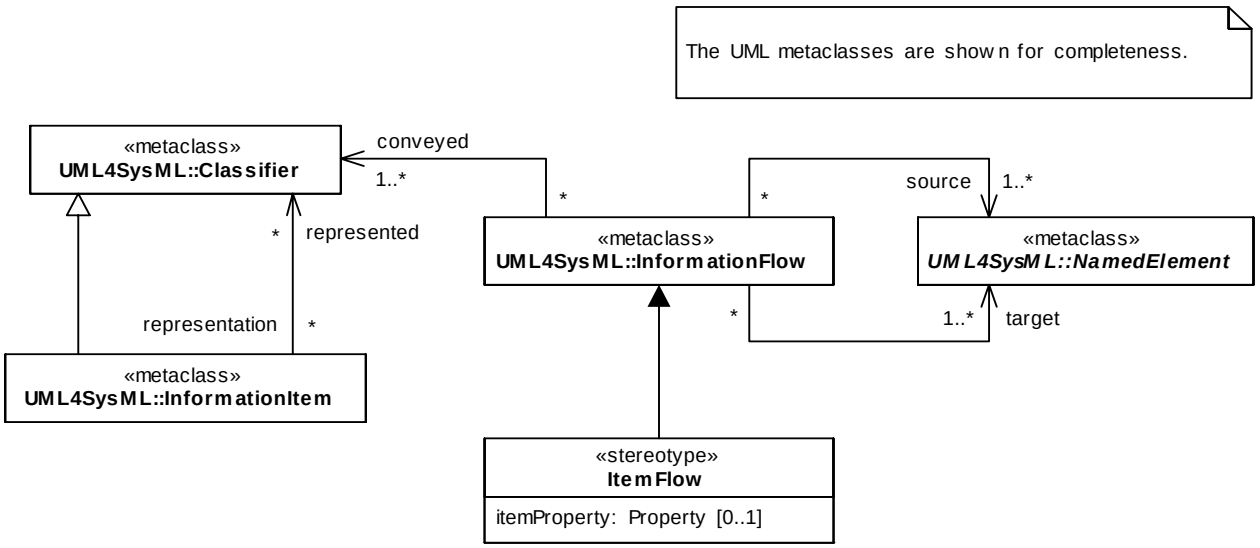


Figure 9.5 - ItemFlow Stereotype

9.3.2.1 AcceptChangeStructuralFeatureEventAction

Description

Accept change structural feature event actions handle change structural feature events (see “DirectedFeature” on page 67). The actions have exactly two output pins. The first output pin holds the values of the structural feature just after the values changed, while the second pin holds the values just before the values changed. The action only accepts events for structural features on the blocks owning the behavior containing the action, or on the behavior itself, if the behavior is not owned by a block.

Constraints

[1] The action has exactly one trigger, the event of which must be a change structural feature event.

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

[3] The structural feature of the trigger event must be owned by or inherited by the context of the behavior containing the action. (The context of a behavior is either its owning block or itself if it is not owned by a block. See definition in the UML 2 Superstructure Specification.)

[4] Visibility of the structural feature of the trigger event must allow access to the object performing the action.

[5] The constraint under sub clause 11.3.2, “AcceptEventAction” in the UML 2 Superstructure Specification, “[2] There are no output pins if the trigger events are only ChangeEvents,” is removed for accept event actions that have AcceptChangeStructuralFeatureEventAction applied.

9.3.2.2 Block

Description

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

9.3.2.3 ChangeStructuralFeatureEvent

Description

A ChangeStructuralFeatureEvent models changes in values of structural features.

Attributes

- structuralFeature : StructuralFeature
  The event models occurrences of changes to values of this structural feature.

Constraints

[1] The structural feature must not be static.

[2] The structural feature must have exactly one featuringClassifier.

9.3.2.4 DirectedFeature

Description

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

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

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

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

  Parameters without types are treated as if their type is more general than all other types.
- Provided parameter multiplicity has the same condition as type, where wider multiplicities are “more general” than narrower ones.
- Provided parameter order (of each parameter separately) has the same condition as type, where unordered parameters are “more general” than ordered ones.
- Provided parameter uniqueness (of each parameter separately) has the same condition as type, where non-unique parameters are “more general” than unique ones.
- Provided operation preconditions are the same as or more general than required ones.
- Provided operation body conditions and postconditions are the same or more specialized than required ones.

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

Reading or writing provided non-flow properties due to required non-flow properties can only occur when the features match. Matching non-flow properties must have the same name. For reading non-flow properties, the types, multiplicities, uniqueness, and ordering must match in the same way as out parameters for behavioral features above. For writing non-flow properties, the types, multiplicities, uniqueness, and ordering must match in the same way as in parameters for

behavioral features above. For both reading and writing non-flow properties, the types, multiplicities, uniqueness, and ordering must be the same. If provided and required non-flow properties both have defaults, the default value specification of the required feature is used for writing and the default specification of the provided feature is used for reading.

Attributes

- featureDirection : FeatureDirection
  Specifies whether the feature is supported by the owning block (featureDirection="provided"), or is to be supported by other blocks for the owning block to use (featureDirection="required"), or both (featureDirection="providedrequired"). The meaning of the direction is reversed for conjugated ports.

Constraints

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

[2] Operations that are not provided must not have or inherit methods.

9.3.2.5 FeatureDirection

Description

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

Literal values are:

- provided:
  Indicates that the feature is supported by the owning block.
- required:
  Indicates that the feature must be supported by other blocks.
- providedrequired:
  Indicates that the feature is both provided and required.

The meanings of the "required" and "provided" literals are switched for conjugated ports. In these cases the actual use is in the opposite direction than the one specified by the enumeration literal.

9.3.2.6 FlowDirection

Description

FlowDirection is an enumeration type that defines literals used for specifying the direction that items can flow to or from a block. FlowDirection is used by flow properties to indicate the direction that its items can flow to or from its owner. (See "FlowProperty" on page 69 for definition of owning block of proxy ports in this case.)

Literal Values are:

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

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

The meanings of the “in” and “out” literals are switched for conjugated ports. In these cases the actual flow is in the opposite direction than the one specified by the enumeration literal.

9.3.2.7 FlowProperty

Description

A FlowProperty signifies a single kind of flow element that can flow to/from a block. A flow property’s values are either received from or transmitted to another block. An “in” flow property value cannot be modified by its owning or realizing block, or blocks contained by its owning or realizing block. An “out” flow property can only be modified by its owning or realizing block, or blocks contained by its owning or realizing block. An “inout” flow property can be used as an “in” flow property or an “out” flow property. (The owning block of a proxy port in this case depends on how the port is nested in the internal structures of blocks, because the block directly owning the port might be used to type ports or parts at different levels of nesting in multiple blocks, or the same block. The owning block of a proxy port in the internal structure of a block is the block typing the innermost full port or part under which the port is nested.) The meaning of the direction is reversed for conjugated ports (UML isConjugated = true). In the description below, flow property direction refers to the direction after conjugation is taken into account.

Flow due to flow properties can only occur when flow properties match. Matching flow properties must have matching direction and types. Matching direction is defined below. Flow property types match when the target flow property type has the same, or a generalization of, the source flow property type. (See “ItemFlow” on page 71 for looser constraints on flow property types across connectors with item flows.) If multiple flow properties on either end of a connector match by direction and type, then the names of the flow properties must also be the same for flow to occur. If multiple flow properties on either end match by direction, type, and name, which can happen for unnamed flow properties, then no flow will occur.

Flow properties enable item flows across connectors between usages typed by blocks having the properties. For Block and ValueType flow properties, setting an “out” or “inout” FlowProperty value of a block usage on one end of a connector will result in assigning the same value of an “in” or “inout” FlowProperty of a block usage at the other end of the connector, provided the flow properties are matched. Flow properties of type Signal imply sending and/or receiving of a signal usage. This paragraph so far does not apply to internal connectors of proxy ports, see next paragraph.An “out” FlowProperty of type Signal means that the owning Block may send the signal via connectors and an “in” FlowProperty means that the owning block is able to receive the Signal.

Items going to or from behavioral ports (UML isBehavior = true) are actually going to or from the owning block. (See “Block” on page 66 for definition of owning block of proxy ports in this case.) Items going to or from non-behavioral ports (UML isBehavior = false) are actually going to the port itself (for full ports) or to internal parts connected to the port (for proxy ports). Because of this, flow properties of a proxy port are the same as flow properties on the owning block or internal parts, so the flow property directions must be the same on the proxy port and owning block or internal parts for items to flow. See“ProxyPort” on page 72 for the definition of internal connectors and the semantics of proxy ports.

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

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

Attributes

- direction : FlowDirection
  Specifies if the property value is received from an external block (direction="in"), transmitted to an external Block (direction="out") or both (direction="inout"). The meaning of the direction is reversed for conjugated ports.

Constraints

[1] A FlowProperty is typed by a ValueType, Block, or Signal.

9.3.2.8 FullPort

Description

Full ports specify a separate element of the system from the owning block or its internal parts. They might have their own internal parts, and behaviors to support interaction with the owning block, its internal parts, or external blocks. They cannot be behavioral ports, or linked to internal parts by binding connectors, because these constructs imply identity with the owning block or internal parts. Full ports also cannot be conjugated, because behaviors defined on their types are defined independently of the ports these types might be used on.

Constraints

[1] Full ports cannot also be proxy ports. This applies even if some of the stereotypes are on subsetted or redefined ports.

[2] Binding connectors cannot link full ports to other composite properties of the block owning the port, except ports that are not full.

[3] Full ports cannot be behavioral (isBehavior=false).

[4] Full ports cannot be conjugated (isConjugated=false).

9.3.2.9 InterfaceBlock

Description

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

Constraints

[1] Interface blocks cannot own or inherit behaviors, have classifier behaviors, or methods for their behavioral features.

[2] Interface blocks cannot have composite properties that are not ports.

[3] Ports owned by interface blocks can only be typed by interface blocks.

9.3.2.10 InvocationOnNestedPortAction

Description

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

Attributes

- onNestedPort : Port [1..*] {ordered, nonunique}
  The onNestedPort list must be a path of containing ports that identify the port receiving the invocation in the context of the target object of the invocation. The ordering of ports is from a port of the target object, through a port of each intermediate block that types the preceding port, until a port is reached that contains a port within its type specified by the onPort property of the invocation action. The onPort port is not included in the onNestedPort list. The same port might appear more than once because a block can own a port with the same block as a type, or another block that has the same property.

Constraints

[1] The onPort property of an invocation action must have a value when this stereotype is applied.

[2] The port at the first position in the onNestedPort property must be owned by the target object of the stereotyped action.

[3] The port at each successive position of the onNestedPort property, following the first position, must be contained in the block that types the port at the immediately preceding position.

[4] Within the stereotyped action, the onPort port of the invocation action must be contained in the type of the port at the last position of the onNestedPort list.

9.3.2.11 ItemFlow

Description

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

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

Item flows on connectors must be compatible with flow properties of the blocks usages at each end of the connector, if any. The direction of the item flow must be compatible wit the direction of flow specified by the flow properties. (See “FlowDirection” on page 68 and “FlowProperty” on page 69 about flow property direction.) Each classifier of conveyed items on an item flow must be the same as, a specialization of, or a generalization of at least one flow property type on each end of the connected block usages (or their accessible nested block usages recursively, see “Block” on page 44 about encapsulated blocks). The target flow property type must be the same as, or a generalization of, a classifier of the item flow or the source flow property type, whichever is more specialized. (See “FlowProperty” on page 69 for tighter constraints on flow property types across connectors without item flows.)

Attributes

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

Constraints

[1] A Connector or an Association, or an inherited Association must exist between the source and the target of the InformationFlow.

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

[3] itemProperty is a property of the common (possibly indirect) owner of the source and the target.

[4] itemProperty cannot have a value if the item flow is realized by an Association.

[5] If an ItemFlow has an itemProperty, one of the classifiers of conveyed items must be the same as the type of the item property.

[6] If an ItemFlow has an itemProperty, its name should be the same as the name of the item flow.

9.3.2.12 ProxyPort

Description

Proxy ports identify features of the owning block or its internal parts that are available to external blocks through external connectors to the ports. They do not specify a separate element of the system from the owning block or internal parts. Actions on features of a proxy port have the same effect as if they were acting on features of the owning block or internal parts the port stands in for, and changes to features of the owning block or internal parts that the proxy port makes available to external blocks are visible to those blocks via connectors to the port. (This applies to provided features; for required features, see “DirectedFeature” on page 67.) Proxy ports do not specify their own behaviors or internal parts, and must be typed by interface blocks. Their nested ports must also be proxy ports. Completely specified proxy ports must be connected to internal parts or be behavioral, to enable the owning block or connected internal parts to handle or initiate any interactions through the port. However, blocks can be defined with non-behavioral proxy ports that do not have internal connectors, with the expectation that these will be added in specialized blocks. Internal connectors to ports are the ones inside the port’s owner (specifically, they are the ones that do not have a UML partwithPort on the connector end linked to the port, assuming NestedConnectorEnd is not applied to that end, or if NestedConnectorEnd is applied to that end, they are the connectors that have only ports in the property path of that end). The rest of the connectors linked to a port are external.

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

Constraints

[1] Proxy ports cannot also be full ports. This applies even if some of the stereotypes are on subsetted or redefined ports.

[2] Proxy ports can only be typed by interface blocks.

[3] Ports owned by the type of a proxy port must be proxy ports.

9.3.2.13 TriggerOnNestedPort

Description

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

Attributes

- onNestedPort : Port [1..*] {ordered, nonunique}
  The onNestedPort list must be a path of containing ports that identify a port on which the event is occurring, in the context of a block in which the trigger is used. The ordering of ports is from a port of the receiving object, through a port of each intermediate block that types the preceding port, until a port is reached that contains a port within its type specified by the port property of the trigger. The port property is not included in the onNestedPort list. The same port might appear more than once because a block can own a port with the same block as a type, or another block that has the same property.

Constraints

[1] The port property of the stereotyped trigger must have exactly one value, and the value cannot be a full port.

[2] The values of the onNestedPort property must not be full ports.

[3] The port at the first position in the onNestedPort property must be owned by a block in which the trigger is used.

[4] The port at each successive position of the onNestedPort property, following the first position, must be contained in the block that types the port at the immediately preceding position.

[5] The value of the port property of the stereotyped trigger must be contained in the type of the port at the last position of the onNestedPort list.

9.4 Usage Examples

9.4.1 Ports with Required and Provided Features

Figure 9.6 is a fragment of the ibd:PwrSys diagram used in the HybridSUV Sample Problem in Annex C. (The complete diagram is in Figure C.19.) The ecu:PowerControlUnit part has three ports with required and provided features, each connected to a port of another part. Each of the ports in this example is typed by a block specifying provided and required features available via connectors to the ports. For example, the ICE block specifies the provided operations setMixture and setThrottle, the provided properties RPM, temperature, and isKnocking, and required property isControlOn, as shown in Figure C.20 in Annex C. This block types the ctrl port of InternalCombustionEngine and the ice port of PowerControlUnit, but is conjugated on the ice port. This means the provided features of ICE are provided by the ctrl port of InternalCombustionEngine, and required by the ice port of PowerControlUnit, while the required features of ICE are required by the ctrl port of InternalCombustionEngine, and provided by the ice port of PowerControlUnit. Since the ecu:PowerControlUnit part and ice:InternalCombustionEngine part are connected via these ports, the ecu:PowerControlUnit part may invoke setThrottle and setMixture on the ice:InternalCombustionEngine part via its ice port, across the connector to the ctrl port of ice:InternalCombustionEngine. By invoking these operations, the PowerControlUnit can set the throttle and mixture of the InternalCombustionEngine. The PowerControlUnit can also read properties of the InternalCombustionEngine across the connector to find out the its rpm, temperature, and whether it is knocking. Inversely, the InternalCombustionEngine can read the isControlOn property of the PowerControlUnit across the connector to determine if the unit is still operating, and possibly shut down if it is not.

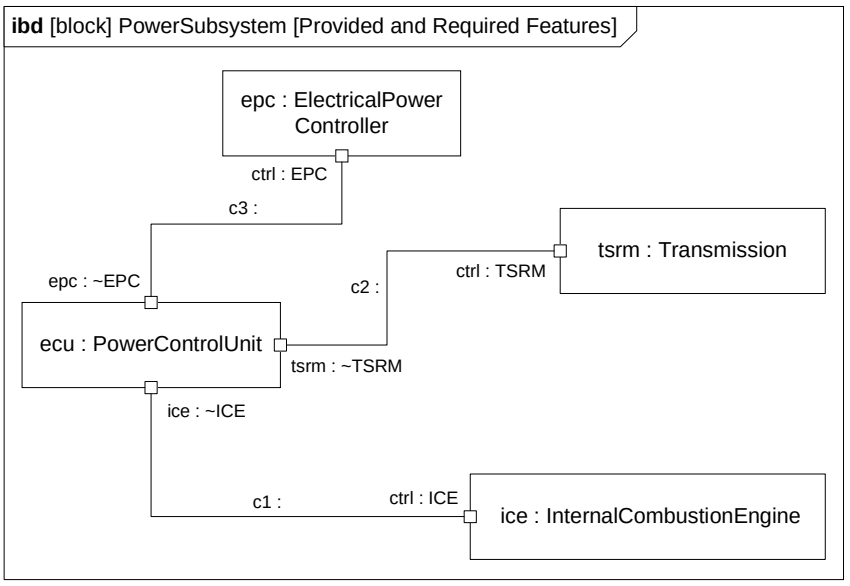


Figure 9.6 - Usage example of ports with provided and required features

9.4.2 Flow Ports and Item Flows

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

9.4.3 Ports with Flow Properties

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

9.4.4 Proxy and Full Ports

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

The ProxyPort and FullPort stereotypes can be applied at any level in a block taxonomy, whether on ports of the most general blocks, the most specialized, or at intermediate levels of generalization. Ports can be specialized through redefinition and subsetting if desired, as long they are not proxy and full at the same time, including the stereotypes they inherit. Figure 9.7 shows an example of a general block for an electrical plug specialized into two other blocks. The general block can be contained in its own package, for export to users of electrical plugs. The specialized blocks are for plug designers. This example has two designs, one using proxy ports and the other full. The proxy design adds internal parts exposed by the ports. The full design redefines the ports with specialized types. The same type is used for the internal parts of the proxy design and the redefined ports of the full design. The net result for the systems as-built are the same.

Modelers can apply stereotypes for proxy and full ports at any stage of model development, or not all if the stereotype constraints are not needed. Figure 9.7 happens to use unstereotyped ports on a general block distributed to users, and stereotyped ports on its specializations for implementation, but the modelers might have not used stereotypes at all, if they did not care whether the model met those constraints (such as no behaviors on proxy ports, or no internal binding connectors to full ports).

Unstereotyped ports do not commit to whether they are proxy or full, and do not prevent or dictate future application of the stereotypes, except for ports that violate constraints of the stereotypes. For example, if the port types on the general block in Figure 9.7 had behaviors defined, then the proxy specialization would be invalid. If the general ports had binding connectors to internal parts, then the full specialization would be invalid. If the general ports had both behaviors and internal binding connectors, then both specializations would be invalid. Unstereotyped ports have the basic functionality of stereotyped ones, including flow properties and nested ports, so they can be used as long as the modeler is not concerned with the distinction between proxy and full, and the constraints they impose.

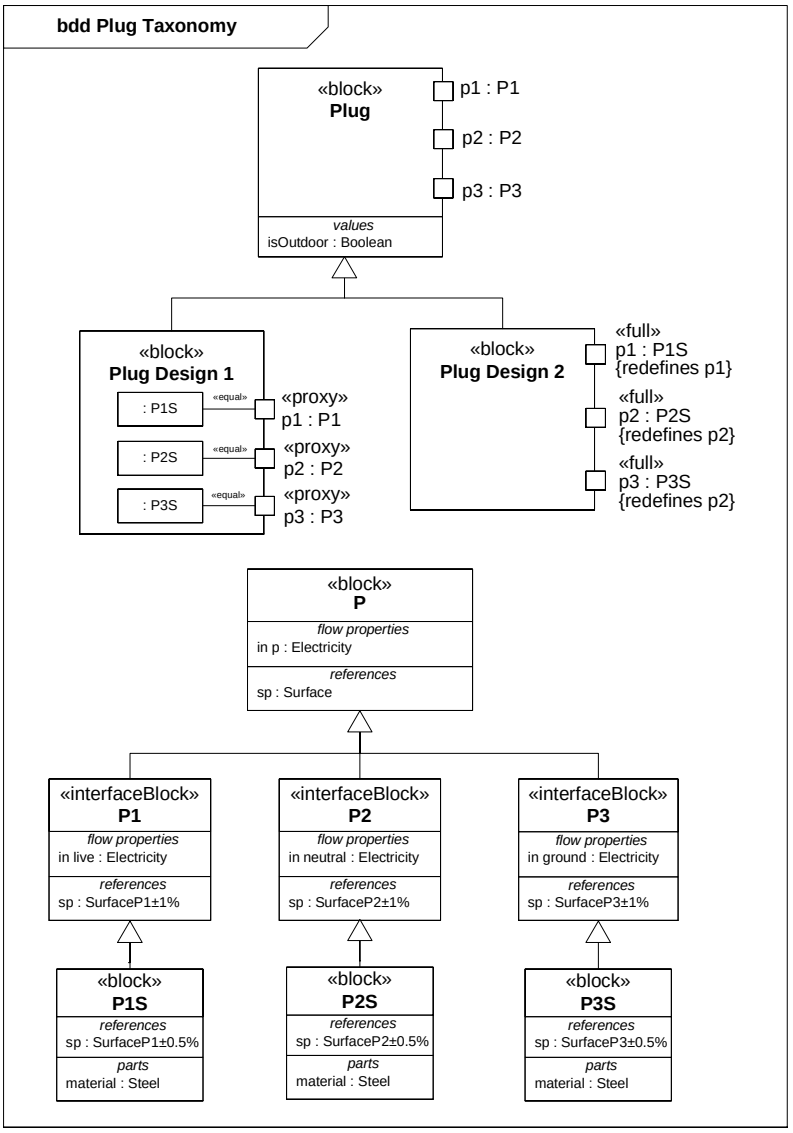


Figure 9.7 Usage example of proxy and full ports

9.4.5 Association and Port Decomposition

Figure 9.8 shows an association block Water Delivery between a bank of spigots and a faucet. Figure 9.9 shows the internal structure of Water Delivery defining connectors between the spigots in the bank and inlets on the faucet. The participant properties identify the spigot bank and faucet being connected. The end property on the stereotype refers to the corresponding association end in Figure 9.8. The type of participant properties is shown for clarity, but is always the same as the association end type and can be elided. They are shown with dashed rectangles because they are reference properties. The internal structure connects hot and cold ports of the participants.

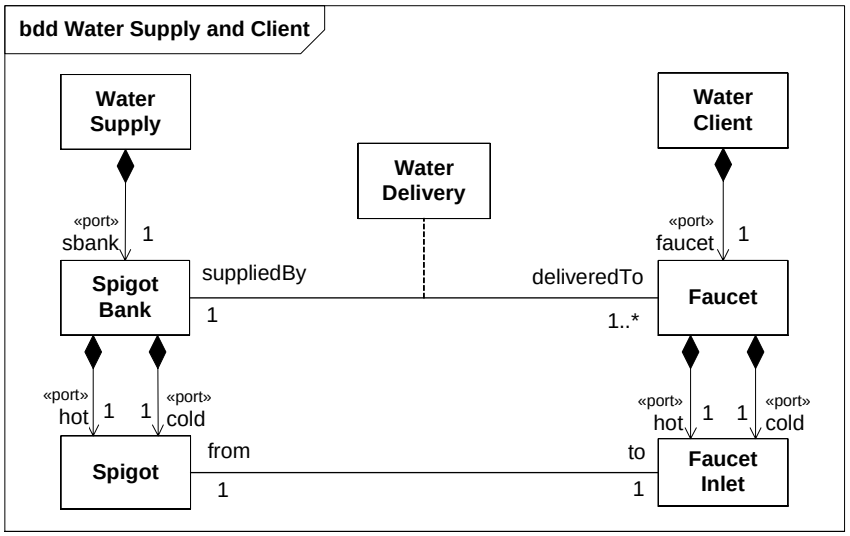


Figure 9.8 - Water Delivery association block

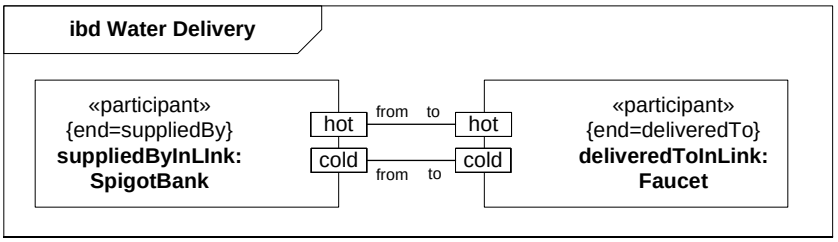


Figure 9.9 - Internal structure of Water Delivery association block

Figure 9.10 shows two views of a block House with a connector of type Water Delivery. The connector in the top view “decomposes” into the subconnectors in the lower view according to the internal structure of Water Delivery. The subconnectors relate the nested ports of :WaterSupply to the nested ports of :WaterClient.

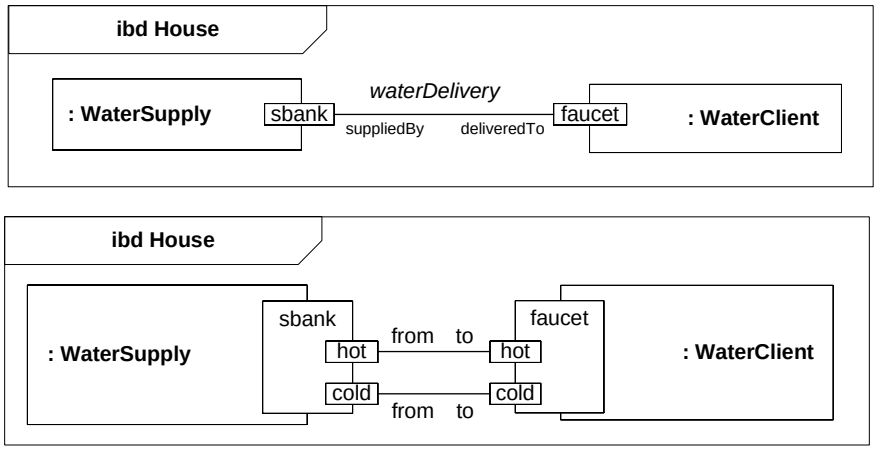


Figure 9.10 - Two views of Water Delivery connector within House block

The top portion of Figure 9.11 shows specializations of the block WaterClient into Bath, Sink, and Shower. These are used as part types in the internal structure of the block House 2 shown in the lower portion of the figure. The composite connector for Water Delivery is reused three times to establish connections between spigots on the water supply and the inlets of faucets on the bath, sink, and shower.

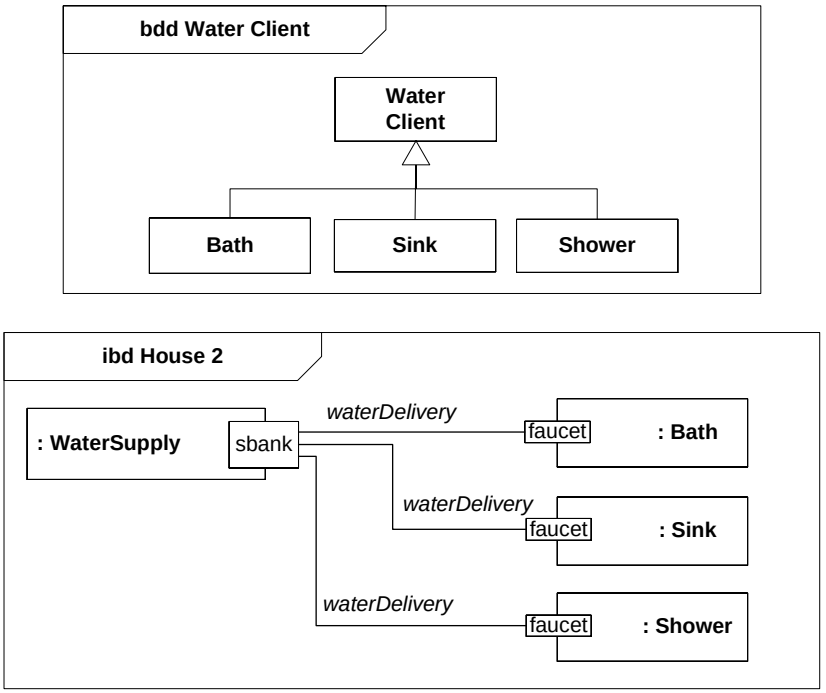


Figure 9.11 - Specializations of Water Client in house example

Figure 9.12 adds a Plumbing association block for the association between Spigot and Faucet Inlet in Figure 9.11. Figure 9.13 shows the internal structure for the Plumbing association block, which includes a pipe and two fittings (the additional part and connector definitions are omitted for brevity).

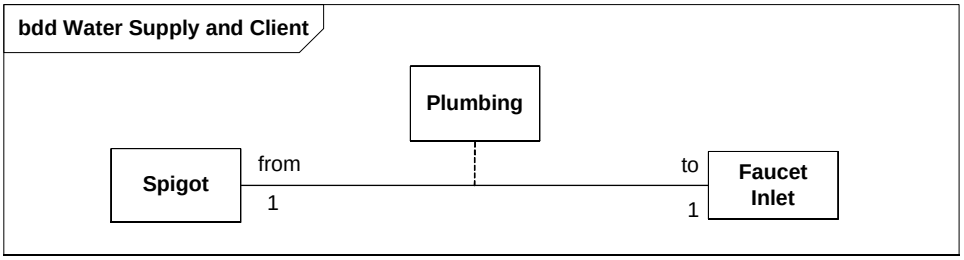


Figure 9.12 - Plumbing association block

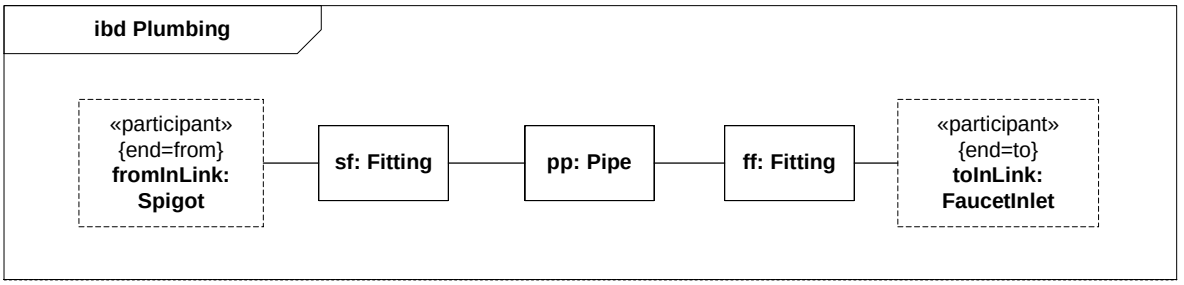


Figure 9.13 - Internal structure of Plumbing association block

Figure 9.14 modifies Figure 9.9 to use Plumbing as a connector type within the Water Delivery association block. The lower connector shows its connector property explicitly, enabling the pipe it contains to be connected to a mounting bracket (the additional part and connector definitions are omitted for brevity).

.

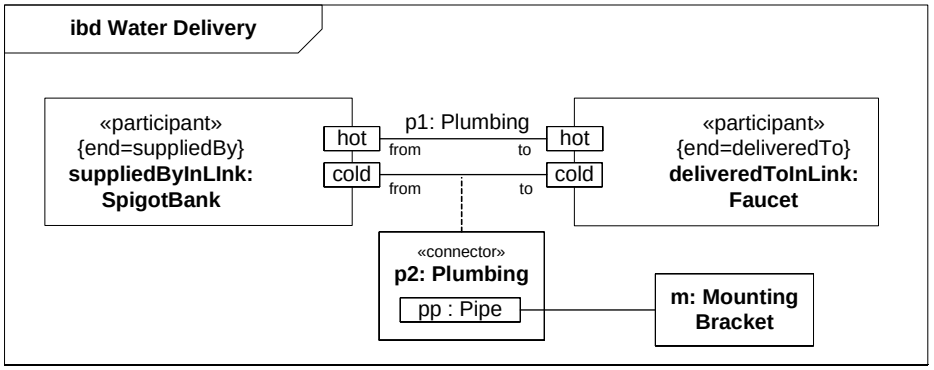


Figure 9.14 - Water Delivery association block with internal Plumbing connector

9.4.6 Item Flow Decomposition

Item flows in internal block diagrams specify flows local to a block. For example, in Figure 9.15 the connector to the output of the water heater has an item flow indicating distilled water is flowing, even though the out flow property of the water heater indicates it produces water. The water heater is fed from a water distiller in this particular usage, so the modeler knows the output will always be distilled water, rather than other kinds of water. The radiator on the left requires distilled water, and its connection to the water heater is compatible because the item flow narrows the items to distilled water. Item flows can also be more general than the actual flow, as shown by the connector on the right. The water distiller produces distilled water, but the item flow is for any kind of fluid. The connection to the water heater is compatible because it accepts any kind of water, including distilled. The item flow does not require the heater to accept any kind of fluid, because the source of flow is still producing water, regardless of the generality of the item flow.

Connectors with item flows can be decomposed by association blocks that have additional item flows. The relationship between an item flow and those in the association block is determined by the modeler. Figures 9.16 and 9.17 are examples of item flow decomposition that modelers might choose, but they are not the only possible decompositions and are not required. In Figure 9.16, the item flow classifier (EnginePart) is a supertype of the classifiers of the item flows in the decomposition. The flow properties are all in the types of the nested ports, while the composing item flow summarizes the kinds of items flowing by generalization. In Figure 9.17, the item flow classifier (Engine) composes the classifiers of the items flows in the decomposition from Figure 9.16. The port types have an additional flow property that is not in the nested ports. These are for the flow of the engine, as opposed to its parts. Constraints can be added between the flow properties for the engine and those for the parts, to indicate the flowing parts are inside the flowing engine, or are separate, for example as spare parts.

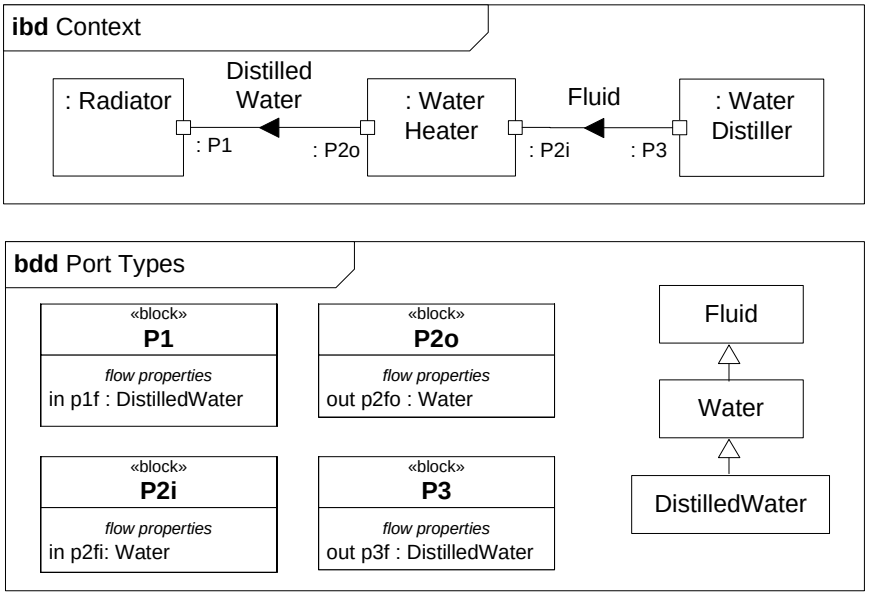


Figure 9.15 - Usage example of item flows in internal block diagrams

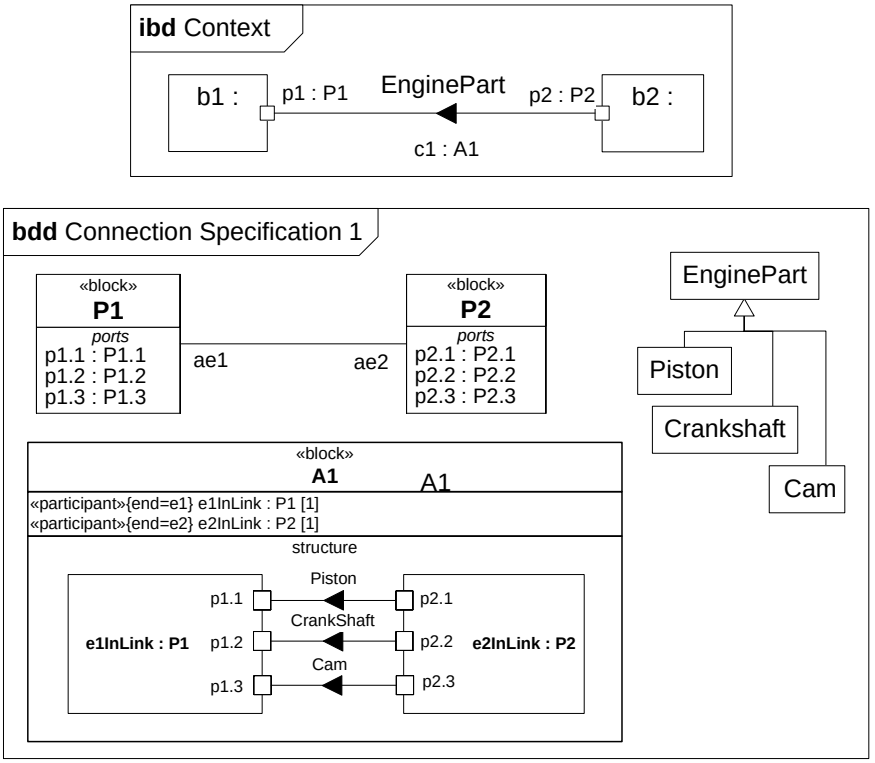


Figure 9.16 - Usage example of item flow decomposition

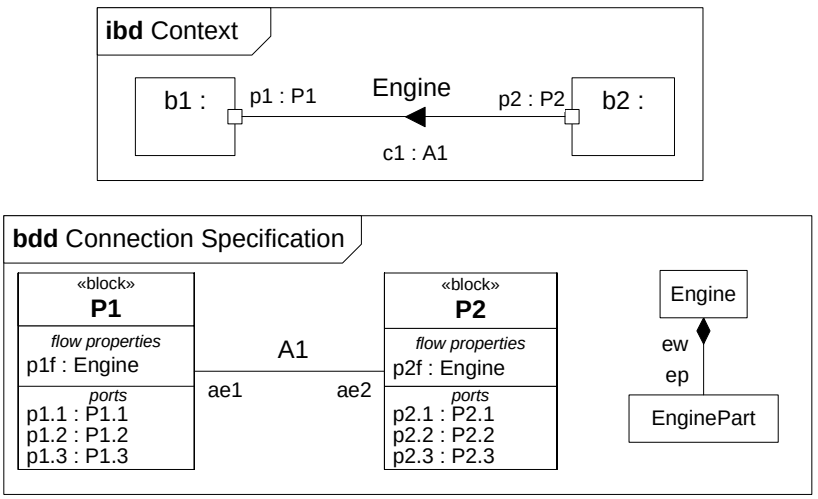


Figure 9.17 - Usage example of item flow decomposition

10 Constraint Blocks

10.1 Overview

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

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

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

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

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

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

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

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

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

10.2 Diagram Elements

10.2.1 Block Definition Diagram

The diagram elements described in this sub clause are additions to the Block Definition Diagram described in Clause 8, “Blocks.”

Table 10.1 - Graphical nodes defined in Block Definition diagrams

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

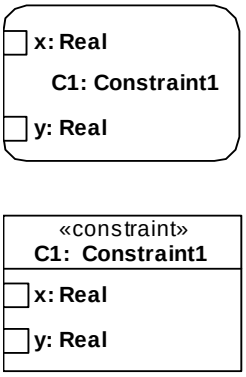
10.2.2 Parametric Diagram

The diagram elements described in this sub clause are additions to the Internal Block Diagram described in Clause 8, “Blocks.” The Parametric Diagram includes all of the notations of an Internal Block Diagram, subject only to the restrictions described in “Parametric Diagram” on page 85.

Table 10.2 - Graphical nodes defined in Parametric diagrams

Element Name	Concrete Syntax Example	Metamodel Reference
ParametricDiagram	**par** Block1 length: Real width: Real **x:** **C1: Constraint1** **y:**	**SysML::ConstraintBlocks::ConstraintBlock** **SysML::Blocks::Block**

Table 10.2 - Graphical nodes defined in Parametric diagrams

Element Name	Concrete Syntax Example	Metamodel Reference
ConstraintProperty		**SysML::ConstraintBlocks::ConstraintProperty**

10.3 UML Extensions

10.3.1 Diagram Extensions

10.3.1.1 Block Definition Diagram

10.3.1.1.1 Constraint block definition

The «constraint» keyword on a block definition states that the block is a constraint block. An expression that specifies the constraint may appear in the constraints compartment of the block definition, using either formal statements in some language, or informal statements using text. This expression can include a formal reference to a language in braces as indicated in Table 10.1. Parameters of the constraint may be shown in a compartment with the predefined compartment label “parameters.”

10.3.1.1.2 Parameters compartment

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

10.3.1.2 Parametric Diagram

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

10.3.1.2.1 Round-cornered rectangle notation for constraint property

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

10.3.1.2.2 «constraint» keyword notation for constraint property

A constraint property may be shown on a parametric diagram using a standard form of internal property rectangle with the «constraint» keyword preceding its name. Parameters are shown within a constraint property using the standard notations for internal properties. The stereotype ConstraintProperty is applied to a constraint property, but only the shorthand keyword «constraint» is used when shown on an internal property**.**

10.3.1.2.3 Small square box notation for an internal property

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

10.3.2 Stereotypes

Package ConstraintBlocks

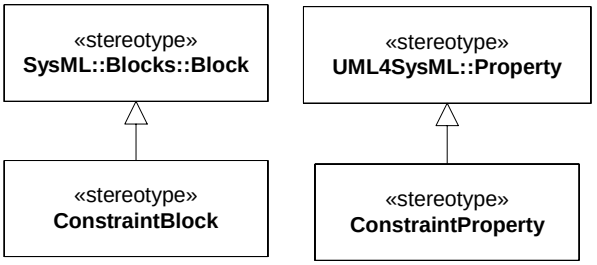


Figure 10.1 - Stereotypes defined in SysML ConstraintBlocks package

10.3.2.1 ConstraintBlock

Description

A constraint block is a block that packages the statement of a constraint so it may be applied in a reusable way to constrain properties of other blocks. A constraint block typically defines one or more constraint parameters, which are bound to properties of other blocks in a surrounding context where the constraint is used. Binding connectors, as defined in Clause 8, “Blocks,” are used to bind each parameter of the constraint block to a property in the surrounding context. All properties of a constraint block are constraint parameters, with the exception of constraint properties that hold internally nested usages of other constraint blocks.

Constraints

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

[2] Any classifier that specializes a ConstraintBlock must also have the ConstraintBlock stereotype applied.

[3] Any property of a block that is typed by a ConstraintBlock must have composite aggregation.

```
INV Block;
self.ownedAttribute->forAll(p | p.type.oclIsKindOf(ConstraintBlock) implies p.aggregation = #composite)
```

10.3.2.2 ConstraintProperty

Description

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

Constraints

[1] A property to which the ConstraintProperty stereotype is applied must be owned by a SysML Block.

[2] The ConstraintProperty stereotype must be applied to any property of a SysML Block that is typed by a ConstraintBlock.

10.4 Usage Examples

10.4.1 Definition of Constraint Blocks on a Block Definition Diagram

Constraint blocks can only be defined on a block definition diagram or a package diagram, where they must have the «constraint» keyword shown. The strings in braces in the compartment labeled “constraints” are ordinary UML constraints, using a special compartment to hold the constraint. This is shown in Figure C.31 in Annex C. These particular constraints are specified only in an informal language, but a more formal language such as OCL or MathML could also be used. The compartment labeled “parameters” shows the parameters of this constraint, which are bound on the parametric diagram.

10.4.2 Usage of Constraint Blocks on a Parametric Diagram

Figure C.29 in Annex C shows the use of constraint properties on a parametric diagram. This diagram shows the use of nested property references to the properties of the parts; parametric diagrams can make use of the nested property name notation to refer to multiple levels of nested property containment, as shown in this example. A parametric diagram is similar to an internal block diagram with the exception that the only connectors that may be shown are binding connectors connected to constraint parameters on at least one end. The Sample Problem in Annex C provides definitions of the containing EconomyContext block for which this parametric diagram is shown.

Part III - Behavioral Constructs

This Part specifies the dynamic, behavioral constructs used in SysML behavioral diagrams, including the activity diagram, sequence diagram, state machine diagram, and use case diagram. This Part includes the following clauses:

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

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

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

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

11 Activities

11.1 Overview

Activity modeling emphasizes the inputs, outputs, sequences, and conditions for coordinating other behaviors. It provides a flexible link to blocks owning those behaviors. The following is a summary of the SysML extensions to UML Activity diagrams. For additional information, see extensions for Enhanced Functional Flow Block Diagrams in Annex D, “Non-normative Extensions,” sub clause “Activity Diagram Extensions” on page 213.”

11.1.1 Control as Data

SysML extends control in activity diagrams as follows:

- In UML Activities, control can only enable actions to start. SysML extends control to support disabling of actions that are already executing. This is accomplished by providing a model library with a type for control values that are treated like data (see ControlValue in Figure 11.9).
- A control value is an input or output of a control operator, which is how control acts as data. A control operator can represent a complex logical operation that transforms its inputs to produce an output that controls other actions (see ControlOperator in Figure 11.8).

11.1.2 Continuous Systems

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

- Restrictions on the rate at which entities flow along edges in an activity, or in and out of parameters of a behavior (see Rate in Figure 11.8). This includes both discrete and continuous flows, either of material, energy, or information. Discrete and continuous flows are unified under rate of flow, as is traditionally done in mathematical models of continuous change, where the discrete increment of time approaches zero.
- Extension of object nodes, including pins, with the option for newly arriving values to replace values that are already in the object nodes (see Overwrite in Figure 11.8). SysML also extends object nodes with the option to discard values if they do not immediately flow downstream (see NoBuffer in Figure 11.8). These two extensions are useful for ensuring that the most recent information is available to actions by indicating when old values should not be kept in object nodes, and for preventing fast or continuously flowing values from collecting in an object node, as well as modeling transient values, such as electrical signals.

11.1.3 Probability

SysML introduces probability into activities as follows (see Probability in Figure 11.8):

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

11.1.4 Activities as Blocks

In UML, all behaviors including activities are classes, and their instances are executions. Behaviors can appear on block definition diagrams, and participate in generalization and associations. SysML clarifies the semantics of composition association between activities, and between activities and the type of object nodes in the activities, and defines consistency rules between these diagrams and activity diagrams. See “Activity” on page 100.”

11.1.5 Timelines

The simple time model in UML can be used to represent timing and duration constraints on actions in an activity model. These constraints can be notated as constraint notes in an activity diagram. Although the UML 2 timing diagram was not included in this version of SysML, it can complement SysML behavior diagrams to notate this information. More sophisticated SysML modeling techniques can incorporate constraint blocks from Clause 10, “Constraint Blocks” to specify resource and related constraints on the properties of the inputs, outputs, and other system properties. (Note: refer to “ObjectNode” on page 102 for constraining properties of object nodes).

11.2 Diagram Elements

11.2.1 Activity Diagram

Table 11.1 - Graphical nodes included in activity diagrams

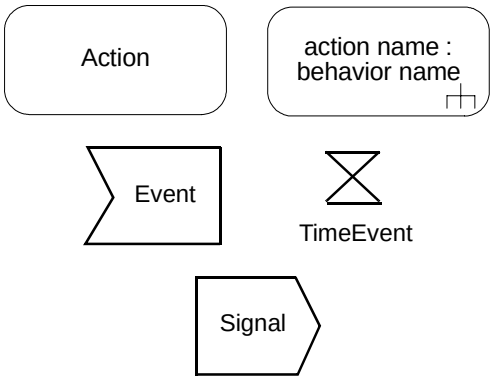
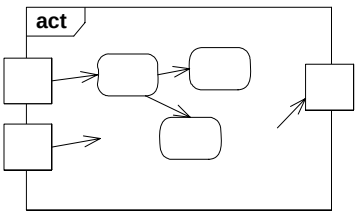
Node Name	Concrete Syntax	Abstract Syntax Reference
Action, CallBehaviorAction, AcceptEventAction, SendSignalAction		**UML4SysML::Action, UML4SysML::CallBehaviorAction UML4SysML::AcceptEventAction UML4SysML::SendSignalAction**
Activity		**UML4SysML::Activity**

Table 11.1 - Graphical nodes included in activity diagrams

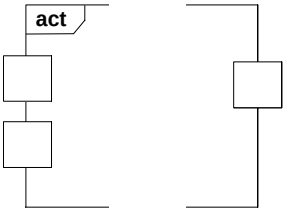
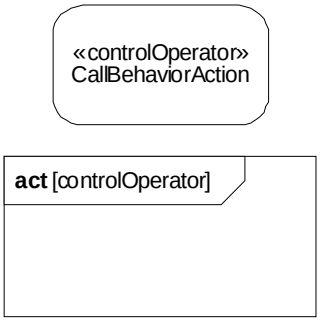
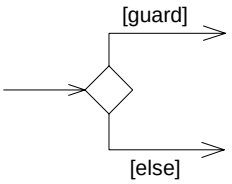
Node Name	Concrete Syntax	Abstract Syntax Reference
ActivityFinal		**UML4SysML::ActivityFinalNode**
ActivityNode	**See ControlNode and ObjectNode.**	**UML4SysML::ActivityNode**
ActivityParameterNode		**UML4SysML::ActivityParameter-Node**
ControlNode	**See DecisionNode, FinalNode, ForkNode, Initial-Node,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

Table 11.1 - Graphical nodes included in activity diagrams

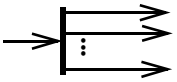
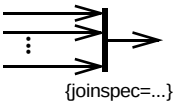
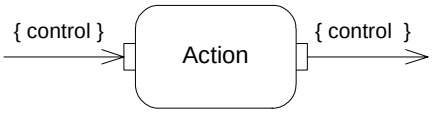
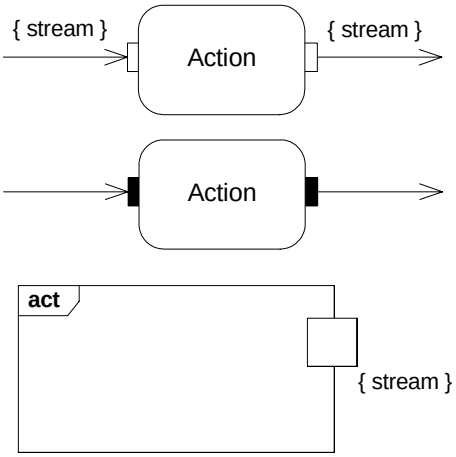
Node Name	Concrete Syntax	Abstract Syntax Reference
ForkNode		**UML4SysML::ForkNode**
InitialNode		**UML4SysML::InitialNode**
JoinNode		**UML4SysML::JoinNode**
isControl		**UML4SysML::Pin.isControl**
isStream		**UML4SysML::Parameter.isStream**

Table 11.1 - Graphical nodes included in activity diagrams

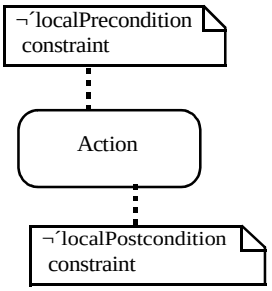
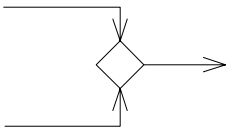
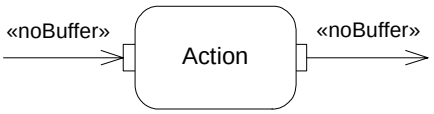
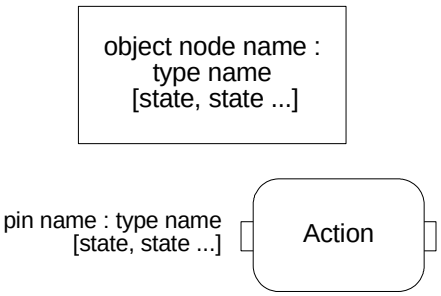
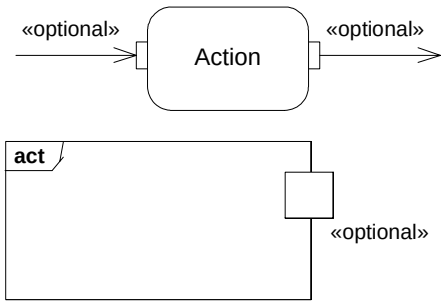
Node Name	Concrete Syntax	Abstract Syntax Reference
Local pre- and postconditions		**UML4SysML::Action.local Precondition, UML4SysML::Action.local Postcondition**
MergeNode		**UML4SysML::MergeNode**
NoBuffer		**SysML::Activities::NoBuffer**
ObjectNode		**UML4SysML::OjectNode and its children, SysML:: Activities::ObjectNode**
Optional		**SysML::Activities::Optional**

Table 11.1 - Graphical nodes included in activity diagrams

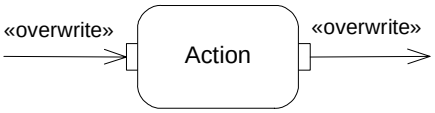
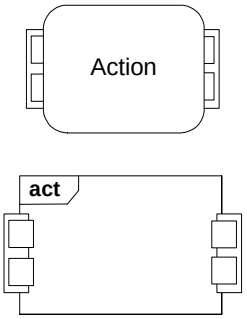
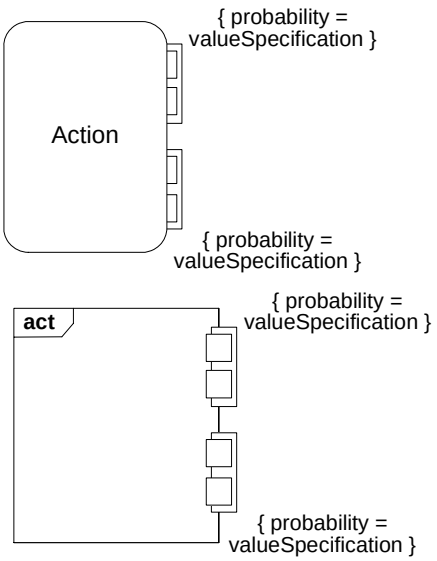
Node Name	Concrete Syntax	Abstract Syntax Reference
OverWrite		**SysML::Activities::Overwrite**
ParameterSet		**UML4SysML::ParameterSet**
Probability		**SysML::Activities::Probability**

Table 11.1 - Graphical nodes included in activity diagrams

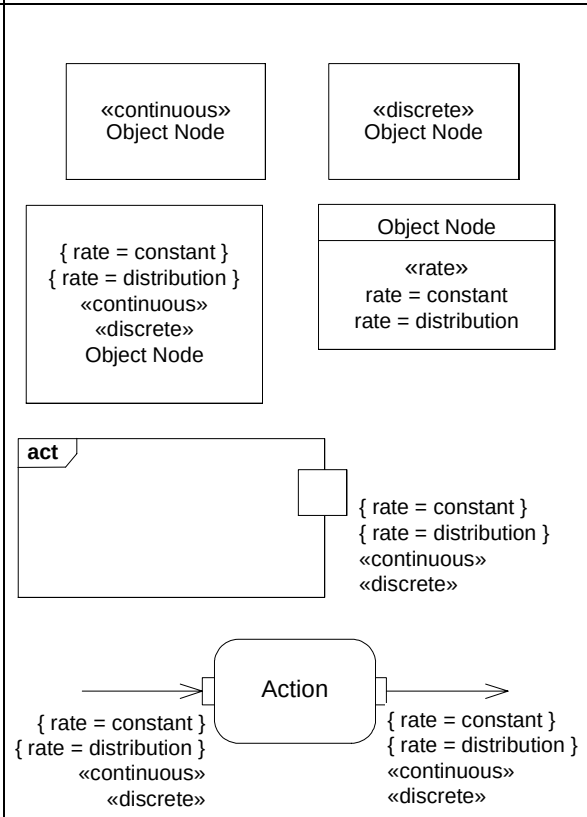
Node Name	Concrete Syntax	Abstract Syntax Reference
Rate		**SysML::Activities::Rate, SysML::Activities::Continuous, SysML::Activities::Discrete**

Table 11.2 - Graphical paths included in activity diagrams

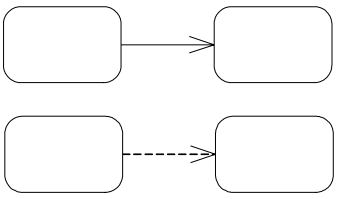
Path Name	Concrete Syntax	Abstract Syntax Reference
ActivityEdge	**See ControlFlow and ObjectFlow**	**UML4SysML::ActivityEdge**
ControlFlow		**UML4SysML::ControlFlow SysML::Activities::ControlFlow**

Table 11.2 - Graphical paths included in activity diagrams

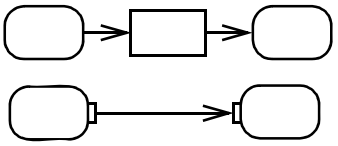
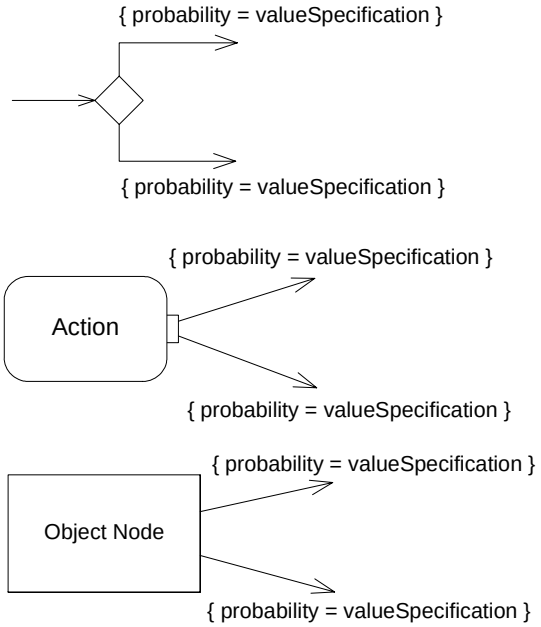
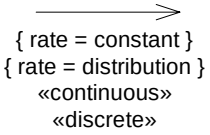
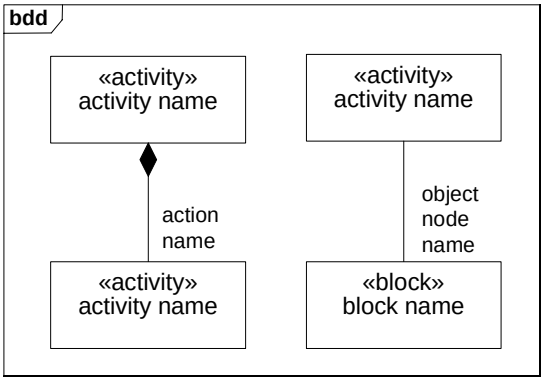
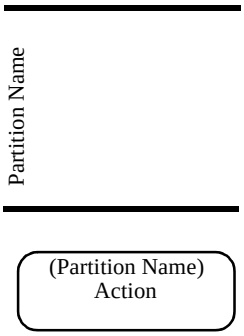
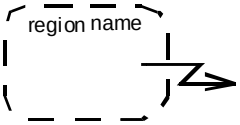
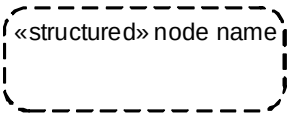
Path Name	Concrete Syntax	Abstract Syntax Reference
ObjectFlow		**UML4SysML::ObjectFlow**
Probability		**SysML::Activities::Probability**
Rate		**SysML::Activities::Rate, SysML::Activities::Continuous, SysML::Activities::Discrete**

Table 11.3 - Other graphical elements included in activity diagrams

Element Name	Concrete Syntax	Abstract Syntax Reference
In Block Definition Diagrams, Activity, Association		**SysML::Activities, Diagram Usage for Block Definition Diagrams**
ActivityPartition		**UML4SysML::ActivityPartition**
InterruptibleActivity Region		**UML4SysML::InterruptibleActivity-Region**
StructuredActivityNode		**UML4SysML::StructuredActivity Node**

11.3 UML Extensions

11.3.1 Diagram Extensions

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

11.3.1.1 Activity

11.3.1.1.1 Notation

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

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

Combining the two aspects above, when an activity invokes other activities, they can be associated by a composition association, with the invoking activity on the whole end, and the invoked activity on the part end. If an execution of an activity on the whole end is terminated, then the executions of the activities on the part end are also terminated. The upper multiplicity on the part end restricts the number of concurrent synchronous executions of the behavior that can be invoked by the containing activity. The lower multiplicity on the part end is always zero, because there will be some time during the execution of the containing activity that the lower level activity is not executing. See Constraints below.

Activities in block definition diagrams appear as regular blocks, except the «activity» keyword may be used to indicate the Block stereotype is applied to an activity, as shown in Figure 11.1. See example in “Usage Examples” on page 108. This provides a means for representing activity decomposition in a way that is similar to classical functional decomposition hierarchies. The names of the CallBehaviorActions that correspond to the association can be used as end names of the association on the part end. Activities in block definition diagrams can also appear with the same notation as CallBehaviorAction, except the rake notation can be omitted, if desired. Also see use of activities in block definition diagrams that include ObjectNodes.

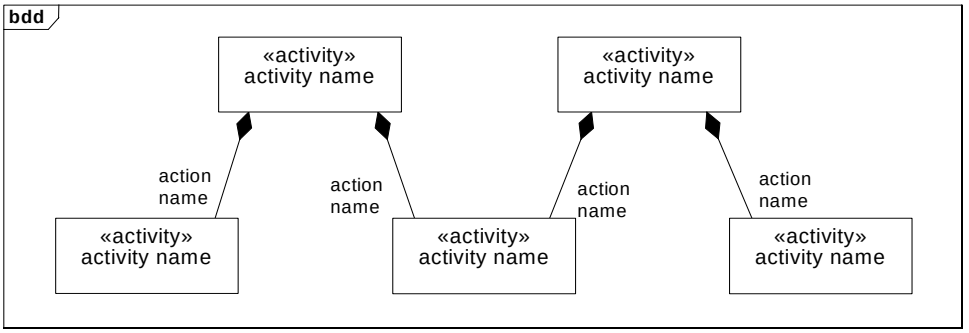


Figure 11.1 - Block definition diagram with activities as blocks

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

Constraints

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

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

```
self.ownedAttribute->forAll(a | (a.type.oclIsKindOf(Activity) and
                                 aggregation= #composite)
   implies self.node->exists(n | n.oclIsKindOf(CallBehaviorAction)
      and n.isSynchronous= #true and a.type = n.behavior and
      (( n.name->notEmpty() and n.name=a.name) or ( n.name->empty()
      and  n.behavior.name = a.name)))
```

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

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

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

11.3.1.2 CallBehaviorAction

Stereotypes applied to behaviors may appear on the notation for CallBehaviorAction when invoking those behaviors, as shown in Figure 11.2.

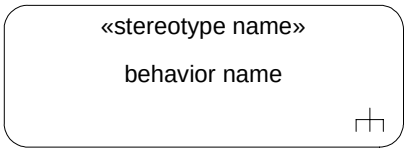


Figure 11.2 - CallBehaviorAction notation.with behavior stereotype

CallBehaviorActions in activity diagrams may optionally show the action name with the name of the invoked behavior using the colon notation shown in Figure 11.3.

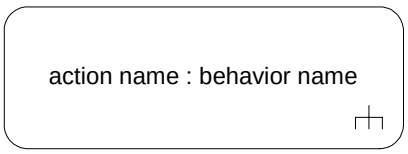


Figure 11.3 - CallBehaviorAction notation.with action name

11.3.1.3 ControlFlow

11.3.1.3.1 Presentation Option

Control flow may be notated with a dashed line and stick arrowhead, as shown in Figure 11.4.

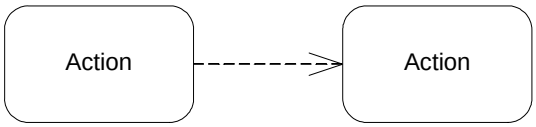


Figure 11.4 - Control flow notation

11.3.1.4 ObjectNode

11.3.1.4.1 Notation

See “Activity” on page 100” with regard to activities appearing in block definition diagrams. Associations can be used between activities and classifiers (blocks or value types) that are the type of object nodes in the activity, as shown in Figure 11.5. This supports linking the execution of the activity with items that are flowing through the activity and happen to be contained by the object node at the time the link exists. The names of the object node that correspond to the association can be used as end names of the association on the end towards the object node type. Like any association end or property these can be the subject of parametric constraints, design values, units, and quantity kinds. The upper multiplicity on the object node end restricts the number of instances of the item type that can reside in the object node at one time, which must be lower than the maximum amount allowed by the object node itself. The lower multiplicity on the object node end is always zero, because there will be some time during the execution of the containing activity that there is no item in the object node. The associations may be composition if the intention is to delete instances of the classifier flowing the activity when the activity is terminated. See example in “Usage Examples” on page 108.

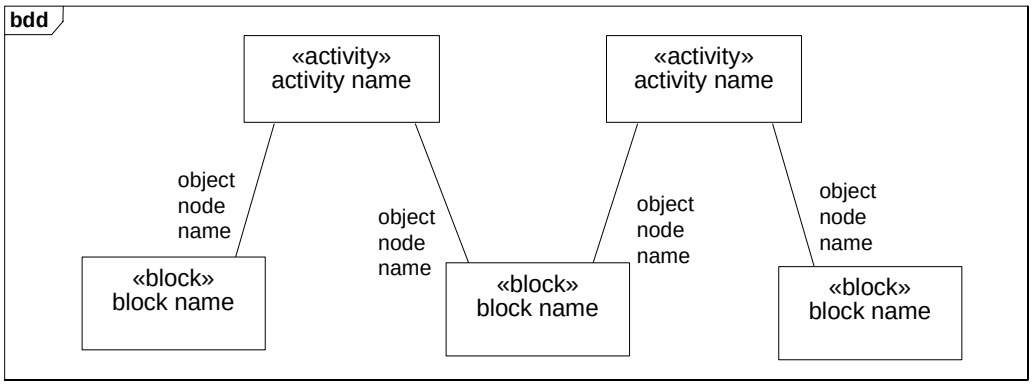


Figure 11.5 - Block definition diagram with activities as blocks associated with types of object nodes

Object nodes in activity diagrams can optionally show the node name with the name of the type of the object node as shown in Figure 11.6.

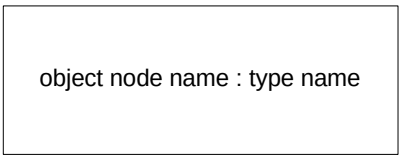


Figure 11.6 - ObjectNode notation in activity diagrams

Stereotypes applying to parameters can appear on object nodes in activity diagrams, as shown in Figure 11.7, when the object node notation is used as a shorthand for pins. The stereotype applies to all parameters corresponding to the pins notated by the object node. Stereotype applying to object nodes can also appear in object nodes, and applies to all the pins notated by the object node.

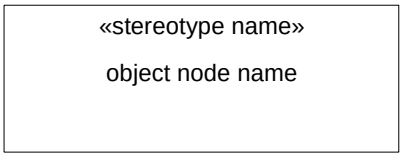


Figure 11.7 - ObjectNode notation in activity diagrams

Constraints

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

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

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

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

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

11.3.2 Stereotypes

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

Package Activities

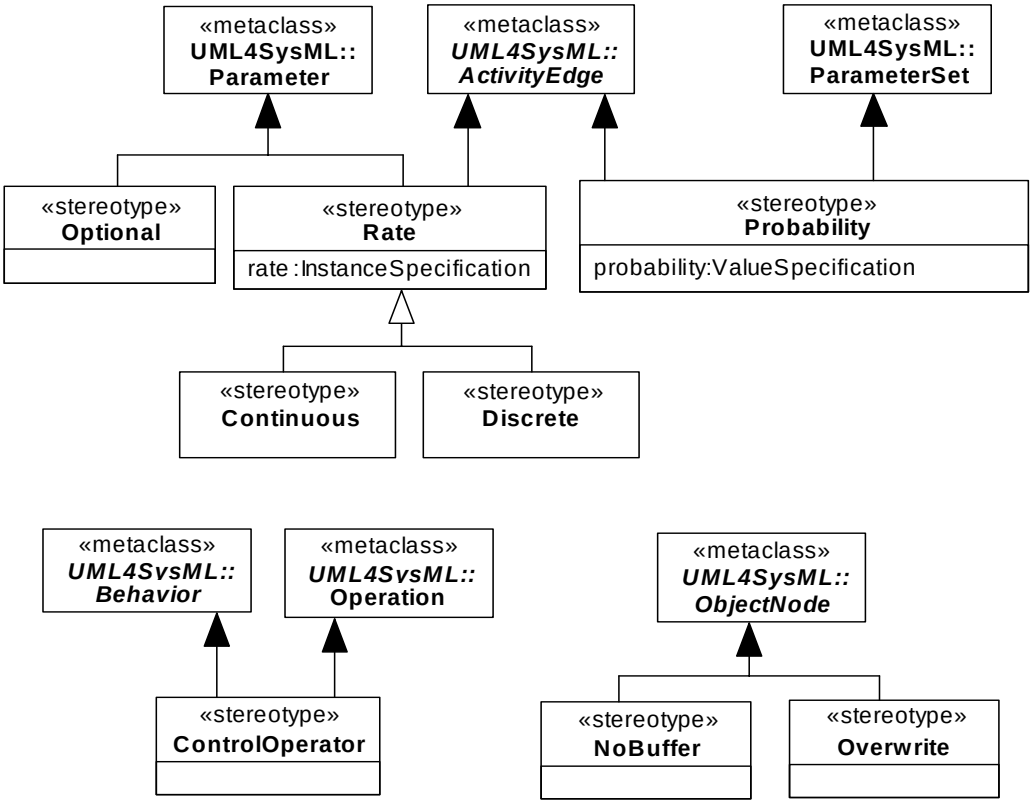


Figure 11.8 - Abstract Syntax for SysML Activity Extensions

11.3.2.1 Continuous

Continuous rate is a special case of rate of flow (see Rate) where the increment of time between items approaches zero. It is intended to represent continuous flows that may correspond to water flowing through a pipe, a time continuous signal, or continuous energy flow. It is independent from UML streaming, see “Rate” on page 107. A streaming parameter may or may not apply to continuous flow, and a continuous flow may or may not apply to streaming parameters.

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

11.3.2.2 ControlOperator

Description

A control operator is a behavior that is intended to represent an arbitrarily complex logical operator that can be used to enable and disable other actions. When the «controlOperator» stereotype is applied to behaviors, the behavior takes control values as inputs or provides them as outputs, that is, it treats control as data (see “ControlValue” on page 107). When the «controlOperator» stereotype is not applied, the behavior may not have a parameter typed by ControlValue. The «controlOperator» stereotype also applies to operations with the same semantics.

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

Constraints

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

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

11.3.2.3 Discrete

Description

Discrete rate is a special case of rate of flow (see “Rate” on page 107) where the increment of time between items is non-zero. Examples include the production of assemblies in a factory and signals set at periodic time intervals.

Constraints

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

11.3.2.4 NoBuffer

Description

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

Constraints

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

11.3.2.5 Overwrite

Description

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

Constraints

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

11.3.2.6 Optional

Description

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

Constraints

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

11.3.2.7 Probability

Description

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

When the «probability» stereotype is applied to output parameter sets, it gives the probability the parameter set will be given values at runtime. These must be between zero and one inclusive, and add up to one for output parameter sets of the same behavior at the time the probabilities are used.

Constraints

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

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

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

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

11.3.2.8 Rate

Description

When the «rate» stereotype is applied to an activity edge, it specifies the expected value of the number of objects and values that traverse the edge per time interval, that is, the expected value rate at which they leave the source node and arrive at the target node. It does not refer to the rate at which a value changes over time. When the stereotype is applied to a parameter, the parameter must be streaming, and the stereotype gives the number of objects or values that flow in or out of the parameter per time interval while the behavior or operation is executing. Streaming is a characteristic of UML behavior parameters that supports the input and output of items while a behavior is executing, rather than only when the behavior starts and stops. The flow may be continuous or discrete, see the specialized rates in “Continuous” on page 104 and “Discrete” on page 105. The «rate» stereotype has a rate property of type InstanceSpecification. The values of this property must be instances of classifiers stereotyped by «valueType» or «distributionDefinition», see Clause 8, “Blocks.” In particular, the denominator for units used in the rate property must be time units.

Constraints

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

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

11.3.3 Model Libraries

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

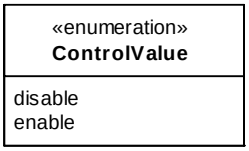


Figure 11.9 - Control values

11.3.3.1 ControlValue

Description

The ControlValue enumeration is a type for treating control values as data (see “ControlOperator” on page 105) and for UML control pins. It can be used as the type of behavior and operation parameters, object nodes, and attributes, and so on. The possible runtime values are given as enumeration literals. Modelers can extend the enumeration with additional literals, such as suspend, resume, with their own semantics.

The disable literal means a termination of an executing behavior that can only be started again from the beginning (compare to suspend). The enable literal means to start a new execution of a behavior (compare to resume).

Constraints

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

11.4 Usage Examples

The following examples illustrate modeling continuous systems (see “Continuous Systems” in “Control as Data”). Figure 11.10 shows a simplified model of driving and braking in a car that has an automatic braking system. Turning the key on has a duration constraint specifying that this action lasts no more than 0.1 seconds. Turning the key on starts two behaviors, Driving and Braking. These behaviors execute until the key is turned off, using streaming parameters to communicate with other behaviors. The Driving behavior outputs a brake pressure continuously to the Braking behavior while both are executing, as indicated by the «continuous» rate and streaming properties (streaming is a characteristic of UML behavior parameters that supports the input and output of items while a behavior is executing, rather than only when the behavior starts and stops). Brake pressure information also flows to a control operator that outputs a control value to enable or disable the Monitor Traction behavior. No pins are used on Monitor Traction, so once it is enabled, the continuously arriving enable control values from the control operator have no effect, per UML semantics. When the brake pressure goes to zero, disable control values are emitted from the control operator. The first one disables the monitor, and the rest have no effect. While the monitor is enabled, it outputs a modulation frequency for applying the brakes as determined by the ABS system. The rake notations on the control operator and Monitor Traction indicate they are further defined by activities, as shown in Figures 11.11 and 11.12. An alternative notation for this activity decomposition is shown in Figure 11.13.

The duration constraint notation associated with the Turn Key To On action is supported by the UML Simple Time model. The Operate Car activity owns a duration constraint specifying that the “Turn Key To On” action lasts no more than 0.1 seconds. The concrete UML element used in this example is a DurationConstraint owned by Operate Car that constrains the Turn Key To On action. The DurationConstraint owns a DurationInterval, which specifies that the action is constrained to last between 0 seconds and 0.1 seconds (both being Duration expressions).

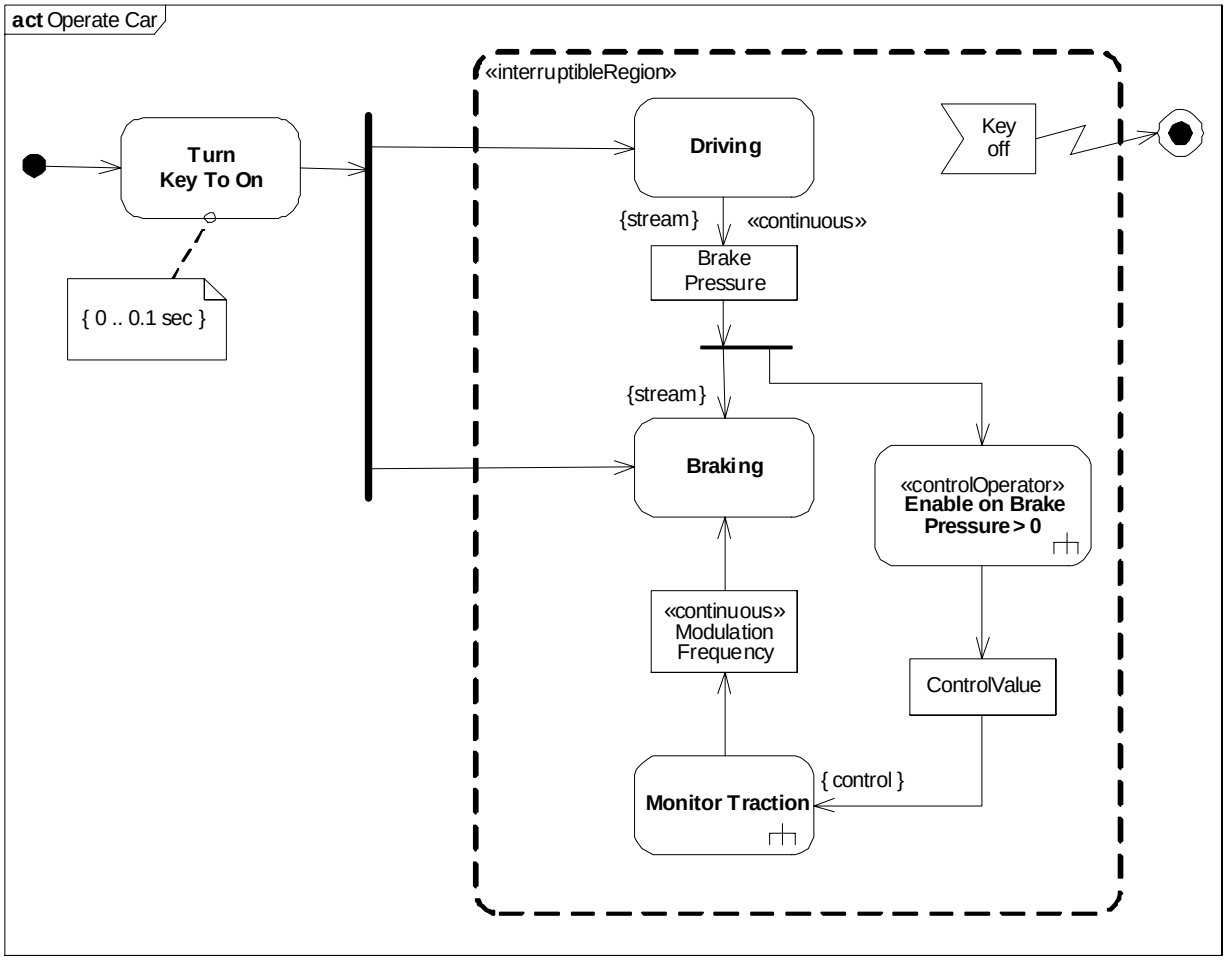


Figure 11.10 - Continuous system example 1

The activity diagram for Monitor Traction is shown in Figure 11.11. When Monitor Traction is enabled, it begins listening for signals coming in from the wheel and accelerometer, as indicated by the signal receipt symbols on the left, which begin listening automatically when the activity is enabled. A traction index is calculated every 10 ms, which is the slower of the two signal rates. The accelerometer signals come in continuously, which means the input to Calculate Traction does not buffer values. The result of Calculate Traction is filtered by a decision node for a threshold value and Calculate Modulation Frequency determines the output of the activity.

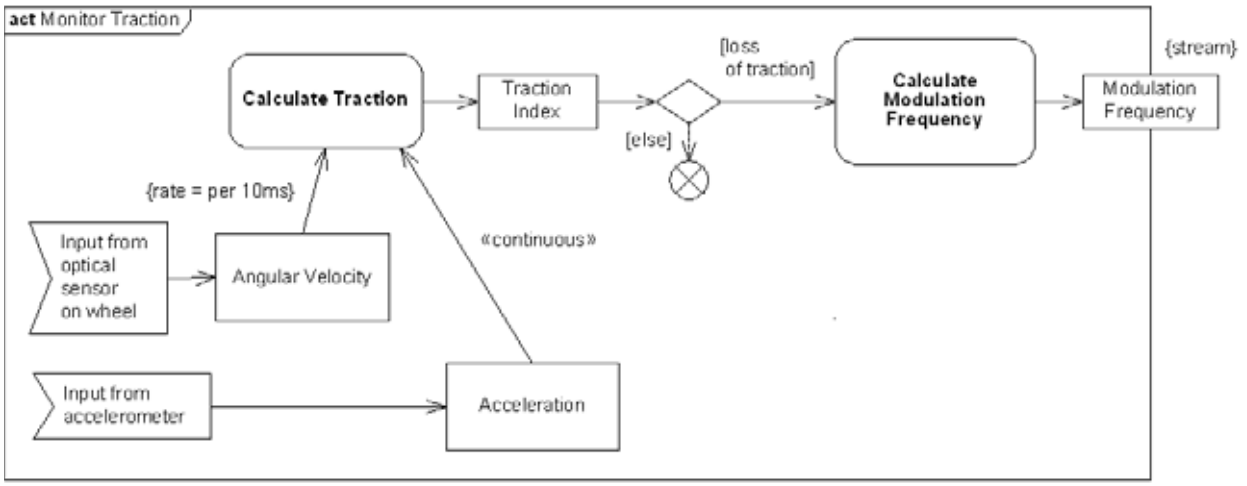


Figure 11.11 - Continuous system example 2

The activity diagram for the control operator Enable on Brake Pressure > 0 is shown in Figure 11.12. The decision node and guards determine if the brake pressure is greater than zero, and flow is directed to value specification actions that output an enabling or disabling control value from the activity. The edges coming out of the decision node indicate the probability of each branch being taken.

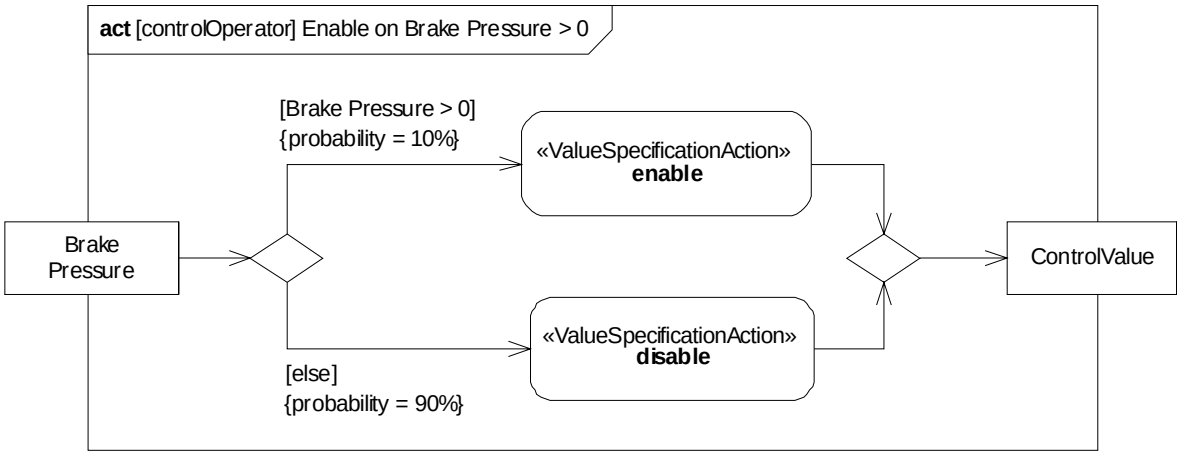


Figure 11.12 Continuous system example 3

Figure 11.13 shows a block definition diagram with composition associations between the activities in Figures 11.10, 11.11, and 11.12, as an alternative way to show the activity decomposition of Figures 11.10, 11.11, and 11.12. Each instance of Operating Car is an execution of that behavior. It owns the executions of the behaviors it invokes synchronously, such as Driving. Like all composition, if an instance of Operating Car is destroyed, terminating the execution, the executions it owns are also terminated.

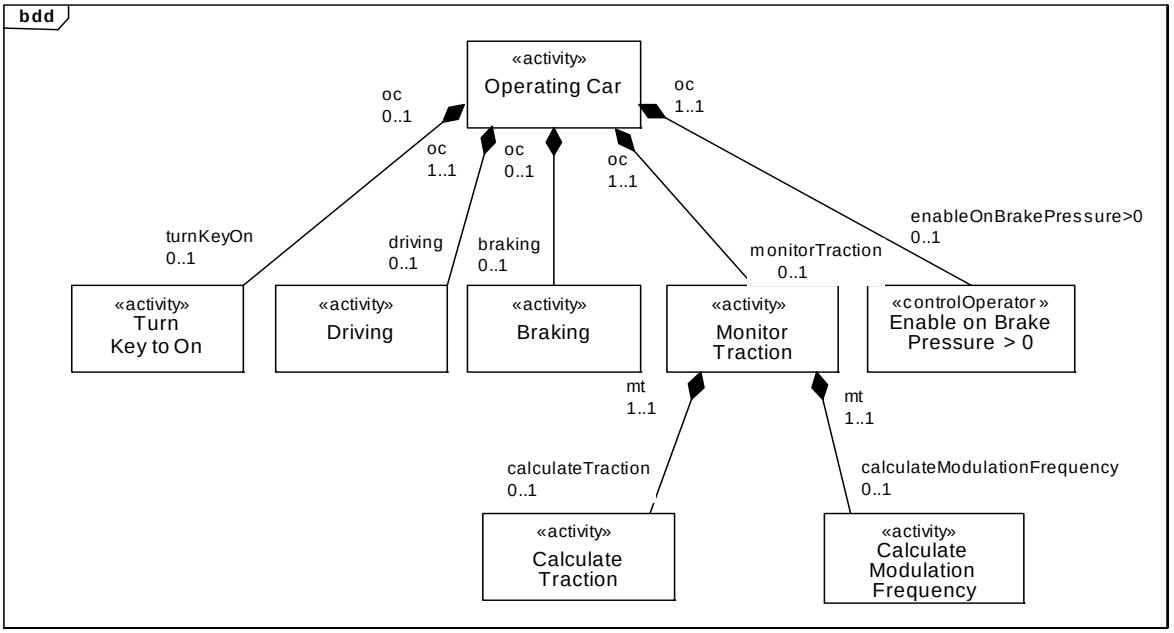


Figure 11.13 - Example block definition diagram for activity decomposition

Figure 11.14 shows a block definition diagram with composition associations between the activity in Figure 11.10 and the types the object nodes in that activity. In an instance of Operating Car, which is one execution of it, instances of Brake Pressure and Modulation Frequency are linked to the execution instance when they are in the object nodes of the activity.

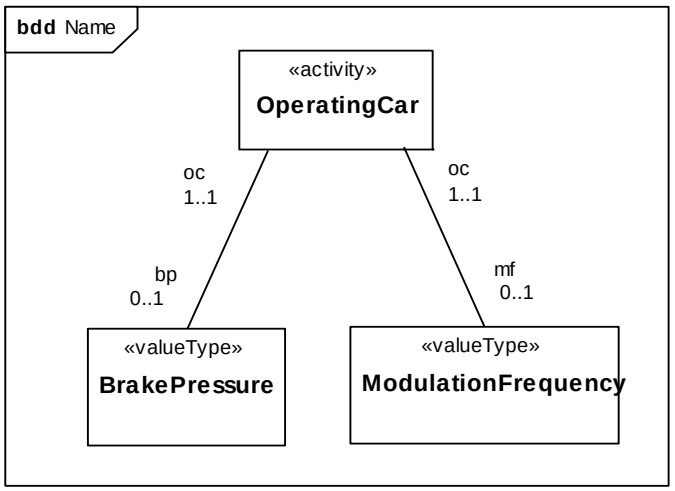


Figure 11.14 - Example block definition diagram for object node types

12 Interactions

12.1 Overview

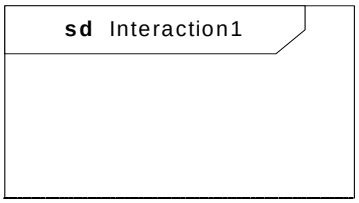
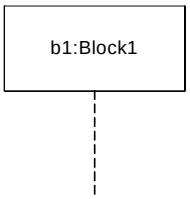
Interactions are used to describe interactions between entities. UML 2.1 Interactions are supported by four diagram types including the Sequence diagram, Communications diagram, Interaction Overview diagram, and Timing diagram. The Sequence diagram is the most common of the Interaction diagrams. SysML includes the Sequence diagram only and excludes the Interaction Overview diagram and Communication diagram, which were considered to offer significantly overlapping functionality without adding significant capability for system modeling applications. The Timing diagram is also excluded due to concerns about its maturity and suitability for systems engineering needs.

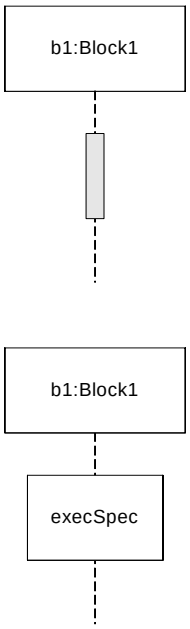
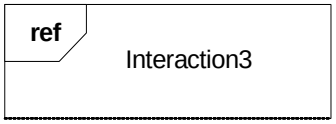
The Sequence diagram describes the flow of control between actors and systems (blocks) or between parts of a system. This diagram represents the sending and receiving of messages between the interacting entities called lifelines, where time is represented along the vertical axis. The sequence diagrams can represent highly complex interactions with special constructs to represent various types of control logic, reference interactions on other sequence diagrams, and decomposition of lifelines into their constituent parts.

12.2 Diagram Elements

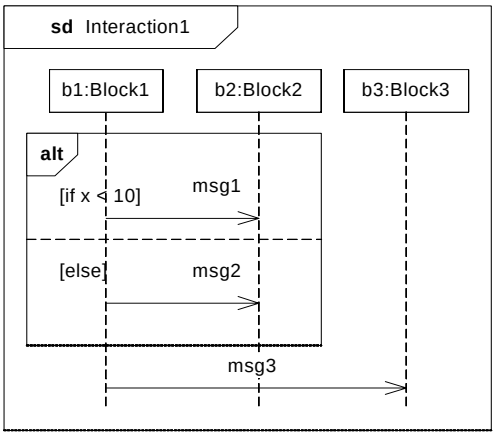
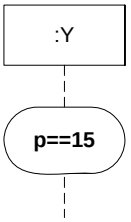
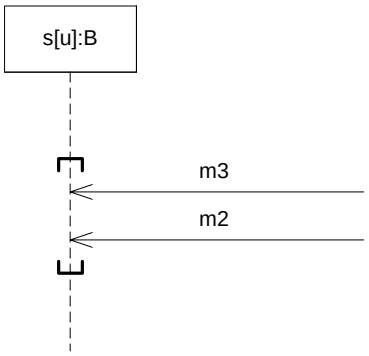
12.2.1 Sequence Diagram

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

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

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

1. Table is compliant with UML 2.1 Superstructure document.

Node Name	Concrete Syntax	Abstract Syntax Reference
CombinedFragment		**UML4SysML::CombinedFragment** A combined fragment is defined by an interaction operator and corresponding interaction operands. Interaction Operators include: **seq** - Weak Sequencing **alt** - Alternatives **opt** - Option **break** - Break **par** - Parallel **strict** - Strict Sequencing **loop** - Loop **critical** - Critical Region **neg** - Negative **assert** - Assertion **ignore** - Ignore **consider** - Consider
StateInvariant / Continuations		**UML4SysML::Continuation** **UML4SysML::StateInvariant**
Coregion		**UML4SysML::CombinedFragment** (under *parallel*)

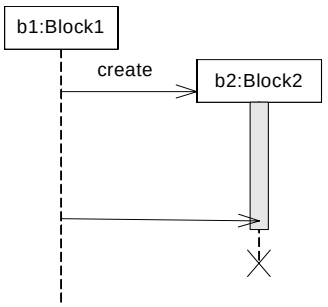
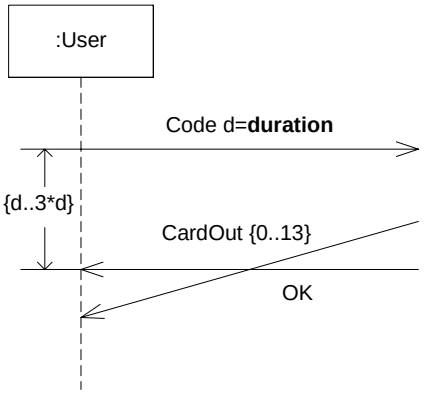
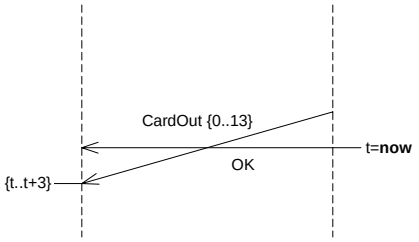
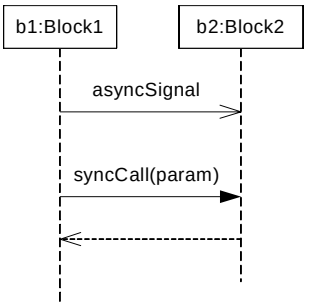
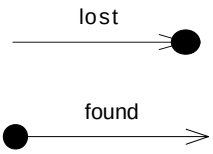
Node Name	Concrete Syntax	Abstract Syntax Reference
CreationEvent **DestructionEvent**		**UML4SysML::CreationEvent** **UML4SysML::DestructionEvent**
DurationConstraint **Duration** **Observation**		**UML4SysML::Interactions**
TimeConstraint **TimeObservation**		**UML4SysML::Interactions**

Table 12.2 - Graphical paths included in sequence diagram

Path Name	Concrete Syntax	Abstract Syntax Reference
Message		**UML4SysML::Message**
Lost Message **Found Message**		**UML4SysML::Message**
GeneralOrdering		**UML4SysML::GeneralOrdering**

12.3 UML Extensions

12.3.1 Diagram Extensions

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

12.3.1.1 Exclusion of Communication Diagram, Interaction Overview Diagram, and Timing Diagram

Communication diagrams and Interaction Overview diagrams are excluded from SysML. The other behavioral diagram representations were considered to provide sufficient coverage without introducing these diagram kinds. Timing diagrams are also excluded due to concerns about their maturity and suitability for systems engineering needs.

12.4 Usage Examples

12.4.1 Sequence Diagrams

Figure C.7 in Annex C illustrates the overall system behavior for operating the vehicle in Sequence diagram format. To manage the complexity, a hierarchical sequence diagram is used which refers to other interactions that further elaborate the system behavior. (“ref StartVehicleBlackBox”) CombinedFragments are used to illustrate that steering can take place at the same time as controlling the speed and that controlling speed can be either idling, accelerating/cruising, or braking.

Figure C.9 in Annex C shows an interaction that includes events and messages communicated between the driver and vehicle during the starting of the vehicle. The “hybridSUV” lifeline represents another interaction which further elaborates what happens inside the “hybridSUV” when the vehicle is started.

Figure C.10 in Annex C shows the sequence of communication that occurs inside the HybridSUV when the vehicle is started successfully.

13 State Machines

13.1 Overview

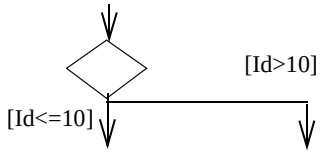
The StateMachine package defines a set of concepts that can be used for modeling discrete behavior through finite state transition systems. The state machine represents behavior as the state history of an object in terms of its transitions and states. The activities that are invoked during the transition, entry, and exit of the states are specified along with the associated event and guard conditions. Activities that are invoked while in the state are specified as “do Activities,” and can be either continuous or discrete. A composite state has nested states that can be sequential or concurrent.

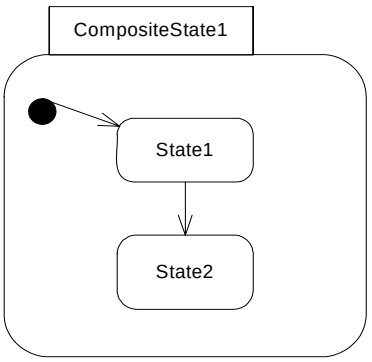
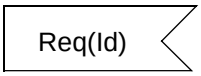
The UML concept of protocol state machines is excluded from SysML to reduce the complexity of the language. The standard UML state machine concept (called behavior state machines in UML) are thought to be sufficient for expressing protocols.

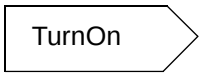
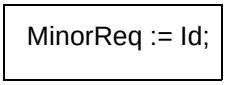
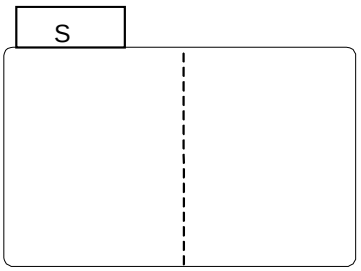
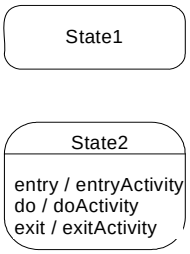
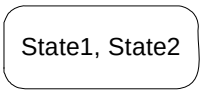
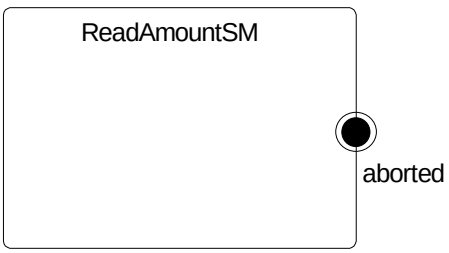
13.2 Diagram Elements

13.2.1 State Machine Diagram

Table 13.1 - Graphical nodes included in state machine diagrams

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

Node Name	Concrete Syntax	Abstract Syntax Reference
Composite state		**UML4SysML::State**
Entry point		**UML4SysML::PseudoState**
Exit point		**UML4SysML::PseudoState**
Final state		**UML4SysML::FinalState**
History, Deep Pseudo state		**UML4SysML::PseudoState**
History, Shallow pseudo state		**UML4SysML::PseudoState**
Initial pseudo state		**UML4SysML::PseudoState**
Junction pseudo state		**UML4SysML::PseudoState**
Receive signal action		**UML4SysML::Transition**

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

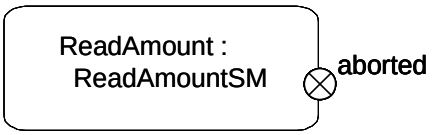
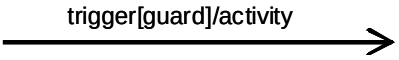
Node Name	Concrete Syntax	Abstract Syntax Reference
Terminate node		**UML4SysML::PseudoState**
Submachine state		**UML4SysML::State**

Table 13.2 - Graphical paths included in state machine diagrams

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

13.3 UML Extensions

None.

13.4 Usage Examples

13.4.1 State Machine Diagram

The high level states or modes of the HybridSUV including the events that trigger changes of state are illustrated in the state machine diagram in Figure C.8 in Annex C.

14 Use Cases

14.1 Overview

The use case diagram describes the usage of a system (subject) by its actors (environment) to achieve a goal, that is realized by the subject providing a set of services to selected actors. The use case can also be viewed as functionality and/or capabilities that are accomplished through the interaction between the subject and its actors. Use case diagrams include the use case and actors and the associated communications between them. Actors represent classifier roles that are external to the system that may correspond to users, systems, and or other environmental entities. They may interact either directly or indirectly with the system. The actors are often specialized to represent a taxonomy of user types or external systems.

The use case diagram is a method for describing the usages of the system. The association between the actors and the use case represent the communications that occur between the actors and the subject to accomplish the functionality associated with the use case. The subject of the use case can be represented via a system boundary. The use cases that are enclosed in the system boundary represent functionality that is realized by behaviors such as activity diagrams, sequence diagrams, and state machine diagrams.

The use case relationships are “communication,” “include,” “extend,” and “generalization.” Actors are connected to use cases via communication paths, that are represented by an association relationship. The “include” relationship provides a mechanism for factoring out common functionality that is shared among multiple use cases, and is required for the goals of the actor of the base use case to be met. The “extend” relationship provides optional functionality (optional in the sense of not being required to meet the goals), which extends the base use case at defined extension points under specified conditions. The “generalization” relationship provides a mechanism to specify variants of the base use case.

The use cases are often organized into packages with the corresponding dependencies between the use cases in the packages.

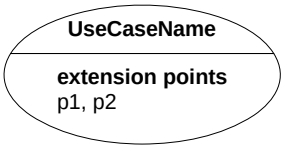
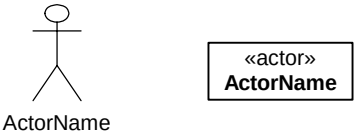
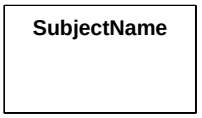
14.2 Diagram Elements

14.2.1 Use Case Diagram

Table 14.1 - Graphical nodes included in Use Case diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Use Case		**UML4SysML::UseCase**

Table 14.1 - Graphical nodes included in Use Case diagrams

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

.

Table 14.2 - Graphical paths included in Use Case diagrams

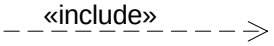
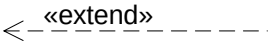
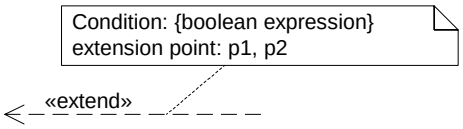
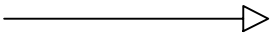
Path Type	concrete Syntax	Abstract Syntax Reference
Communication path		**UML4SysML::Association**
Include		**UML4SysML::Include**
Extend		**UML4SysML::Extend**

Table 14.2 - Graphical paths included in Use Case diagrams

Path Type	concrete Syntax	Abstract Syntax Reference
Extend with Condition		**UML4SysML::Extend**
Generalization		**UML4SysML::Kernel**

14.3 UML Extensions

None.

14.4 Usage Examples

Figure C.5 in Annex C is a top-level set of use cases for the Hybrid SUV System. Figure C.6 in Annex C shows the decomposition of the Operate the Vehicle use case. In this diagram, the frame represents the package that contains the lower level use cases. The convention of naming the package with the same name as the top level use case has been employed. This practice offers an implicit tracing mechanism that complements the explicit trace relationships in SysML.

In Figure C.6 in Annex C, the Extend relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case. Note, however, that the extended use case is defined independently of the extending use case and is meaningful independently of the extending use case. On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions. In Table 14.2, the “Start the Vehicle” use case is modeled as an extension of “Drive the Vehicle.” This means that there are conditions that may exist that require the execution of an instance of “Start the Vehicle” before an instance of “Drive the Vehicle” is executed.

The use cases “Accelerate,” “Steer,” and “Brake” are modeled using the include relationship. Include is a DirectedRelationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case. This means that “Accelerate,” “Steer,” and “Brake” are all part of the normal process of executing an instance of “Drive the Car.”

In many situations, the use of the Include and Extend relationships is subjective and may be reversed, based on the approach of an individual modeler.

Part IV - Crosscutting Constructs

This Part specifies crosscutting constructs that apply to both structure and behavior. These constructs are defined in the following clauses:

15 - Allocations defines a basic allocation relationship that can be used to allocate a set of model elements to another, such as allocating behavior to structure or allocating logical to physical components.

16 - Requirements specifies constructs for system requirements and their relationships.

17 - Profiles & Model Libraries specifies the approach to further customize and extend SysML for specific applications.

15 Allocations

15.1 Overview

Allocation is the term used by systems engineers to denote the organized cross-association (mapping) of elements within the various structures or hierarchies of a user model. The concept of “allocation” requires flexibility suitable for abstract system specification, rather than a particular constrained method of system or software design. System modelers often associate various elements in a user model in abstract, preliminary, and sometimes tentative ways. Allocations can be used early in the design as a precursor to more detailed rigorous specifications and implementations. The allocation relationship can provide an effective means for navigating the model by establishing cross relationships, and ensuring the various parts of the model are properly integrated.

This clause does not try to limit the use of the term “allocation,” but provides a basic capability to support allocation in the broadest sense. It does include some specific subclasses of allocation for allocating behavior, structure, and flows. A typical example is the allocation of activities to blocks (e.g., functions to components). This clause specifies an extension for an allocation relationship and selected subclasses of allocation, along with the notation to represent allocations in a SysML model.

15.2 Diagram Elements

The diagram elements defined in this clause may be shown on some or all SysML diagram types, in addition to the diagram elements that are specific for each diagram type.

15.2.1 Representing Allocation on Diagrams

Table 15.1 - Extension to graphical nodes included in diagrams

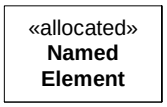
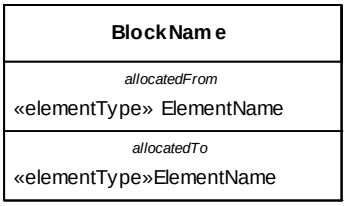
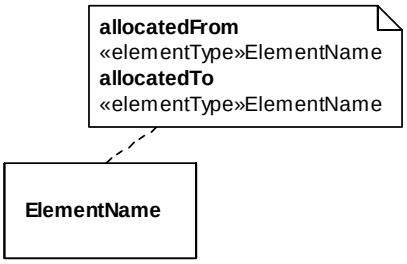
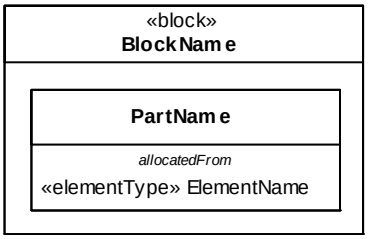
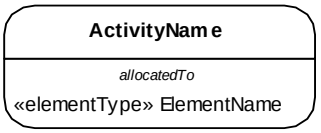
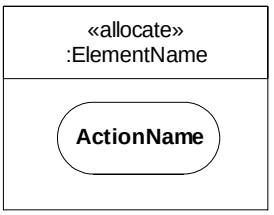
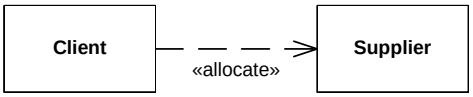
Node Name	Concrete Syntax	Abstract Syntax Reference
Allocated stereotype		**SysML::Allocation:Allocated**
Allocation derived properties displayed in compartment of a Block.		**SysML::Allocation:Allocated**
Allocation derived properties displayed in Comment.		**SysML::Allocation:Allocated**
Allocation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

Table 15.1 - Extension to graphical nodes included in diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
Allocation Activity Partition		**SysML::Allocation:Allocate ActivityPartition**
Allocation (general)		**SysML::Allocation:Allocate**

15.3 UML Extensions

15.3.1 Diagram Extensions

15.3.1.1 Tables

Allocation relationships may be depicted in tables. A separate row is provided for each «allocate» dependency. “from” is the client of the «allocate» dependency, and “to” is the supplier. Both ElementType and ElementName for client and supplier appear in this table.

15.3.1.2 Allocate Relationship Rendering

The “allocate” relationship is a dashed line with an open arrow head. The arrow points in the direction of the allocation. In other words, the directed line points “from” the element being allocated “to” the element that is the target of the allocation.

15.3.1.3 Allocated Property Compartment Format

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

15.3.1.4 Allocated Property Callout Format

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

15.3.1.5 AllocatedActivityPartition Label

For brevity, the keyword used on an AllocatedActivityPartition is «allocate», rather than the stereotype name («allocateActivityPartition»). For brevity, the «elementType» portion of the AllocatedFrom or AllocatedTo property may be elided from the diagram.

15.3.2 Stereotypes

Package Allocations

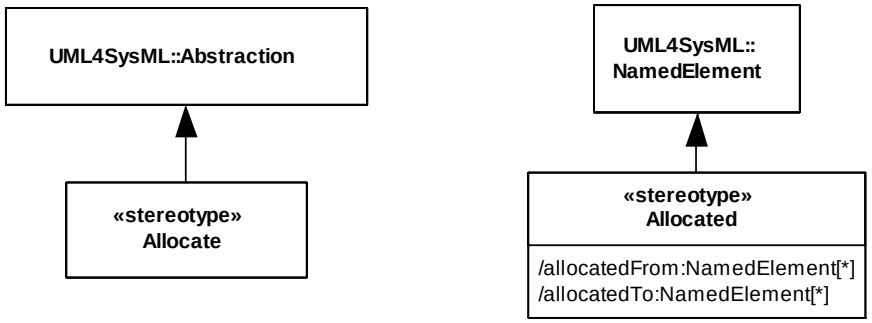


Figure 15.1 - Abstract syntax extensions for SysML Allocation

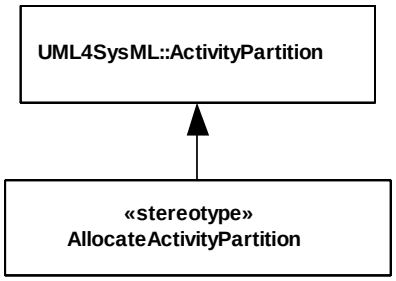


Figure 15.2 - Abstract syntax expression for AllocatedActivityPartition

15.3.2.1 Allocate(from Allocations)

Description

Allocate is a dependency based on UML::Abstraction. It is a mechanism for associating elements of different types, or in different hierarchies, at an abstract level. Allocate is used for assessing user model consistency and directing future design activity. It is expected that an «allocate» relationship between model elements is a precursor to a more concrete relationship between the elements, their properties, operations, attributes, or sub-classes.

Allocate is a stereotype of a UML4SysML::Abstraction which is permissible between any two NamedElements. It is depicted as a dependency with the “allocate” keyword attached to it.

Allocate is directional in that one NamedElement is the “from” end (no arrow), and at least one NamedElement is the “to” end (the end with the arrow).

The following paragraphs describe types of allocation that are typical in systems engineering.

Behavior allocation relates to the systems engineering concept segregating form from function. This concept requires independent models of “function” (behavior) and “form” (structure), and a separate, deliberate mapping between elements in each of these models. It is acknowledged that this concept does not support a standard object-oriented paradigm, nor is this always even desirable. Experience on large scale, complex systems engineering problems have proven, however, that segregation of form and function is a valuable approach. In addition, behavior allocation may also include the allocation of Behaviors to BehavioralFeatures of Blocks (e.g., Operations).

Flow allocation specifically maps flows in functional system representations to flows in structural system representations.

Flow between activities can either be control or object flow. The figures in the Usage Examples show concrete syntax for how object flow is mapped to connectors on Activity Diagrams. Allocation of control flow is not specifically addressed in SysML, but may be represented by relating an ItemFlow to the Control Flow using the UML relationship InformationFlow.realizingActivityEdge.

Note that allocation of ObjectFlow to Connector is an Allocation of Usage, and does NOT imply any relation between any defining Blocks of ObjectFlows and any defining associations of connectors.

The figures in the Usage Examples illustrate an available mechanism for relating the objectNode from an activity diagram to the itemFlow on an internal block diagram. ItemFlow is discussed in Clause 9, “Ports and Flows.”

Pin to Port allocation is not addressed in this release of SysML.

Structure allocation is associated with the concept of separate “logical” and “physical” representations of a system. It is often necessary to construct separate depictions of a system and define mappings between them. For example, a complete system hierarchy may be built and maintained at an abstract level. In turn, it must then be mapped to another complete assembly hierarchy at a more concrete level. The set of models supporting complex systems development may include many of these levels of abstraction. This specification will not define “logical” or “physical” in this context, except to acknowledge the stated need to capture allocation relationships between separate system representations.

Constraints

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

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

15.3.2.2 Allocated(from Allocations)

Description

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

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

Attributes

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

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

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

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

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

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

15.3.2.3 AllocateActivityPartition(from Allocations)

Description

AllocateActivityPartition is used to depict an «allocate» relationship on an Activity diagram. The AllocateActivityPartition is a standard UML2::ActivityPartition, with modified constraints as stated below.

Constraints

An Action appearing in an “AllocateActivityPartition” will be the /client (from) end of an “allocate” dependency. The element that represents the “AllocateActivityPartition” will be the /supplier (to) end of the same “allocate” dependency. In the «AllocateActivityPartition» name field, Properties are designated by the use of a fully qualified name (including colon, e.g., “part_name:Block_Name”), and Classifiers are designated by a simple name (no colons, e.g., “Block_Name”).

The «AllocateActivityPartition» maintains the constraints, but not the semantics, of the UML2::ActivityPartition. Classifiers or Properties represented by an «AllocateActivityPartition» do not have any direct responsibility for invoking behavior depicted within the partition boundaries. To depict this kind of direct responsibility, the modeler is directed to the UML 2 Superstructure specification, sub clause 12.3.10, “ActivityPartition,” Semantics topic.

15.4 Usage Examples

The following examples depict allocation relationships as property callout boxes (basic), property compartment of a Block (basic), and property compartments of Activities and Parts (advanced). Figure 15.3 shows generic allocation for Blocks.

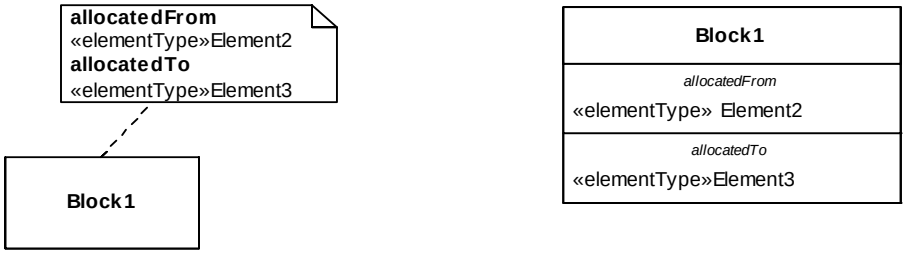


Figure 15.3 - Generic Allocation, including /from and /to association ends

15.4.1 Behavior Allocation of Actions to Parts and Activities to Blocks

Specific behavior allocation of Actions to Parts are depicted in Figure 15.4. Note that the AllocateActivityPartition, if used in this manner, is unambiguously associated with behavior allocation.

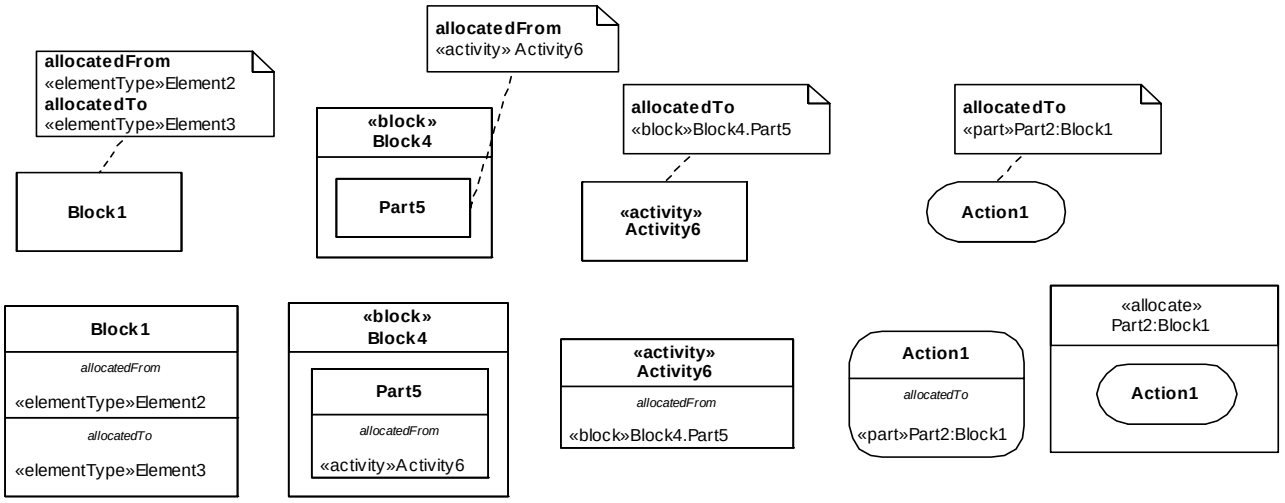


Figure 15.4 - Behavior allocation

15.4.2 Allocate Flow

Figure 15.5 shows flow allocation of ObjectFlow to a Connector, or alternatively to an ItemFlow. Allocation of ControlFlow is not shown as an example, but it is not prohibited in SysML.

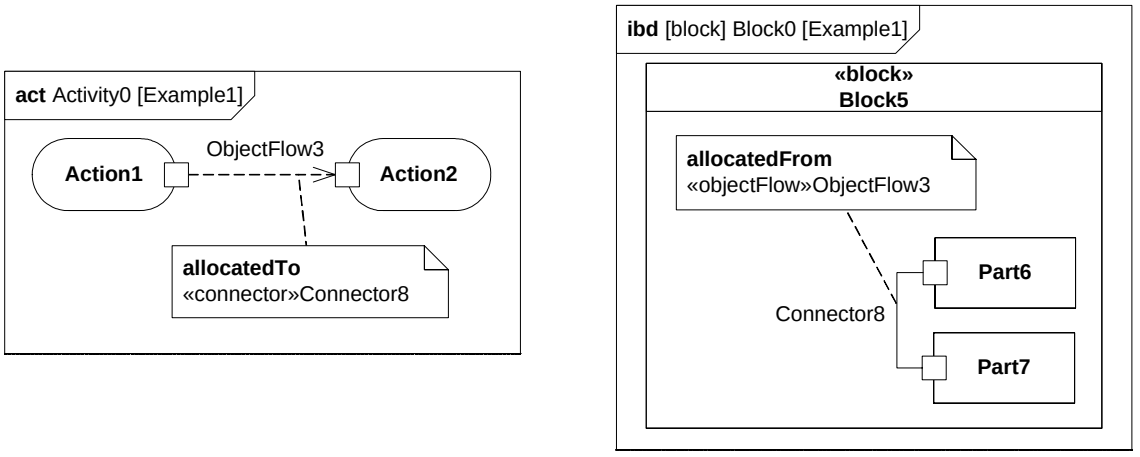


Figure 15.5 - Example of flow allocation from ObjectFlow to Connector

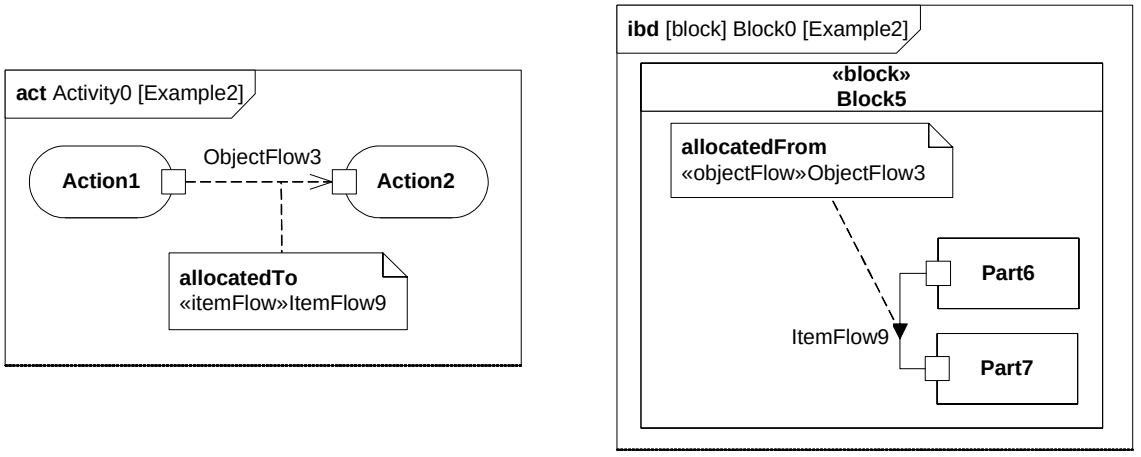


Figure 15.6 - Example of flow allocation from ObjectFlow to ItemFlow

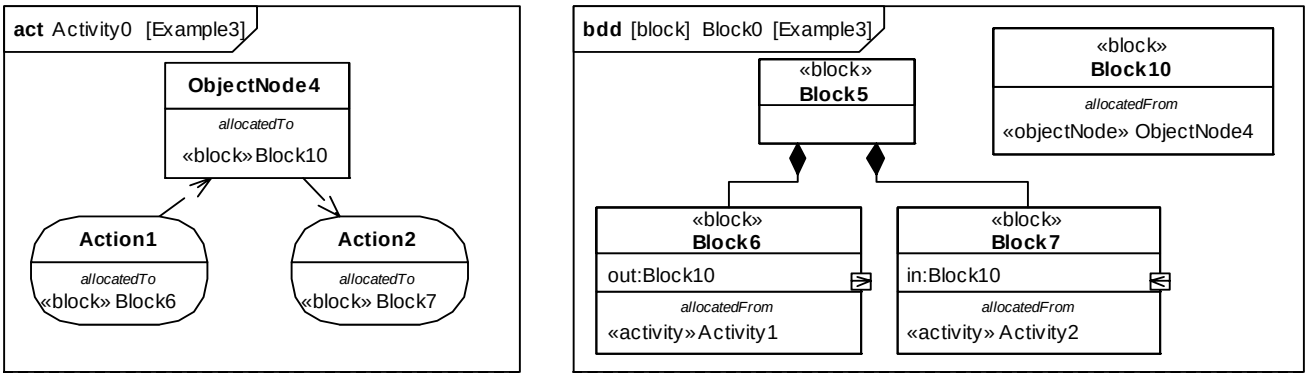


Figure 15.7 - Example of flow allocation from ObjectNode to FlowProperty

15.4.2.1 Allocating Structure

Systems engineers have a frequent need to allocate structural model elements (e.g., blocks, parts, or connectors) to other structural elements. For example, if a particular user model includes an abstract logical structure, it may be important to show how these model elements are allocated to a more concrete physical structure. The need also arises, when adding detail to a structural model, to allocate a connector (at a more abstract level) to a part (at a more concrete level).

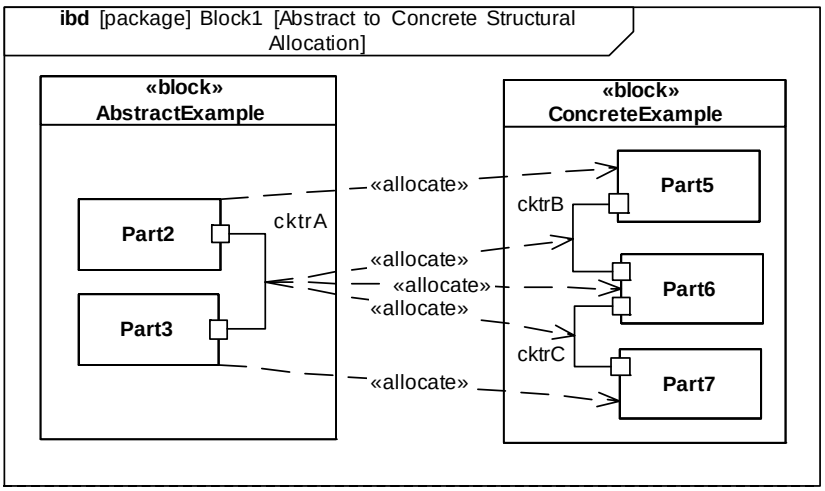


Figure 15.8 - Example of Structural Allocation

15.4.2.2 Automotive Example

Example: consider the functions required to portion and deliver power for a hybrid SUV. The activities for providing power are allocated to blocks within the Hybrid SUV, as shown in Figure C.35 in Annex C.

Figure C.36 in Annex C shows an internal block diagram showing allocation for the HybridSUV Accelerate example.

15.4.3 Tabular Representation

The table shown in Figure C.37 in Annex C is provided as a specific example of how the «allocate» dependency may be depicted in tabular form, consistent with the automotive example above.

The allocation table can also be shown using a sparse matrix style as in the following example shown in Figure 15.9.

:

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

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

Figure 15.9 - Allocation Matrix Showing Allocation for Hybrid SUV Accelerate Example

16 Requirements

16.1 Overview

A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition a system must achieve. SysML provides modeling constructs to represent text-based requirements and relate them to other modeling elements. The requirements diagram described in this clause can depict the requirements in graphical, tabular, or tree structure format. A requirement can also appear on other diagrams to show its relationship to other modeling elements. The requirements modeling constructs are intended to provide a bridge between traditional requirements management tools and the other SysML models.

A requirement is defined as a stereotype of UML Class subject to a set of constraints. A standard requirement includes properties to specify its unique identifier and text requirement. Additional properties such as verification status, can be specified by the user.

Several requirements relationships are specified that enable the modeler to relate requirements to other requirements as well as to other model elements. These include relationships for defining a requirements hierarchy, deriving requirements, satisfying requirements, verifying requirements, and refining requirements.

A composite requirement can contain subrequirements in terms of a requirements hierarchy, specified using the UML namespace containment mechanism. This relationship enables a complex requirement to be decomposed into its containing child requirements. A composite requirement may state that the system shall do A and B and C, which can be decomposed into the child requirements that the system shall do A, the system shall do B, and the system shall do C. An entire specification can be decomposed into children requirements, which can be further decomposed into their children to define the requirements hierarchy.

There is a real need for requirement reuse across product families and projects. Typical scenarios are regulatory, statutory, or contractual requirements that are applicable across products and/or projects and requirements that are reused across product families (versions/variants). In these cases, one would like to be able to reference a requirement, or requirement set in multiple contexts with updates to the original requirements propagated to the reused requirement(s).

The use of namespace containment to specify requirements hierarchies precludes reusing requirements in different contexts since a given model element can only exist in one namespace. Since the concept of requirements reuse is very important in many applications, SysML introduces the concept of a slave requirement. A slave requirement is a requirement whose text property is a read-only copy of the text property of a master requirement. The text property of the slave requirement is constrained to be the same as the text property of the related master requirement. The master/slave relationship is indicated by the use of the copy relationship.

The “derive requirement” relationship relates a derived requirement to its source requirement. This typically involves analysis to determine the multiple derived requirements that support a source requirement. The derived requirements generally correspond to requirements at the next level of the system hierarchy. A simple example may be a vehicle acceleration requirement that is analyzed to derive requirements for engine power, vehicle weight, and body drag.

The satisfy relationship describes how a design or implementation model satisfies one or more requirements. A system modeler specifies the system design elements that are intended to satisfy the requirement. In the example above, the engine design satisfies the engine power requirement.

The verify relationship defines how a test case or other model element verifies a requirement. In SysML, a test case or other named element can be used as a general mechanism to represent any of the standard verification methods for inspection, analysis, demonstration, or test. Additional subclasses can be defined by the user if required to represent the different verification methods. A verdict property of a test case can be used to represent the verification result. The SysML test case is defined consistent with the UML testing profile to facilitate integration between the two profiles.

The refine requirement relationship can be used to describe how a model element or set of elements can be used to further refine a requirement. For example, a use case or activity diagram may be used to refine a text-based functional requirement. Alternatively, it may be used to show how a text-based requirement refines a model element. In this case, some elaborated text could be used to refine a less fine-grained model element.

A generic trace requirement relationship provides a general-purpose relationship between a requirement and any other model element. The semantics of trace include no real constraints and therefore are quite weak. As a result, it is recommended that the trace relationship not be used in conjunction with the other requirements relationships described above.

The rationale construct that is defined in Clause 7, “Model Elements” is quite useful in support of requirements. It enables the modeler to attach a rationale to any requirements relationship or to the requirement itself. For example, a rationale can be attached to a satisfy relationship that refers to an analysis report or trade study that provides the supporting rationale for why the particular design satisfies the requirement. Similarly, this can be used with the other relationships such as the derive relationship. It also provides an alternative mechanism to capture the verify relationship by attaching a rationale to a satisfy relationship that references a test case.

Modelers can customize requirements taxonomies by defining additional subclasses of the Requirement stereotype. For example, a modeler may want to define requirements categories to represent operational, functional, interface, performance, physical, storage, activation/deactivation, design constraints, and other specialized requirements such as reliability and maintainability, or to represent a high level stakeholder need. The stereotype enables the modeler to add constraints that restrict the types of model elements that may be assigned to satisfy the requirement. For example, a functional requirement may be constrained so that it can only be satisfied by a SysML behavior such as an activity, state machine, or interaction. Some potential Requirement subclasses are defined in Non-normative Extensions.

16.2 Diagram Elements

16.2.1 Requirement Diagram

Table 16.1 - Graphical nodes included in Requirement diagrams

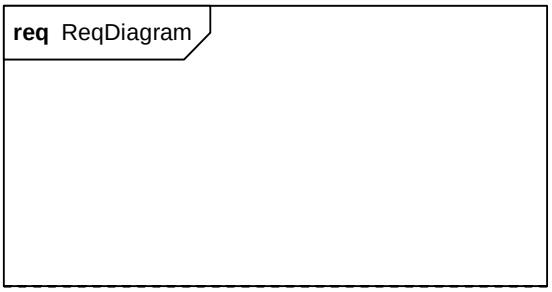
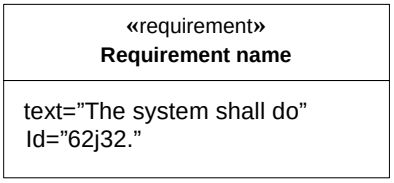
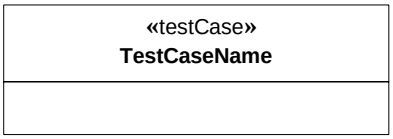
Node Name	Concrete Syntax	Abstract Syntax Reference
Requirement Diagram		**SysML::Requirements:: Requirement, SysML:: ModelElements::Package**
Requirement		**SysML::Requirements:: Requirement**
TestCase		**SysML::Requirements:: TestCase**

Table 16.2 - Graphical paths included in Requirement diagrams

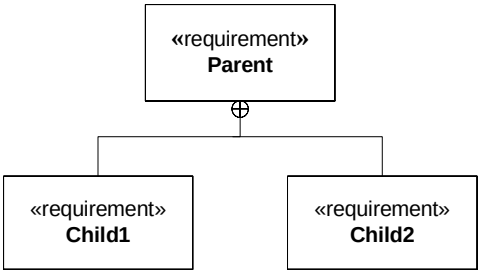
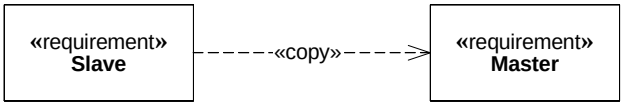
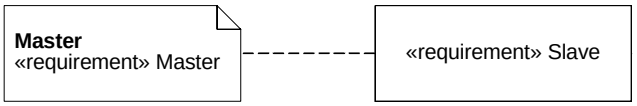
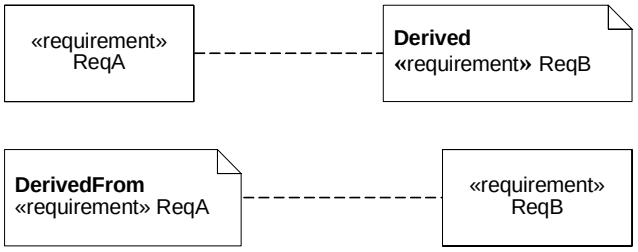
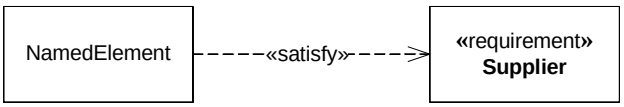
Path Type	Concrete Syntax	Abstract Syntax Reference
Requirement containment relationship		**UML4SysML:: NestedClassifier**
CopyDependency		**SysML::Requirements:: Copy**
MasterCallout		**SysML::Requirements:: Copy**
Derive Dependency		**SysML::Requirements:: DeriveReqt**
DeriveCallout		**SysML::Requirements::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

Table 16.2 - Graphical paths included in Requirement diagrams

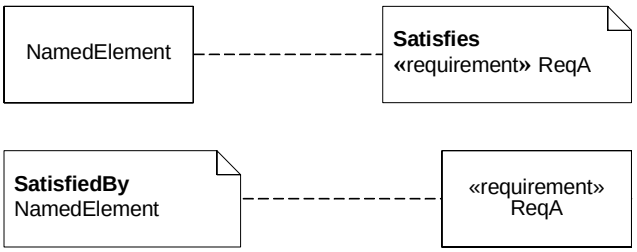
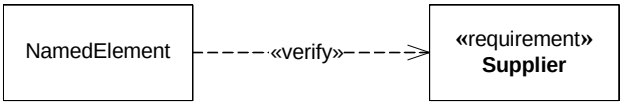
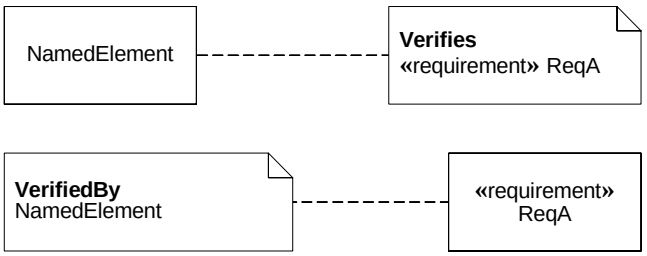
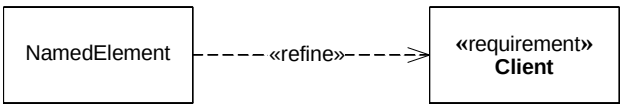
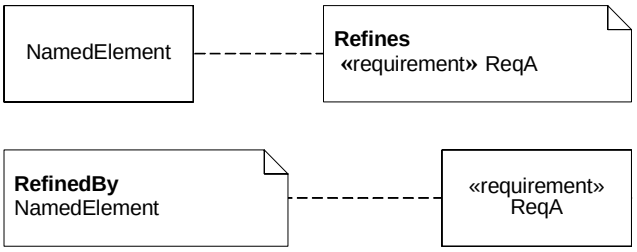
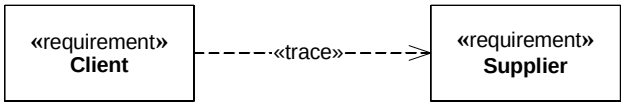
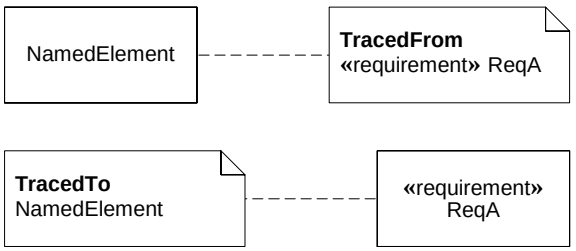
Path Type	Concrete Syntax	Abstract Syntax Reference
SatisfyCallout		**SysML::Requirements::Satisfy**
Verify Dependency		**SysML::Requirements::Verify**
VerifyCallout		**SysML::Requirements::Verify**
Refine Dependency		**UML4SysML::Refine**
RefineCallout		**UML4SysML::Refine**

Table 16.2 - Graphical paths included in Requirement diagrams

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

16.3 UML Extensions

16.3.1 Diagram Extensions

16.3.1.1 Requirement Diagram

The Requirement Diagram can only display requirements, packages, other classifiers, test cases, and rationale. The relationships for containment, deriveReqt, satisfy, verify, refine, copy, and trace can be shown on a requirement diagram. The callout notation can also be used to reflect the relationship of other model elements to a requirement.

16.3.1.2 Requirement Notation

The requirement is represented as shown in Table 16-1. The «requirement» compartment label for the stereotype properties compartment (e.g., id and text) can be elided.

16.3.1.3 Requirement Property Callout Format

A callout notation can be used to represent derive, satisfy, verify, refine, copy, and trace relationships as indicated in Table 16.2. For brevity, the «elementType» may be elided.

16.3.1.4 Requirements on Other Diagrams

Requirements can also be represented on other diagrams to show their relationship to other model elements. The compartment and callout notation described in 16.3.1.2 and 16.3.1.3 can be used. The callouts represent the requirement that is attached to another model element such as a design element.

16.3.1.5 Requirements Table

The tabular format is used to represent the requirements, their properties and relationships, and may include:

- Requirements with their properties in columns.
- A column that includes the supplier for any of the dependency relationships (Derive, Verify, Refine, Trace).
- A column that includes the model elements that satisfy the requirement.
- A column that represents the rationale for any of the above relationships, including reference to analysis reports for trace rationale, trade studies for design rationale, or test procedures for verification rationale.

The relationships between requirements and other objects can also be shown using a sparse matrix style that is similar to the table used for allocations (sub clause 15.4.3, “Tabular Representation”). The table should include the source and target elements names (and optionally kinds) and the requirement dependency kind.

table [requirement] Performance [Decomposition of Performance Requirement]

id	name	text
2	Performance	The Hybrid SUV shall have the braking, acceleration, and off-road capability of a typical SUV, but have dramatically better fuel economy.
2.1	Braking	The Hybrid SUV shall have the braking capability of a typical SUV.
2.2	FuelEconomy	The Hybrid SUV shall have dramatically better fuel economy than a typical SUV.
2.3	OffRoadCapability	The Hybrid SUV shall have the off-road capability of a typical SUV.
2.4	Acceleration	The Hybrid SUV shall have the acceleration of a typical SUV.

table [requirement] Performance [Tree of Performance Requirements]

id	name	relation	id	name	relation	id	name
2.1	Braking	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.2	Range			
4.2	FuelCapacity	deriveReqt	d.2	Range			
2.3	OffRoadCapability	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
2.4	Acceleration	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
4.1	CargoCapacity	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement

16.3.2 Stereotypes

Package Requirements

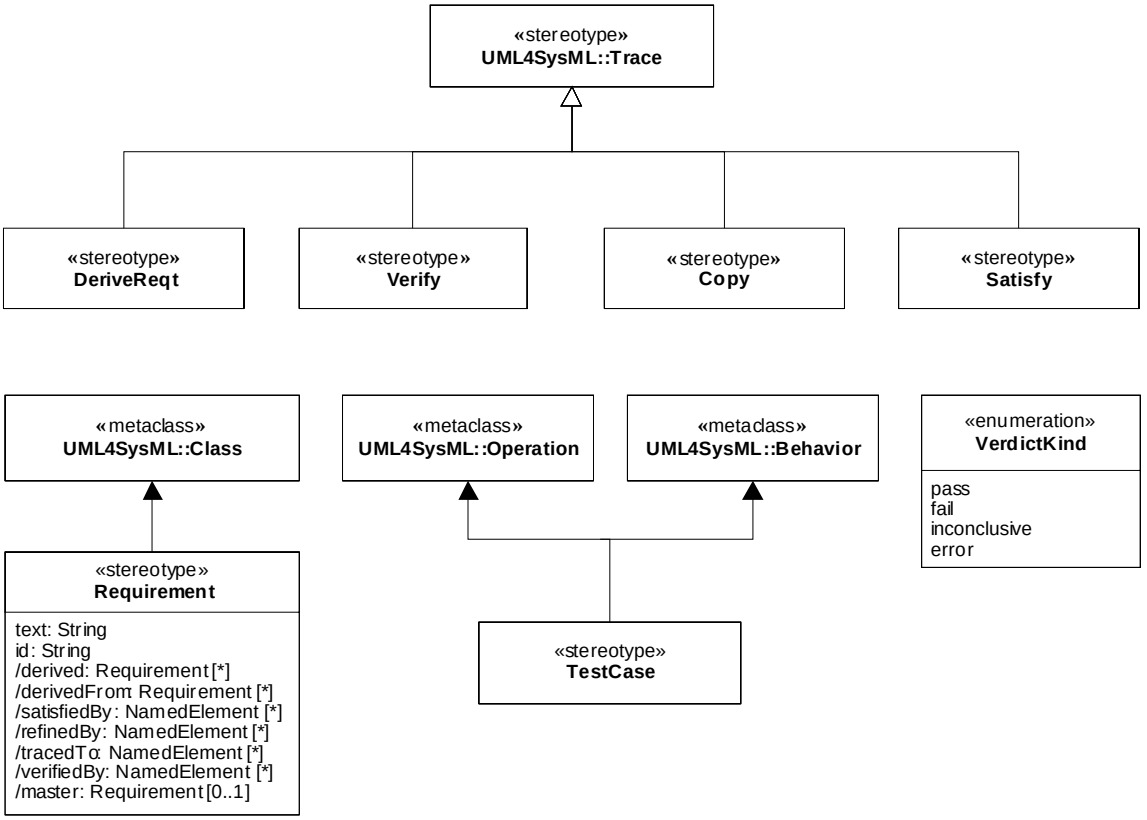


Figure 16.1 - Abstract Syntax for Requirements Stereotypes

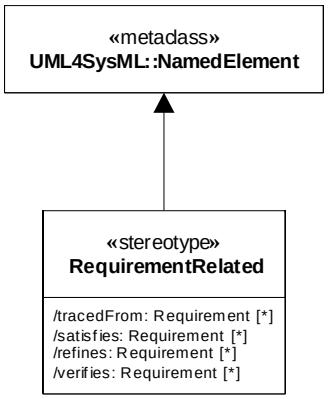


Figure 16.2 - Abstract Syntax for Requirements Stereotypes (cont)

16.3.2.1 Copy

Description

A Copy relationship is a dependency between a supplier requirement and a client requirement that specifies that the text of the client requirement is a read-only copy of the text of the supplier requirement.

A Copy dependency created between two requirements maintains a master/slave relationship between the two elements for the purpose of requirements re-use in different contexts. When a Copy dependency exists between two requirements, the requirement text of the client requirement is a read-only copy of the requirement text of the requirement at the supplier end of the dependency.

Constraints

[1] A Copy dependency may only be created between two classes that have the “requirement” stereotype, or a subtype of the “requirement” stereotype applied.

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

[3] Constraint [2] is applied recursively to all subrequirements.

16.3.2.2 DeriveReqt

Description

A DeriveReqt relationship is a dependency between two requirements in which a client requirement can be derived from the supplier requirement. For example, a system requirement may be derived from a business need, or lower-level requirements may be derived from a system requirement. As with other dependencies, the arrow direction points from the derived (client) requirement to the (supplier) requirement from which it is derived.

Constraints

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

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

16.3.2.3 Requirement

Description

A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition that a system must satisfy. Requirements are used to establish a contract between the customer (or other stakeholder) and those responsible for designing and implementing the system.

A requirement is a stereotype of Class. Compound requirements can be created by using the nesting capability of the class definition mechanism. The default interpretation of a compound requirement, unless stated differently by the compound requirement itself, is that all its subrequirements must be satisfied for the compound requirement to be satisfied. Subrequirements can be accessed through the “nestedClassifier” property of a class. When a requirement has nested requirements, all the nested requirements apply as part of the container requirement. Deleting the container requirement deleted the nested requirements, a functionality inherited from UML.

Attributes

- text: String
  The textual representation or a reference to the textual representation of the requirement.

- id: String
  The unique id of the requirement.
- /satisfiedBy: NamedElement [*]
  Derived from all elements that are the client of a «satisfy» relationship for which this requirement is a supplier.
- /verifiedBy: NamedElement [*]
  Derived from all elements that are the client of a «verify» relationship for which this requirement is a supplier.
- /tracedTo: NamedElement [*]
  Derived from all elements that are the supplier of a «trace» relationship for which this requirement is a client.
- /derived: Requirement [*]
  Derived from all requirements that are the client of a «deriveReqt» relationship for which this requirement is a supplier.
- /derivedFrom: Requirement [*]
  Derived from all requirements that are the supplier of a «deriveReqt» relationship for which this requirement is a client.
- /refinedBy: NamedElement [*]
  Derived from all elements that are the client of a «refine» relationship for which this requirement is a supplier.
- /master: Requirement [0..1
  This is a derived property that lists the master requirement for this slave requirement. The master attribute is derived from the supplier of the Copy dependency that has this requirement as the slave.

Constraints

[1] The property “ownedOperation” must be empty.

[2] The property “ownedAttribute” must be empty.

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

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

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

16.3.2.4 RequirementRelated

Description

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

Attributes

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

16.3.2.5 TestCase

Description

A test case is a method for verifying a requirement is satisfied.

Constraints

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

16.3.2.6 Satisfy

Description

A Satisfy relationship is a dependency between a requirement and a model element that fulfills the requirement. As with other dependencies, the arrow direction points from the satisfying (client) model element to the (supplier) requirement that is satisfied.

Constraints

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

16.3.2.7 Verify

Description

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

Constraints

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

16.4 Usage Examples

All the examples in this clause are based on a set of publicly available (on-line) requirement specifications from the *National Highway Traffic Safety Administration (NHTSA*.) Excerpts of the original requirement text used to create the models are shown in Figure 16.3. The name and ID of these requirements are referred to in the SysML usage examples that follow. See *NHTSA specification 49CFR571.135* for the complete text from which these examples are taken.

16.4.1 Requirement Decomposition and Traceability

The diagram in Figure 16.3 shows an example of a compound requirement decomposed into multiple subrequirements.

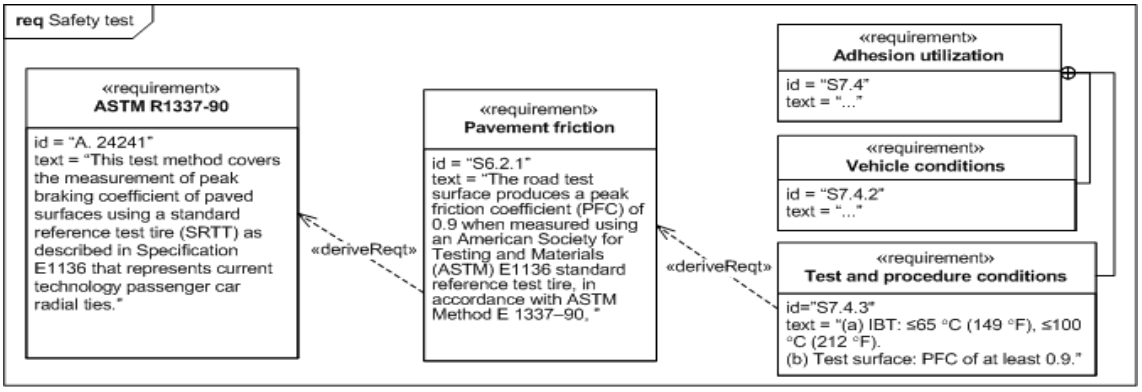


Figure 16.3 - Requirements Derivation

16.4.2 Requirements and Design Elements

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

.

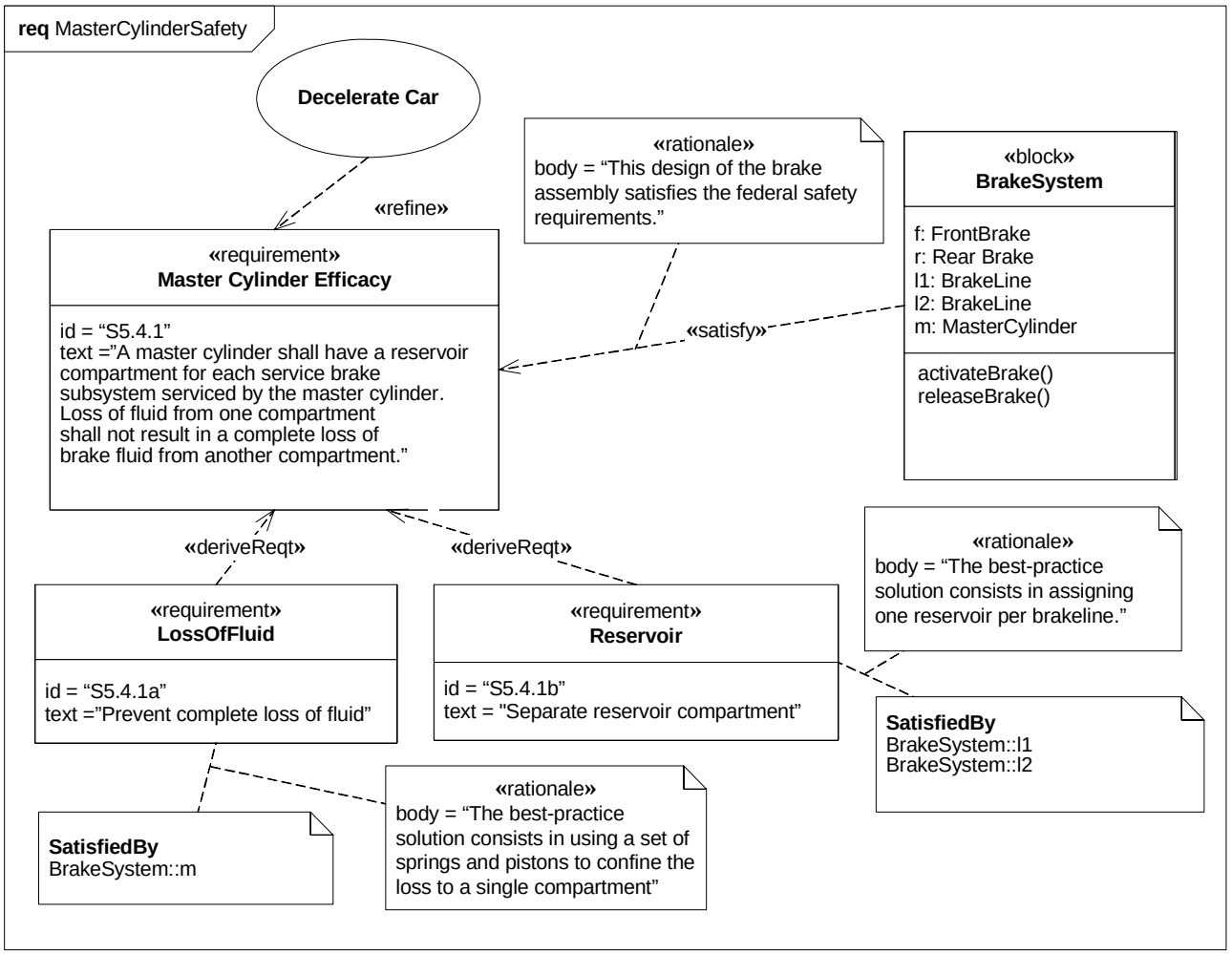


Figure 16.4 - Links between requirements and design

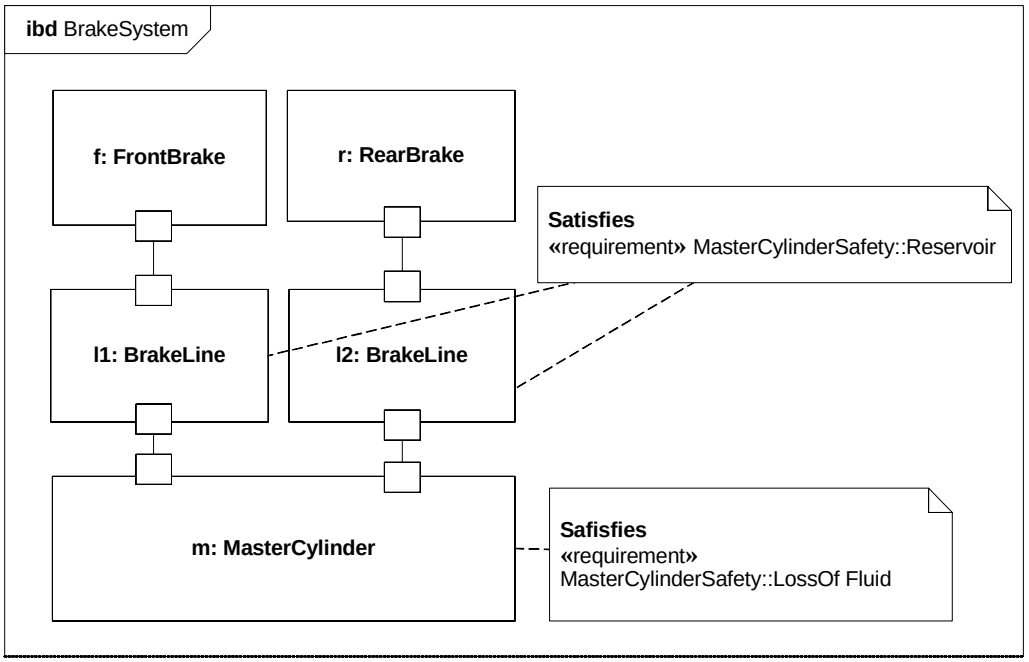


Figure 16.5 - Requirement satisfaction in an internal block diagram.

16.4.3 Requirements Reuse

Figure 16.6 illustrates the use of the Copy dependency to allow a single requirement to be reused in several requirements hierarchies. The master tag provides a textual reference to the reused requirement.

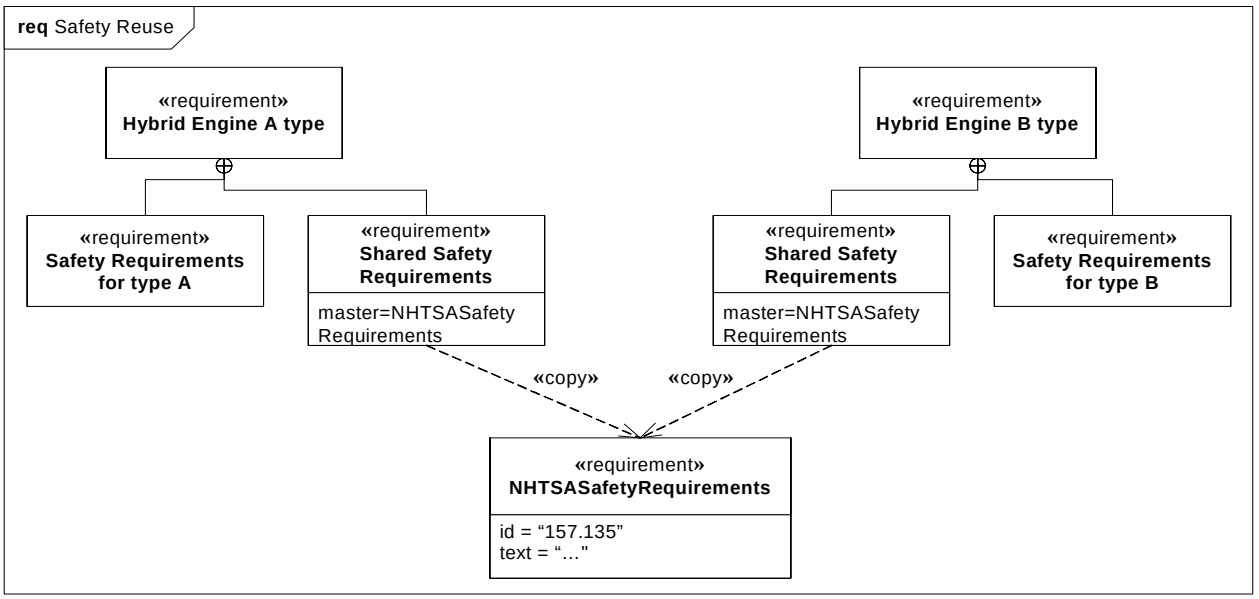


Figure 16.6 - Use of the copy dependency to facilitate reuse

16.4.4 Verification Procedure (Test Case)

The example in Figure 16.7 shows how a complex test case, in this example a test for a passenger-car brake system given as a series of steps in text form, can be described using another type of diagram representation. Figure 16.8 shows a Test Case linked back to a requirement in Figure 16.7 using a Verify callout. Note that the modeling of test cases can also be addressed using the UML Testing Profile, available from the Object Management Group.

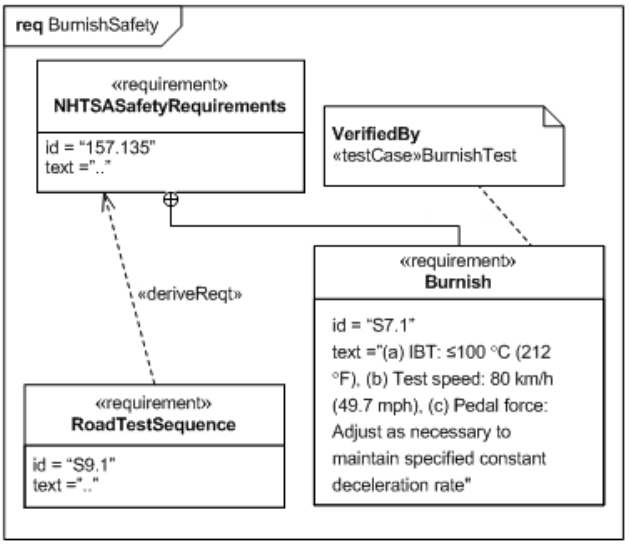


Figure 16.7 - Linkage of a Test Case to a requirement: This figure shows the Requirement Diagram

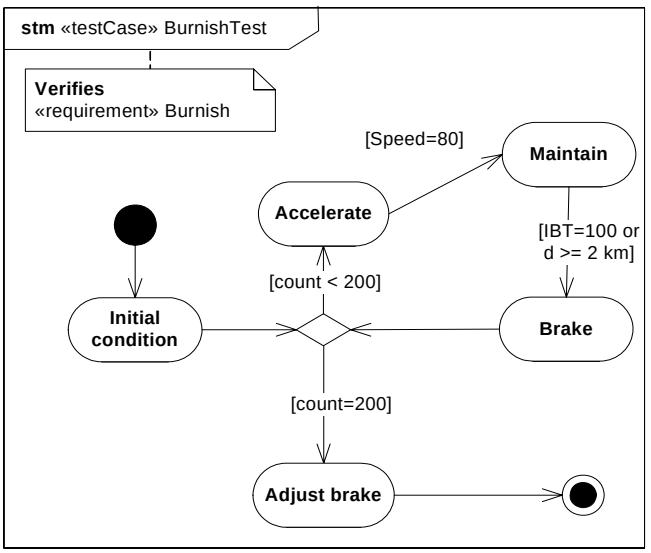


Figure 16.8 - Linkage of a Test Case to a requirement: This figure shows the Test Case as a State Diagram

17 Profiles & Model Libraries

17.1 Overview

The Profiles package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different domains. The profiles mechanism is consistent with the OMG Meta Object Facility (MOF). SysML has added some notational extensions to represent stereotype properties in compartments as well as notes.

The stereotype is the primary mechanism used to create profiles to extend the metamodel. Stereotypes are defined by extending a metaclass, and then have them applied to the applicable model elements in the user model. A stereotype of a requirement could be extended to create a «functionalRequirement» as described in Non-normative Extensions. This would allow specific properties and constraints to be created for a functional requirement. For example, a functional requirement may be constrained such that it must be satisfied by an operation or behavior. When the stereotype is applied to a requirement, then the requirement would include the notation «functionalRequirement» in addition to the name of the particular functional requirement. Extending the metaclass requirement is different from creating a subclass of requirement called functionalRequirement.

The Usage Examples sub clause provides guidance both on how to use existing profiles and how to create new profiles. In addition, the examples provide guidance on the use of model libraries. A model library is a library of model elements including class and other type definitions that are considered reusable for a given domain. These guidelines can be applied to further customize SysML for domain specific applications such as automotive, military, or space systems.

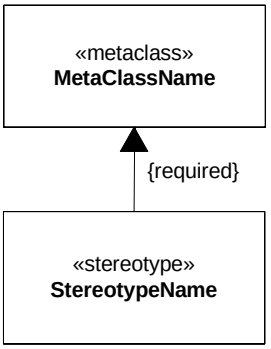
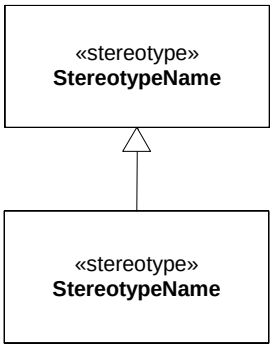
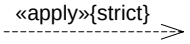
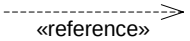
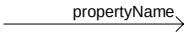
17.2 Diagram Elements

17.2.1 Profile Definition in Package Diagram

Table 17.1 - Graphical nodes used in profile definition

Node Name	Concrete Syntax	Abstract Syntax Reference
Stereotype	«stereotype» **StereotypeName**	**UML4SysML::Stereotype**
Metaclass	«metaclass» **MetaClassName**	**UML4SysML::Class**
Profile	«profile» **ProfileName**	**UML4SysML::Profile**
Model Library	«modelLibrary» **LibraryName**	**UML::StandardProfileL2**

Table 17.2 - Graphical paths used in profile definition

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

NOTE: In the above table, boolean properties can be displayed alternatively as BooleanPropertyName=[True|False].

17.2.1.1 Extension

In Figure 17.1, a simple stereotype Clock is defined to be applicable at will (dynamically) to instances of the metaclass Class and describes a clock software component for an embedded software system. It has description of the operating system version supported, an indication of whether it is compliant to the POSIX operating system standard and a reference to the operation that starts the clock.

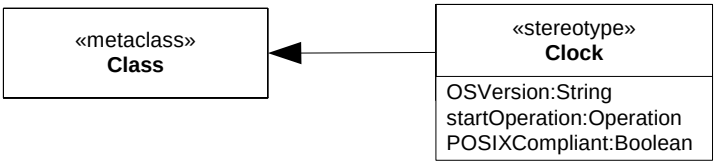


Figure 17.1 - Defining a stereotype

17.2.2 Stereotypes Used On Diagrams

Table 17.3 - Notations for Stereotype Use

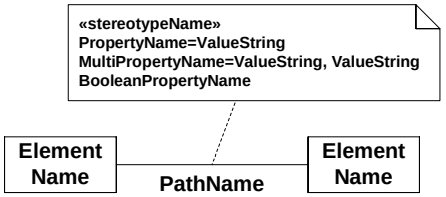
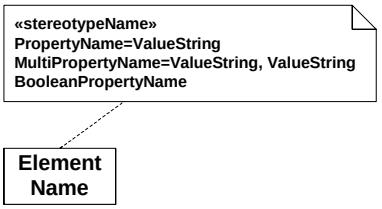
StereotypeNote		**UML4SysML::Element**
StereotypeNote		**UML4SysML::Element**

Table 17.4 - Notations for Stereotype Use (continued)

Node Name	Concrete Syntax	Abstract Syntax Reference
StereotypeInNode	«stereotypeName» {PropertyName=ValueString; BooleanPropertyName} **NodeName**	**UML4SysML::Element**
StereotypeInCompartment Element	**NodeName** «stereotypeName»{PropertyName=ValueString}ElementName «stereotypeName»{PropertyName=ValueString; BooleanPropertyName} ElementName	**UML4SysML::Element**
StereotypeOnEdge	**Element Name** **«stereotypeName»** **{PropertyName=ValueString;** **BooleanPropertyName}PathName** **Element Name**	**UML4SysML::Element**
Stereotype Compartment	**«stereotypeName»** **NodeName** «stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName	**UML4SysML::Element**

17.2.2.1 StereotypeInNode

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

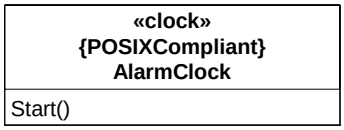


Figure 17.2 - Using a stereotype

17.2.2.2 StereotypeInComment

When, two stereotypes, Clock and Creator, are applied to the same model element, as is shown in Figure 17.3, the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

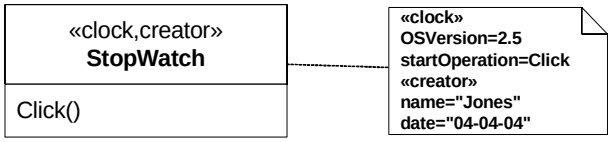


Figure 17.3 - Using stereotypes and showing values

17.2.2.3 StereotypeInCompartment

Finally, the compartment form is shown.

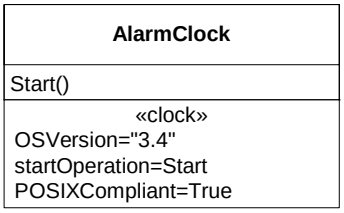


Figure 17.4 - Other notational forms for showing values

In this case, AlarmClock is valid for OS version 3.4, is POSIX-compliant and has a starting operation called Start. Note that multiple stereotypes can be shown using multiple compartments.

17.3 UML Extensions

None.

17.4 Usage Examples

17.4.1 Defining a Profile

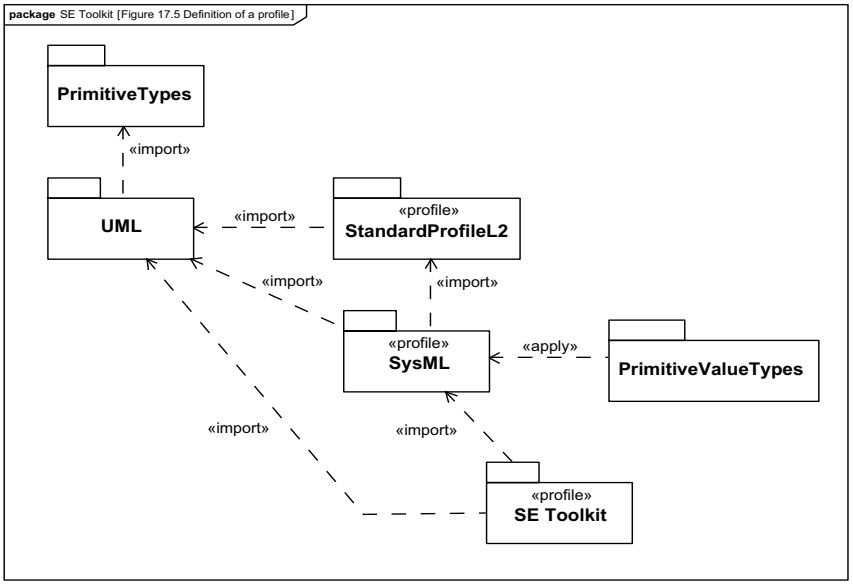


Figure 17.5 - Definition of a profile

In this example, the modeler has created a new profile called SE Toolkit, which imports the SysML profile, so that it can build upon the stereotypes it contains. The set of metaclasses available to users of the SysML profile is identified by a reference to a metamodel, in this case a subset of UML specific to SysML. The SE Toolkit can extend those metaclasses from UML that the SysML profile references.

17.4.2 Adding Stereotypes to a Profile

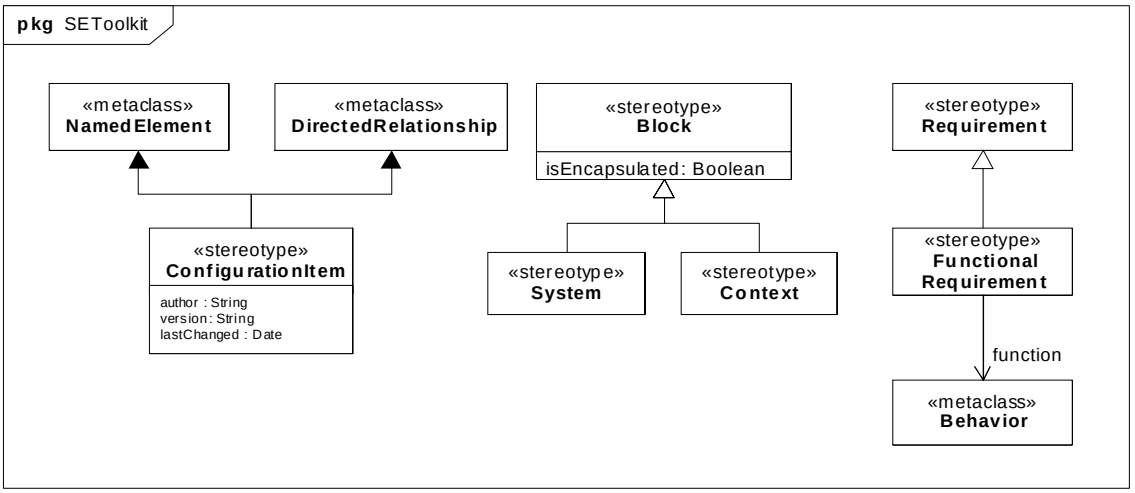


Figure 17.6 - Profile Contents

In SE Toolkit, both the mechanisms for adding new stereotypes are used. The first, exemplified by configurationItem, is called an extension, shown by a line with a filled triangle; this relates a stereotype to a reference (called base) class or classes, in this case NamedElement and DirectedRelationship from UML and adds new properties that every NamedElement or DirectedRelationship stereotyped by configurationItem must have. NamedElement and DirectedRelationship are abstract classes in UML so it is their subclasses that can have the stereotype applied. The second mechanism is demonstrated by the system and context stereotypes which are sub-stereotypes of an existing SysML stereotype, Block; sub-stereotypes inherit any properties of their super-stereotype and also extend the same base class or classes. Note that TypedElements whose type is extended by «system» do not display the «system» stereotype; this also applies to InstanceSpecifications. Any notational conventions of this have to be explicitly specified in a diagram extension.

There is also an example of how stereotypes (in this case FunctionalRequirement) can have unidirectional associations to metaclasses in the reference metamodel (in this case Behavior).

17.4.3 Defining a Model Library that Uses a Profile

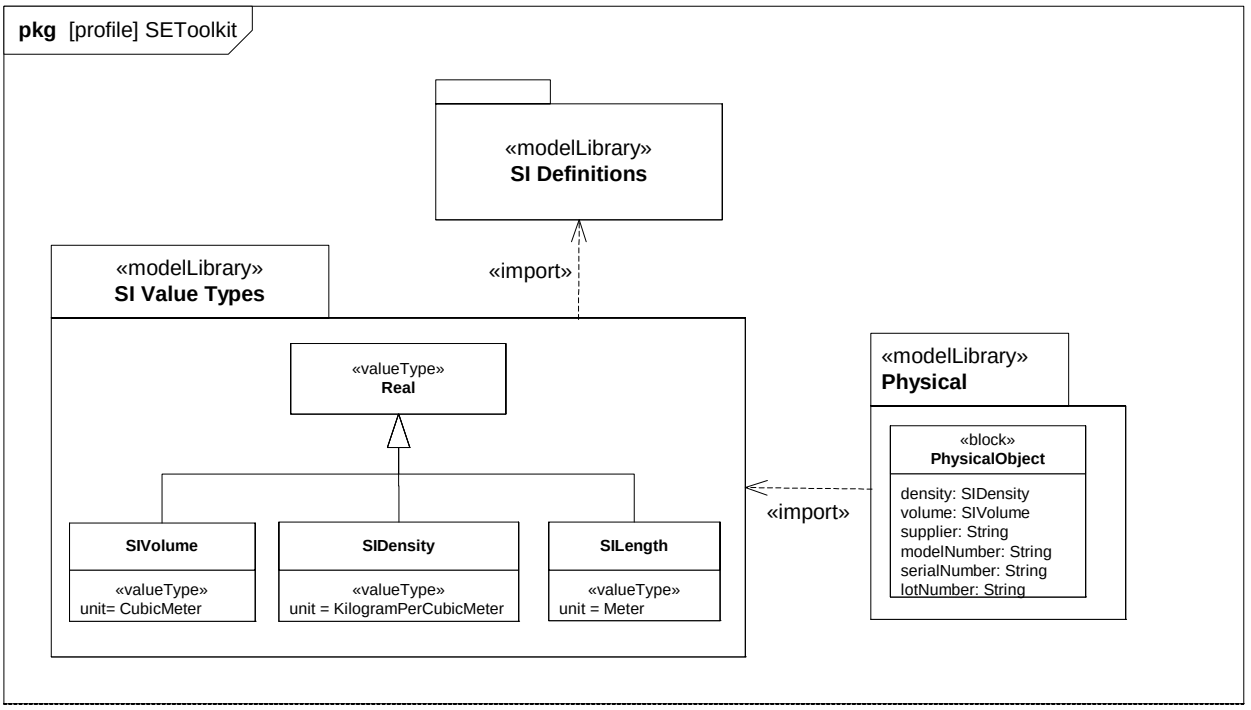


Figure 17.7 - Two model libraries

The model library SI Value Types imports a model library called SI Definitions, so it can use model elements from them in its own definition. It defines value types having specific units which can be used when property values are measured in SI units. SI Definitions is a separately published model library, containing definitions of standard SI units and quantity kinds such as shown in Annex D, sub clause D.4. A further model library, Physical, imports SI Value Types so it can define properties that have those types. One model element, PhysicalObject, is shown, a block that can be used as a supertype for a physical object.

17.4.4 Guidance on Whether to Use a Stereotype or Class

This sub clause provides guidance on when to use stereotypes. Stereotypes can be applied to any model element. Stereotyping a model element allows the model element to be identified with the «guillemet» notation. In addition, the stereotyped model element can have stereotype properties, and the stereotype can specify constraints on the model element.

The modeler must decide when to create a stereotype of a class versus when to specialize (subclass) the class. One reason is to be able to identify the class with the «guillemet» notation. In addition, the stereotype properties are different from properties of classes. Stereotype properties represent properties of the class that are not instantiated and therefore do not have a unique value for each instance of the class, although a class thus stereotyped can have a separate value for the property.

SE Toolkit::functionalRequirement, which extends Class through its superstereotype, Requirement, is an example where a stereotype is appropriate because every modeling element stereotyped by SE Toolkit::functionalRequirement has a reference to another modeling element. In another example, SE Toolkit::configurationItem defined above, which applies to classes among other concepts, is a stereotype because its properties characterize the author, version, and last changed date of the modeling element themselves. One test of this is whether the new properties are inheritable; in this case author, version, and last-changed date are not, because it is only those classes under configuration control that need the properties. To summarize, in the following circumstances a stereotype is appropriate:

- Where the model concept to be extended is not a class or class-based.
- Where the extensions include properties that reference other model elements.
- Where the extensions include properties that describe modeling data, not system data.

An example where a class is more appropriate is PhysicalObject from Figure 17.7. In this case, the properties density and volume, and the component numbers, have distinct values for each system element described by the class, and are inherited by every subclass of PhysicalObject.

17.4.5 Using a Profile

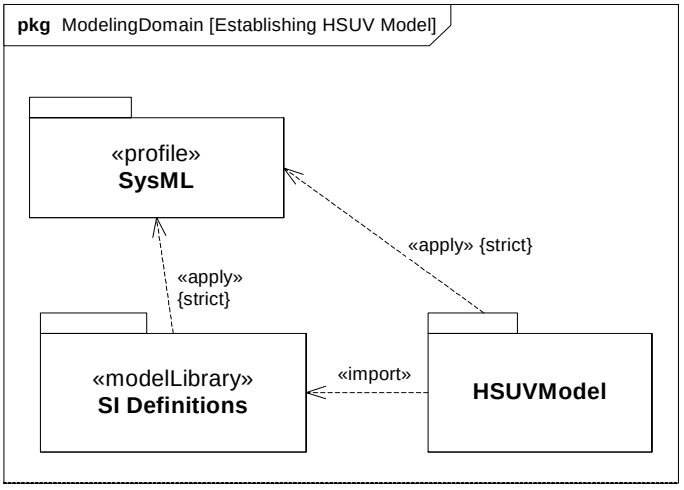


Figure 17.8 - A model with applied profile and imported model library

The HSUVModel is a systems engineering model that needs to use stereotypes from SysML. It therefore needs to have the SysML profile applied to it. In order to use the predefined SI units, it also needs to import the SI Definitions model library. Having done this, elements in HSUVModel can be extended by SysML stereotypes and types like SIVolume can be used to type properties. Both the SI Definitions model library and HSUVModel have applied the profile strictly which means that only those metaclasses directly referenced by SysML can be used in those models.

17.4.6 Using a Stereotype

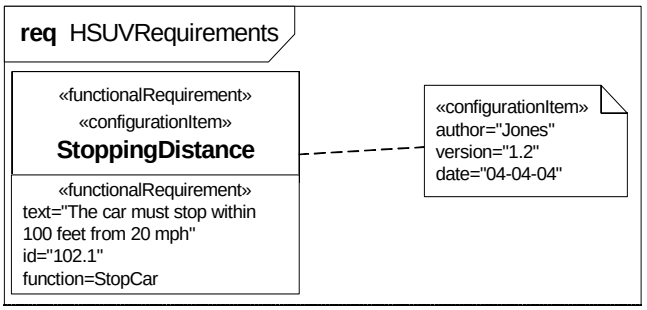


Figure 17.9 - Using two stereotypes on a model element

StoppingDistance has two stereotypes applied: functionalRequirement, which identifies it as a requirement that is satisfied by a function, and configurationItem, which allows it to have configuration management properties. The modeler has provided values for all the newly available properties; those for criticalRequirement are shown in a compartment in the node symbol for StoppingDistance; those for configurationItem are shown in a separate note.

17.4.7 Using a Model Library Element

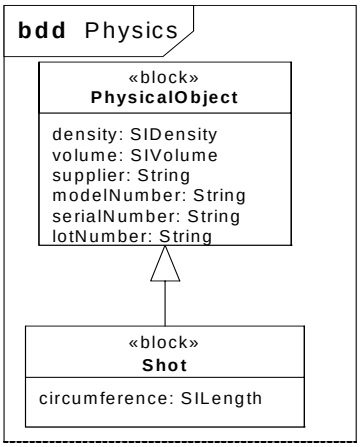


Figure 17.10 - Using model library elements

Model library elements can be used just like any other model element of the same type. In this case, Shot is a specialization of PhysicalObject from the Physical model library. It adds a new property, circumference, of type SILength to measure the circumference of the (spherical) shot.

Part V - Annexes

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

- A - Diagrams
- B - Deprecated Elements
- C - Sample Problem
- D - Non-normative Extensions
- E - Model Interchange
- F - Requirements Traceability

Annex A: Diagrams

(informative)

A.1 Overview

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

The SysML diagram taxonomy is shown in Figure A.1. This taxonomy is one example of how to organize the SysML diagrams. Other categories could also be defined, such as a grouping of the use case diagram and the requirement diagram into a category called Specification Diagrams.

SysML reuses many of the major diagram types of UML. In some cases, the UML diagrams are strictly reused, such as use case, sequence, state machine, and package diagrams, whereas in other cases they are modified so that they are consistent with SysML extensions. For example, the block definition diagram and internal block diagram are similar to the UML class diagram and composite structure diagram respectively, but include extensions as described in Clause 8, “Blocks.” Activity diagrams have also been modified via the activity extensions. Tabular representations, such as the allocation table, are used in SysML but are not considered part of the diagram taxonomy.

SysML does not use all of the UML diagram types such as the object diagram, communication diagram, interaction overview diagram, timing diagram, deployment diagram, and profile diagram. This is consistent with the approach that SysML represents a subset of UML. In the case of deployment diagrams, the deployment of software to hardware can be represented in the SysML internal block diagram. In the case of interaction overview and communication diagrams, it was felt that the SysML behavior diagrams provided adequate coverage for representing behavior without the need to include these diagram types. In the case of the profile diagram, profile definitions can be captured on a package diagram, which is also allowed in SysML. Two new diagram types have been added to SysML including the requirement diagram and the parametric diagram.

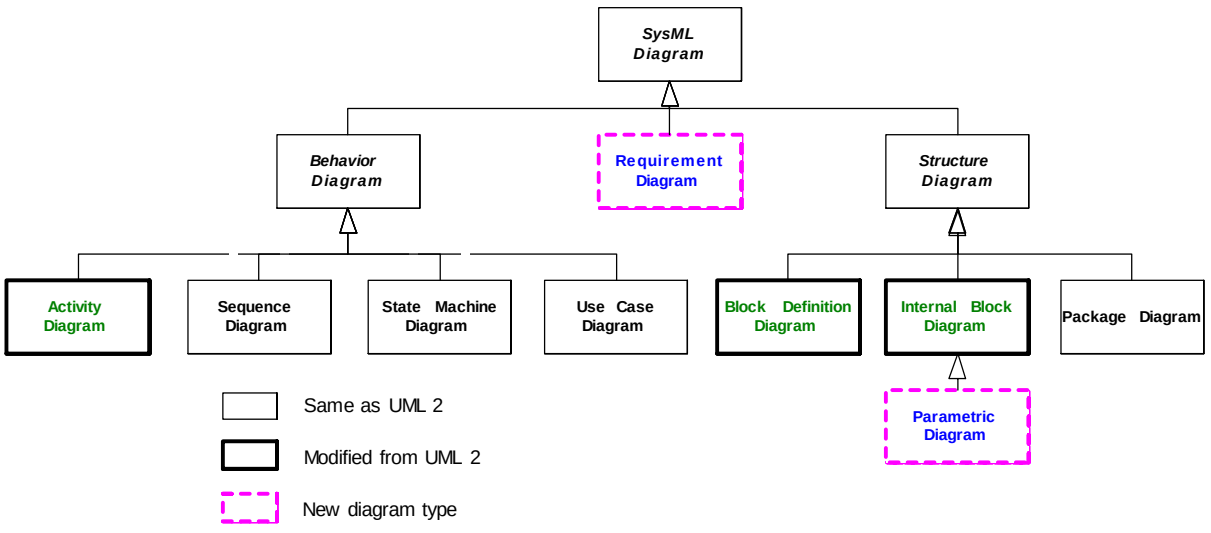


Figure A.1 - SysML Diagram Taxonomy

The requirement diagram is a new SysML diagram type. A requirement diagram provides a modeling construct for text-based requirements, and the relationship between requirements and other model elements that satisfy or verify them.

The parametric diagram is a new SysML diagram type that describes the constraints among the properties associated with blocks. This diagram is used to integrate behavior and structure models with engineering analysis models such as performance, reliability, and mass property models.

Although the taxonomy provides a logical organization for the various major kinds of diagrams, it does not preclude the careful mixing of different kinds of diagram types, as one might do when one combines structural and behavioral elements (e.g., showing a state machine nested inside a compartment of a block). However, it is critical that the types of diagram elements that can appear on a particular diagram kind be constrained and well-specified. The diagram elements tables in each clause describe what symbols can appear in the diagram, but do not specify the different combinations of symbols that can be used.

The package diagram and the callout notation are two mechanisms that SysML provides for adding flexibility to represent a broad range of diagram elements on diagrams. The package diagram can be used quite flexibly to organize the model in packages and views. As such, a package diagram can include a wide array of packageable elements. The callout notation provides a mechanism for representing relationships between model elements that appear on different diagram kinds. In particular, they are used to represent allocations and requirements, such as the allocation of an activity to a block on a block definition diagram, or showing a part that satisfies a particular requirement on an internal block diagram. There are other mechanisms for representing this including the compartment notation that is generally described in Clause 17, “Profiles & Model Libraries,” Clause 16, “Requirements,” and Clause 15, “Allocations” provide specific guidance on how these notations are used.

The model elements and corresponding concrete syntax that are represented in each of the nine SysML diagram kinds are described in the SysML clauses as indicated below.

- activity diagram - Activities (Clause 11)
- block definition diagram - Blocks (Clause 8), Ports and Flows (Clause 9)
- internal block diagram - Blocks (Clause 8), Ports and Flows (Clause 9)
- package diagram - Model Elements (Clause 7)
- parametric diagram - Constraint Blocks (Clause 10)
- requirement diagram - Requirements (Clause 16)
- state machine diagram - State Machines (Clause 13)
- sequence diagram - Interactions (Clause 12)
- use case diagram - Use Cases (Clause 14)

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

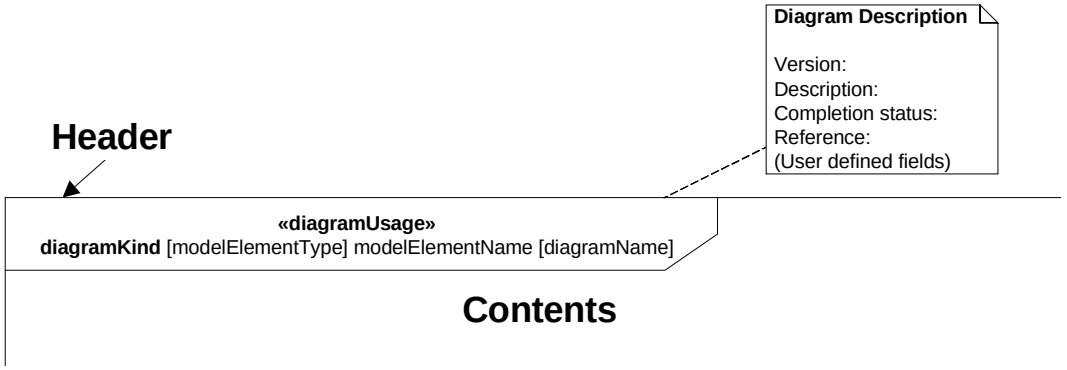


Figure A.2 - Diagram Frame

The frame is a rectangle that is required for SysML diagrams (Note: the frame is optional in UML). The frame must designate a model element that is the default namespace for the model elements enclosed in the frame. A qualified name for the model element within the frame must be provided if it is not contained within default namespace associated with the frame. The following are some of the designated model elements associated with the different diagram kinds.

- activity diagram - activity
- block definition diagram - block, package, or constraint block
- internal block diagram - block or constraint block
- package diagram - package or model
- parametric diagram - block or constraint block
- requirement diagram - package or requirement
- sequence diagram - interaction
- state machine diagram - state machine
- use case diagram - package

The frame may include border elements associated with the designated model element, like

- ports for blocks,
- entry/exit points on statemachines,
- gates on interactions,
- parameters for activities, and
- constraint parameters for constraint blocks.

The frame may sometimes be defined by the border of the diagram area provided by a tool.

The diagram contents area contains the graphical symbols. The diagram type and usage defines the type of primary graphical symbols that are supported, e.g., a block definition diagram is a diagram where the primary symbols in the contents area are blocks and association symbols along with their adornments.

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

<diagramKind> [modelElementType] <modelElementName> [diagramName]

A space separates each of these entries. The diagramKind is bolded. The modelElementType and diagramName are in brackets. The heading name should always contain the diagram kind and model element name, and include the model element type and additional information to remove ambiguity. Ambiguity can occur if there is more than one model element type for a given diagram kind, or where there is more than one diagram for the same model element. If a model element type has a stereotype applied to the base model element, such as “modelLibrary” applied to a package or “controlOperator” applied to an activity, then either the stereotype name or the base model element may be used as the name for the model element type. In either case, the initial character of the name is shown in lower case. For a stereotype name, guillemet characters (« and ») are not shown. If more than one stereotype has been applied to the base model element, either the name of one of the applied stereotypes or a comma-separated list of any or all of the applied stereotype names may be shown. If a base model element name is used, this element is either a UML metaclass which SysML uses directly, such as package or activity, or a stereotype which SysML defines on a UML metaclass, such as block or view.

SysML diagram kinds should have the following names or (abbreviations) as part of the heading:

- activity diagram (act)
- block definition diagram (bdd**)**
- internal block diagram (ibd)
- package diagram (pkg)
- parametric diagram (par)
- requirement diagram (req)
- sequence diagram (sd)
- state machine diagram (stm**)**
- use case diagram (uc)

The diagram description can be defined by a comment attached to a diagram frame as indicated in Figure A.2 that includes version, description, references to related information, a completeness field that describes the extent to which the modeler asserts the diagram is complete, and other-user defined fields. In addition, the diagram description may identify the view associated with the diagram, and the corresponding viewpoint that identifies the stakeholders and their concerns (refer to Model Elements clause). The diagram description can be made more explicit by the tool implementation.

SysML also introduces the concept of a diagram usage. This represents a unique usage of a particular diagram type, such as a context diagram as a usage of an block definition diagram, internal block diagram, or use case diagram. The diagram usage can be identified in the header above the diagramKind as «diagramUsage». An example of a diagram usage extension is shown in Figure A.3. For this example, the header in Figure A.2 would replace diagram kind with “uc” and «diagramUsage» with «ContextDiagram». Applying a stereotype approach to specify a diagram usage can allow a tool implementation to check that the diagram constraints defined by the stereotype are satisfied.

NOTE: A diagram is not a metaclass in UML or SysML and therefore cannot be extended by a stereotype. However, the concept of extending a diagram for a particular diagram usage was considered to be of value. The stereotype notation is used to designate this concept without the formal semantics.

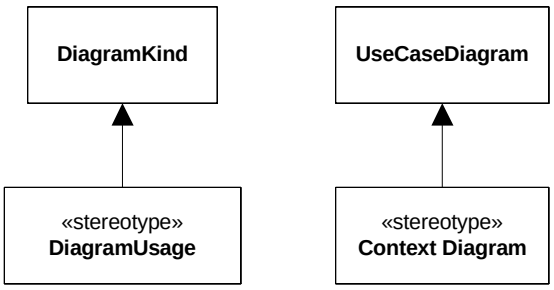


Figure A.3 - Diagram Usages

Some typical diagram usages may include:

- Activity diagram usage with swim lanes - SwimLane Diagram.
- Block definition diagram usage for a block hierarchy - Block Hierarchy where block can be replaced by system, item, activity, etc.
- Use case diagram or internal block diagram to represent a Context Diagram.

A.2 Guidelines

The following provides some general guidelines that apply to all diagram types.

- Decomposition of a model element can be represented by the rake symbol. This does not always mean decomposition in a formal sense, but rather a reference to a more elaborated diagram of the model element that includes the rake symbol. The rake on a model element may include the following:
  - activity diagram - call behavior actions that can refer to another activity diagram.
  - internal block diagram - parts that can refer to another internal block diagram.
  - package diagram - package that can refer to another package diagrams.
  - parametric diagram - constraint property that can refer to another parametric diagram
  - requirement diagram - requirement that can refer to another requirement diagram.
  - sequence diagram - interaction fragments that can refer to another sequence diagram.
  - state machine diagram - state that can refer to another state machine diagram.
  - use case diagram - use case can that may be realized by other behavior diagrams (activity, state, interactions).
- The primary mechanism for linking a text label outside of a symbol to the symbol is through proximity of the label to its symbol. This applies to ports, item flows, pins, etc.
- Page connectors (on-page connectors and off-page connectors) can be used to reduce the clutter on diagrams, but should be used sparingly since they are equivalent to go-to(s) in programming languages, and can lead to “spaghetti diagrams.” Whenever practical, elaborate the model element designated by the frame instead of using a page connector. A page connector is depicted as a circle with a label inside (often a letter). The circle is shown at both ends of a line break and means that the two line end connect at the circle.

- When two lines cross, the crossing optionally may be shown with a small semicircular jog to indicate that the lines do not intersect (as in electrical circuit diagrams), as shown in Figure A.4.

Figure A.4 - Optional Form of Line Crossing

- Diagram overlays are diagram elements that may be used on any diagram kind. An example of an overlay may be a geographic map to provide a spatial context for the symbols.
- SysML 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 described in this sub clause are intended to conform to diagram definition and interchange standards to facilitate exchange of diagram and layout information.
- Tabular and matrix representation is an optional alternative notation that can be used in conjunction with the graphical symbols as long as the information is consistent with the underlying metamodel. Tabular and matrix representations are often used in systems engineering to represent detailed information and other views of the model such as interface definitions, requirements traceability, and allocation relationships between various types of model elements. They also can be convenient mechanisms to represent property values for selected properties, and basic relationships such as function and inputs/outputs in N2 charts. The UML superstructure contains a tabular representation of a sequence diagram in an interaction matrix (refer to Superstructure Annex with interaction matrix). The implementations of tabular and matrix representations are defined by the tool implementations and are not standardized in SysML at this time. However, tabular or matrix representations may be included in a frame with the heading designator «table» or «matrix» in bold.
- Graph and tree representations are also optional, alternative notations that can be used in conjunction with graphical symbols as long as the information is consistent with the underlying metamodel. These representations can be used for describing complex series of relationships that represent other views of the model. One example is the browser window in many tools that depicts a hierarchical view of the model. The implementations of graphs and trees are defined by the tool implementations and are not standardized in SysML at this time. However, graph and tree representations may be included in a frame with the heading designator «graph» or «tree» in bold.

Annex B: Deprecated Elements

(normative)

B.1 Overview

Flow Port and Flow Specification are deprecated in this version of SysML and are defined for backward compatibility. This annex contains the definition of these concepts as they are defined by SysML 1.2. In addition it provides some guidelines on how to convert FlowPort to ports in this version of SysML.

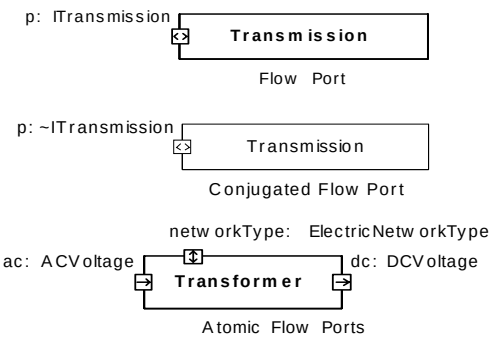
B.1.1 Flow Ports

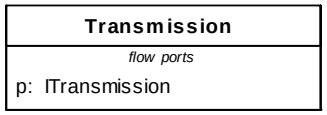
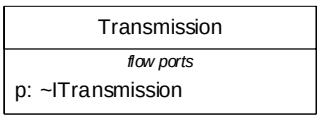
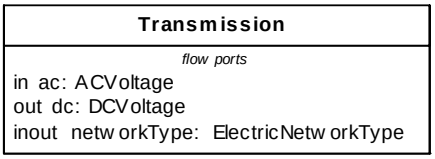
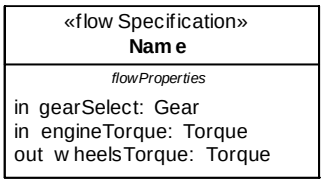
A flow port specifies the input and output items that may flow between a block and its environment. Flow ports are interaction points through which data, material, or energy can enter or leave the owning block. The specification of what can flow is achieved by typing the flow port with a specification of things that flow. This can include typing an atomic flow port with a single type representing the items that flow in or out, or typing a nonatomic flow port with a flow specification which lists multiple items that flow. A block representing an automatic transmission in a car could have an atomic flow port that specifies “Torque” as an input and another atomic flow port that specifies “Torque” as an output. A more complex flow port could specify a set of signals and/or properties that flow in and out of the flow port. In general, flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions. Flow ports extend UML 2 ports.

B.2 Diagram Elements

B.2.1 Block Definition Diagram

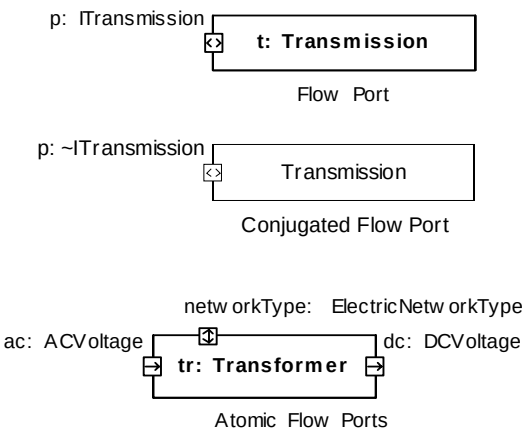
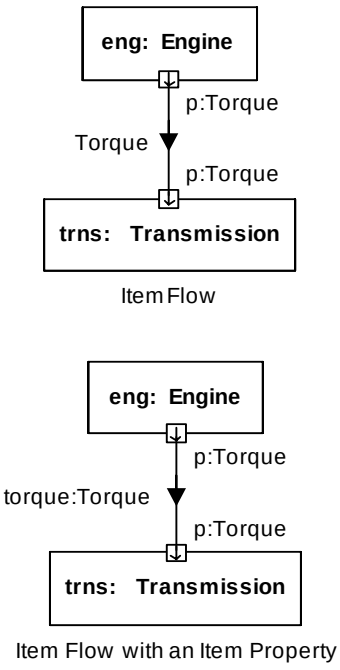
Table B.1 - Graphical nodes defined in block definition diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
FlowPort		**SysML::Ports&Flows::FlowPort**

Node Name	Concrete Syntax	Abstract Syntax Reference
FlowPort (Compartment Notation)	 Flow Port  Conjugated Flow Port  Atomic Flow Ports	**SysML::Ports&Flows::FlowPort**
FlowSpecification		**SysML::Ports&Flows:: FlowSpecification**

B.2.2 Internal Block Diagram

Table B.2 - Graphical nodes defined in internal block diagrams

Node Name	Concrete Syntax	Abstract Syntax Reference
FlowPort		**SysML::Ports&Flows::FlowPort**
ItemFlow		**SysML::Ports&Flows::ItemFlow**

B.3 UML Extensions

B.3.1 Diagram Extensions

B.3.1.1 FlowPort

A FlowPorts is an interaction point through which input and/or output of items such as data, material, or energy may flow. The notation of flow port is a square on the boundary of the owning block or its usage. The label of the flow port is in the format *portName: portType*. Atomic flow ports have an arrow inside them indicating the direction of the port with respect to the owning Block. A nonatomic flow port has two open arrow heads facing away from each other (i.e., < >). The fill color of the square is white and the line and text colors are black.

In addition, flow ports can be listed in a special compartment labeled “flow ports.” The format of each line is:

in | out | inout portName:portType [{conjugated}]

B.3.1.2 FlowSpecification

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

B.3.2 Stereotypes

B.3.2.1 Package Ports&Flows

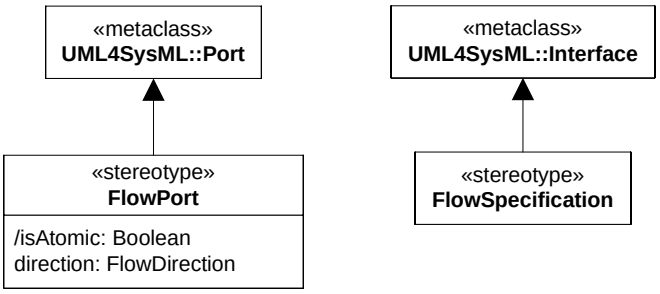


Figure B.1 - Deprecated Stereotypes

B.3.2.2 FlowPort

Description

A FlowPort is an interaction point through which input and/or output of items such as data, material, or energy may flow. This enables the owning block to declare which items it may exchange with its environment and the interaction points through which the exchange is made.

We distinguish between atomic flow port and a nonatomic flow port. Atomic flow ports relay items that are classified by a single Block, ValueType, or Signal classifier. A nonatomic flow port relays items of several types as specified by a FlowSpecification.

The distinction between atomic and nonatomic flow ports is made according to the flow port's type: If a flow port is typed by a flow specification, then it is nonatomic; if a flow port is typed by a Block, ValueType, or Signal classifier, then it is atomic.

Flow ports and associated flow specifications define "what can flow" between the block and its environment, whereas item flows specify "what does flow" in a specific usage context.

Flow ports relay items to their owning block or to a connector that connects them with their owner's internal parts (internal connector).

The isBehavior attribute inherited from UML port is interpreted in the following way: if isBehavior is set to true, then the items are relayed to/from the owning block. More specifically, every flow property within the flow port is bound to a property owned by the port's owning block or to a parameter of its behavior. If isBehavior is set to false, then the flow port must be connected to an internal connector, which in turn related the items via the port. The need for isBehavior is mainly to allow specification of internal parts relaying items to their containing part via flow ports.

The isConjugated attribute inherited from the UML Port metaclass is interpreted as follows: It indicates if the flows of items of a nonatomic flow port maintain the directions specified in the flow specification or if the direction of every flow property specified in the flow specification is reversed (IN becomes OUT and vice versa). If set to True, then all the directions of the flow properties specified by the flow specification that types a nonatomic flow port are relayed in the opposite direction (i.e., an "in" flow property is treated as an "out" flow property by the flow port and vice-versa). By default, the value is False. This attribute applies only to nonatomic flow ports since atomic flow ports have a direction attribute signifying the direction of the flow.

In case of flow properties or atomic flow ports of type Signal, inbound properties or atomic flow port are mapped to a Reception of the signal type (or a subtype) of the flow property's type. Outbound flow properties only declare the ability of the flow port to relay the signal over external connectors attached to it and are not mapped to a property of the flow port's owning block.

The item flows specified as flowing on a connector between flow ports must match the flow properties of the ports at each end of the connector: the source of the item flow should be the port that has an outbound/bidirectional flow property that matches the item flow's type and the target of the item flow should be the port that has an inbound/bidirectional flow property that matches the type of the item flow.

If a flow port is connected to multiple external and/or internal connectors, then the items are propagated (broadcast) over all connectors that have matching properties at the other end.

B.3.2.3 Semantic Variation Points

The binding of the flow properties on the ports to behavior parameters and/or block properties is a semantic variation point. One approach is to perform name and type matching. Another approach is to explicitly use binding relationships between the ports properties and behavior parameters or block properties.

Attributes

- /isAtomic : Boolean (derived)
  This is a derived attribute (derived from the flow port's type). For a flow port typed by a flow specification the value of this attribute is False, otherwise the value is True.
- direction : FlowDirection
  Indicates the direction in which an atomic flow port relays its items. If the direction is set to "in," then the items are relayed from an external connector via the flow port into the flow port's owner (or one of its parts). If the direction is set to "out," then the items are relayed from the flow port's owner, via the flow port, through an

external connector attached to the flow port. If the direction is set to “inout,” then items can flow both ways. By default, the value is inout.

Constraints

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

[2] If the FlowPort is atomic (by its type), then isAtomic=True, the direction must be specified (has a value), and isConjugated is not specified (has no value).

[3] If the FlowPort is nonatomic, and the FlowSpecification typing the port has flow properties with direction “in,” the FlowPort direction is “in” (or “out” if isConjugated=true). If the flow properties are all out, the FlowPort direction is out (or in if isConjugated=true). If flow properties are both in and out, the direction is inout.

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

1. Type Matching: The type being sent is the same type or a subtype of the type being received.
2. Direction Matching: If the connector connects two parts that are external to one another, then the direction of the flow properties must be opposite, or at least one of the ends should be inout. If the connector is internal to the owner of one of the flow ports, then the direction should be the same or at least one of the ends should be inout.
3. Name Matching: In case there is type and direction match to several flow properties at the other end, the property that has the same name at the other end is selected. If there is no such property, then the connection is ambiguous (ill-formed).

[5] If a flow port is not connected to an internal part, then isBehavior should be set to true.

B.3.2.4 FlowSpecification

Description

A FlowSpecification specifies inputs and outputs as a set of flow properties. A flow specification is used by flow ports to specify what items can flow via the port.

Constraints

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

[2] Every “ownedAttribute” of a FlowSpecification must be a FlowProperty.

B.3.2.5 ItemFlow (deprecated compatibility rule)

ItemFlows are not deprecated, but when used with atomic flows ports, have a deprecated modification of item flow compatibility rules that treats types of source and target atomic ports as if they were types of flow properties on types of those ports.

B.4 Transitioning SysML 1.2 Flow Ports to SysML 1.3 Ports (informative)

To convert a SysML 1.2 flow port to ports in this version of SysML it is recommended to use the following guidelines:

1. Decide if the port should be converted to an proxy port,a full port, or an unstereotyped port.

2. Based on the decision in step 1, create a block (for proxy ports, it must be an interface block specifically).

3. If the original flow port is non-atomic:

   a. Copy all the flow properties owned by the flow port's type, a flow specification, to the block created in step 2 (meaning the flow properties will be owned by the newly created block).

   b. Replace the type of the port with the block created in step 2.

   c. Remove the flow port stereotype from the port.

   d. Based on the decision in step 1, apply the ProxyPort or FullPort stereotype, or do nothing if the decision is not to use either one.

   e. If the proxy stereotype is applied in step 3d, and there is a single connector from the port to a part, the BindingConnector may be applied to the connector.

   f. If the flow specification is not referenced by other model elements, delete it.

4. If the original flow port is atomic:

   a. On the block created in step 2, specify a flow property typed by the same type as the flow port and with the same direction as the original flow port.

   b. Do steps b to d from step 3 about non-atomic flow ports.

Annex C: Sample Problem

(informative)

C.1 Purpose

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

C.2 Scope

The scope of this example is to provide at least one diagram for each SysML diagram type. The intent is to select simplified fragments of the problem to illustrate how the diagrams can be applied, and to demonstrate some of the possible inter-relationships among the model elements in the different diagrams. The sample problem does not highlight all of the features of the language. The reader should refer to the individual clauses for more detailed features of the language. The diagrams selected for representing a particular aspect of the model, and the ordering of the diagrams are intended to be representative of applying a typical systems engineering process, but this will vary depending on the specific process and methodology that is used.

C.3 Problem Summary

The sample problem describes the use of SysML as it applies to the development of an automobile, in particular a Hybrid gas/electric powered Sport Utility Vehicle (SUV). This problem is interesting in that it has inherently conflicting requirements, viz. desire for fuel efficiency, but also desire for large cargo carrying capacity and off-road capability. Technical accuracy and the feasibility of the actual solution proposed were not high priorities. This sample problem focuses on design decisions surrounding the power subsystem of the hybrid SUV; the requirements, performance analyses, structure, and behavior.

This annex is structured to show each diagram in the context of how it might be used on such an example problem. The first sub clause shows SysML diagrams as they might be used to establish the system context; establishing system boundaries, and top level use cases. The next sub clause is provided to show how SysML diagrams can be used to analyze top level system behavior, using sequence diagrams and state machine diagrams. The following sub clause focuses on use of SysML diagrams for capturing and deriving requirements, using diagrams and tables. A sub clause is provided to illustrate how SysML is used to depict system structure, including block hierarchy and part relationships. The relationship of various system parameters, performance constraints, analyses, and timing diagrams are illustrated in the next sub clause. A sub clause is then dedicated to illustrating definition and depiction of interfaces and flows in a structural context. The final sub clause focuses on detailed behavior modeling, functional and flow allocation.

C.4 Diagrams

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

C.4.1.1 Package Diagram - Applying the SysML Profile

As shown in Figure C.1, the HSUVModel is a package that represents the user model. The SysML Profile must be applied to this package in order to include stereotypes from the profile. The HSUVModel may also require model libraries, such as the SI Units Types model library. The model libraries must be imported into the user model as indicated.

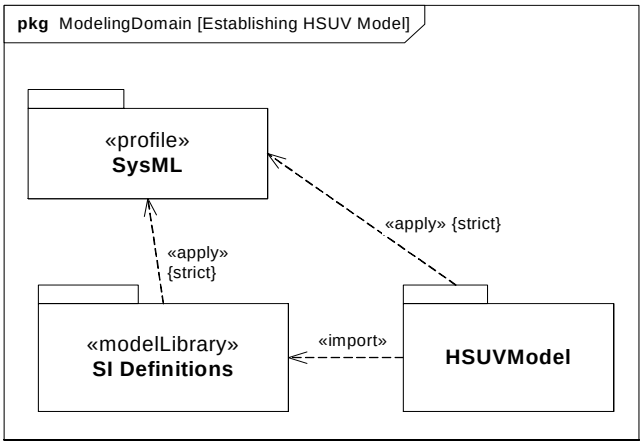


Figure C.1 - Establishing the User Model by Importing and Applying SysML Profile & Model Library (Package Diagram)

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

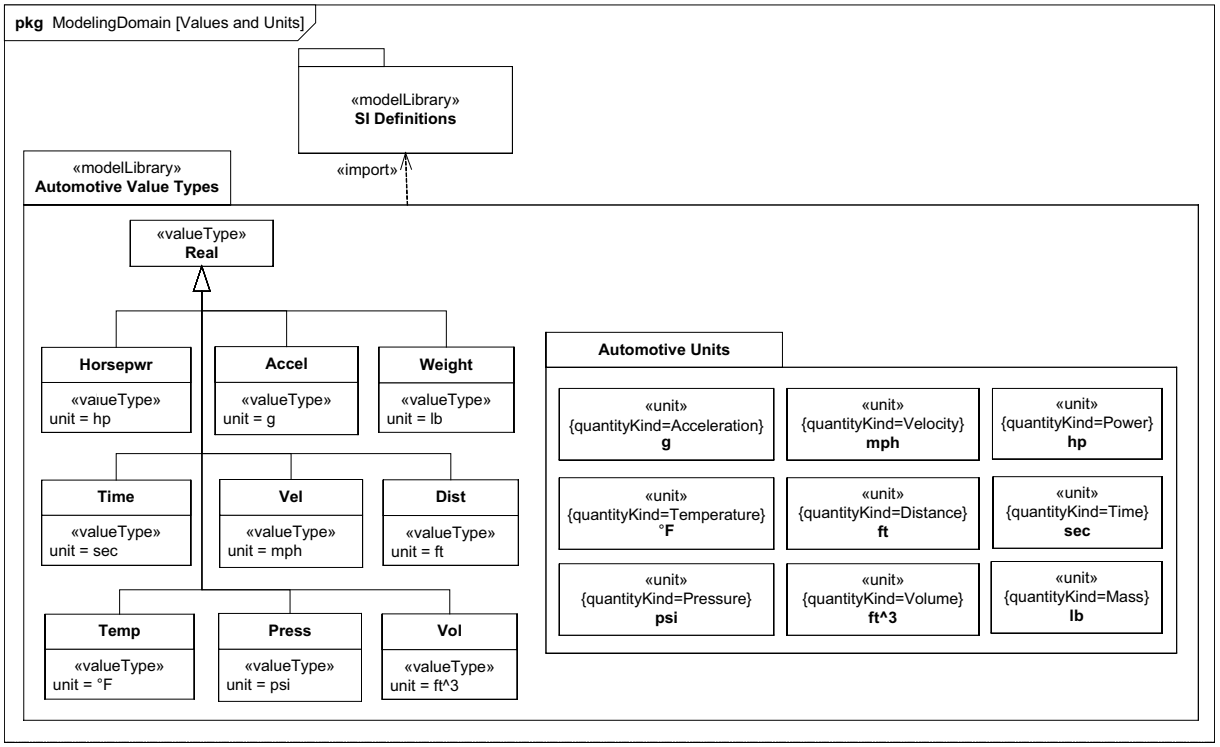


Figure C.2 - Defining valueTypes and units to be Used in the Sample Problem

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

The package diagram (Figure C.3) shows the structure of the model used to evaluate the sample problem. Model elements are contained in packages, and relationships between packages (or specific model elements) are shown on this diagram. The relationship between the views (OperationalView and PerformanceView) and the rest of the user model are explicitly expressed using the «import» relationship. Note that the «view» models contain no model elements of their own, and that changes to the model in other packages are automatically updated in the Operational and Performance Views.

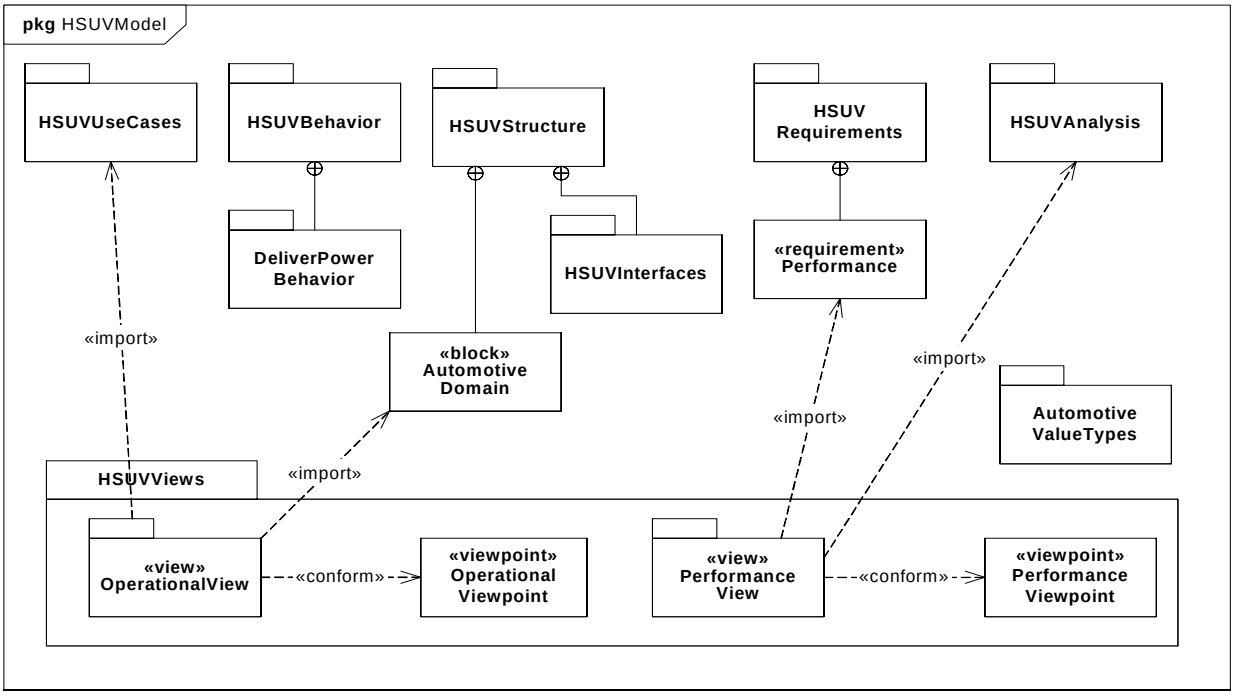


Figure C.3 - Establishing Structure of the User Model using Packages and Views (Package Diagram)

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

C.4.2.1 Internal Block Diagram - Setting Context

The term “context diagram,” in Figure C.4, refers to a user-defined usage of an internal block diagram, which depicts some of the top-level entities in the overall enterprise and their relationships. The diagram usage enables the modeler or methodologist to specify a unique usage of a SysML diagram type using the extension mechanism described in Annex A, “Diagrams.” The entities are conceptual in nature during the initial phase of development, but will be refined as part of the development process. The «system» and «external» stereotypes are user defined, not specified in SysML, but help the modeler to identify the system of interest relative to its environment. Each model element depicted may include a graphical icon to help convey its intended meaning. The spatial relationship of the entities on the diagram sometimes conveys understanding as well, although this is not specifically captured in the semantics. Also, a background such as a map can be included to provide additional context. The associations among the classes may represent abstract conceptual relationships among the entities, which would be refined in subsequent diagrams. Note how the relationships in this diagram are also reflected in the Automotive Domain Model Block Definition Diagram, Figure C.15.

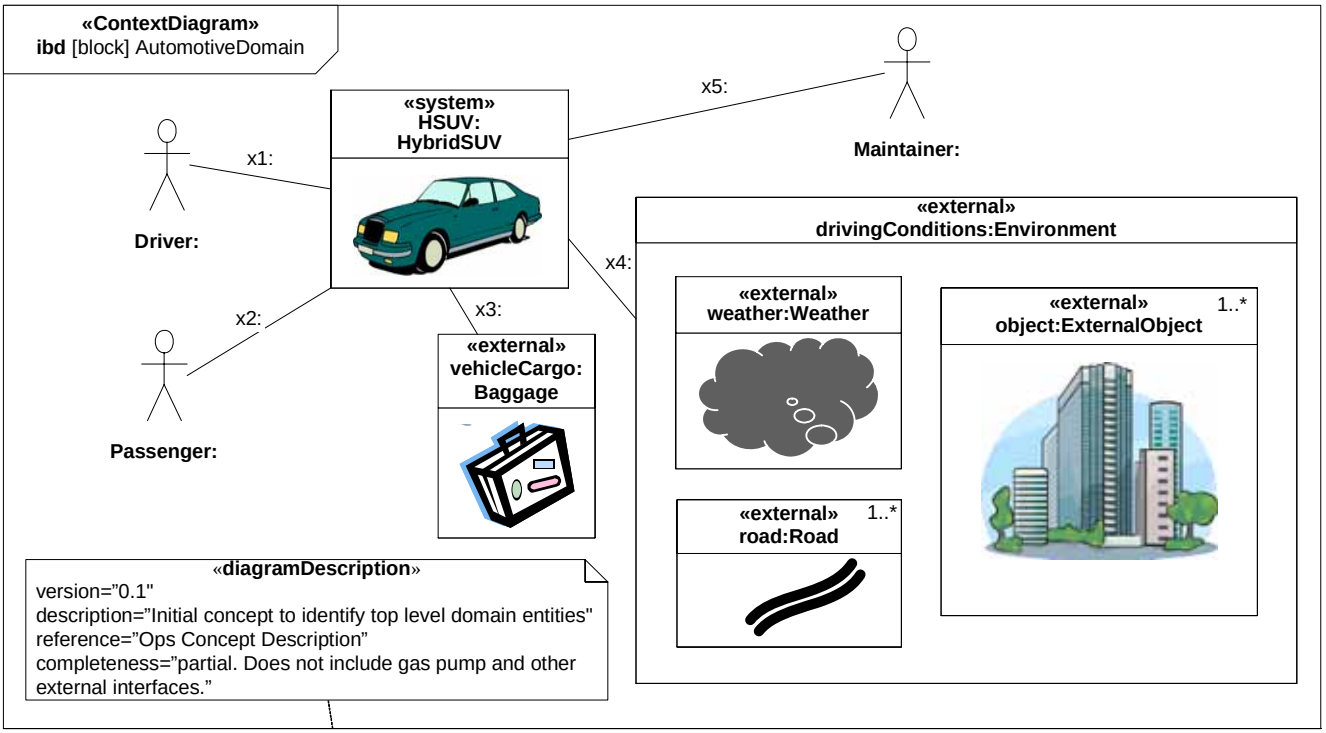


Figure C.4 - Establishing the Context of the Hybrid SUV System using a User-Defined Context Diagram. (Internal Block Diagram) Completeness of Diagram Noted in Diagram Description

C.4.2.2 Use Case Diagram - Top Level Use Cases

The use case diagram for "Drive Vehicle" in Figure C.5 depicts the drive vehicle usage of the vehicle system. The subject (HybridSUV) and the actors (Driver, Registered Owner, Maintainer, Insurance Company, DMV) interact to realize the use case.

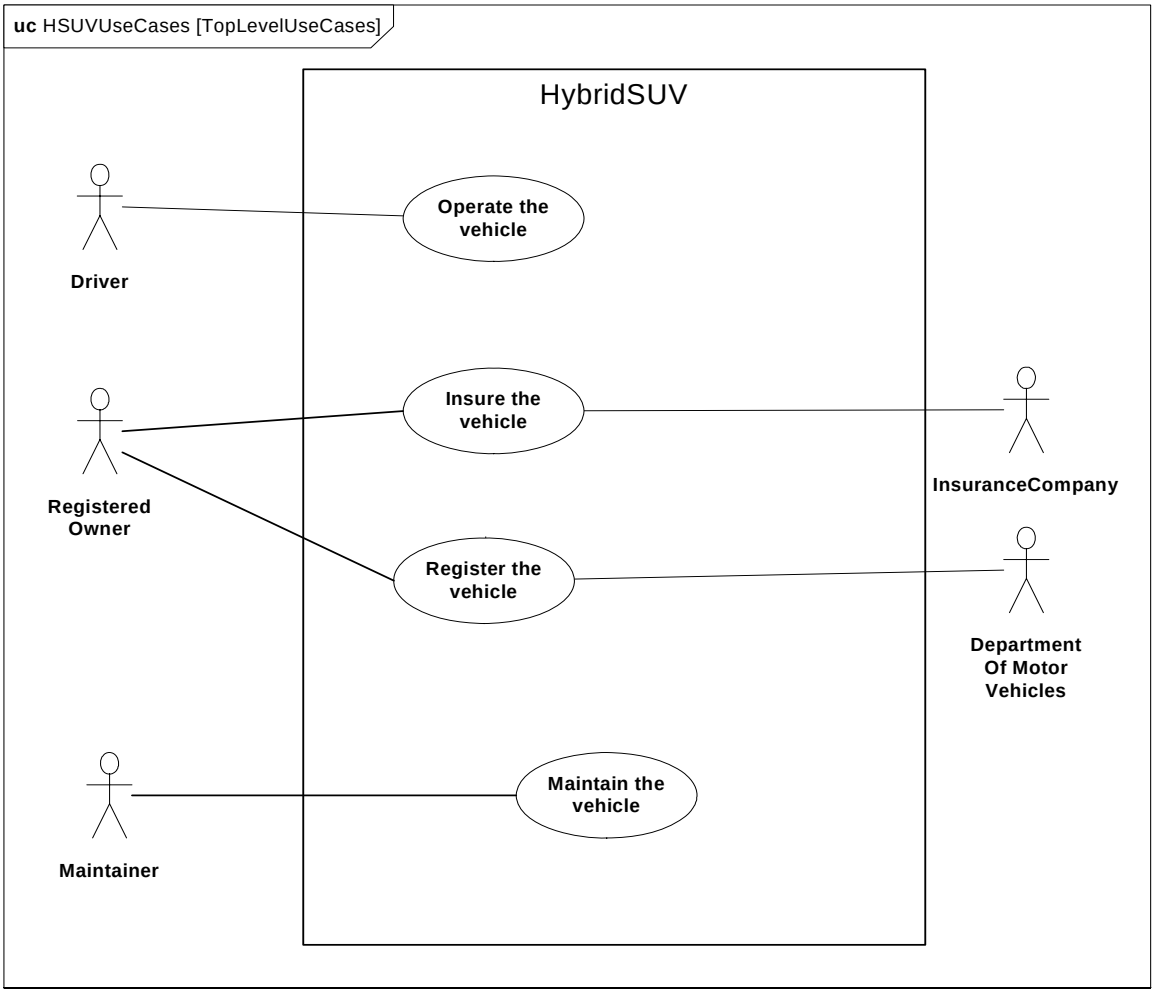


Figure C.5 - Establishing Top Level Use Cases for the Hybrid SUV (Use Case Diagram)

C.4.2.3 Use Case Diagram - Operational Use Cases

Goal-level Use Cases associated with “Operate the Vehicle” are depicted in the following diagram. These use cases help flesh out the specific kind of goals associated with driving and parking the vehicle. Maintenance, registration, and insurance of the vehicle would be covered under a separate set of goal-oriented use cases.

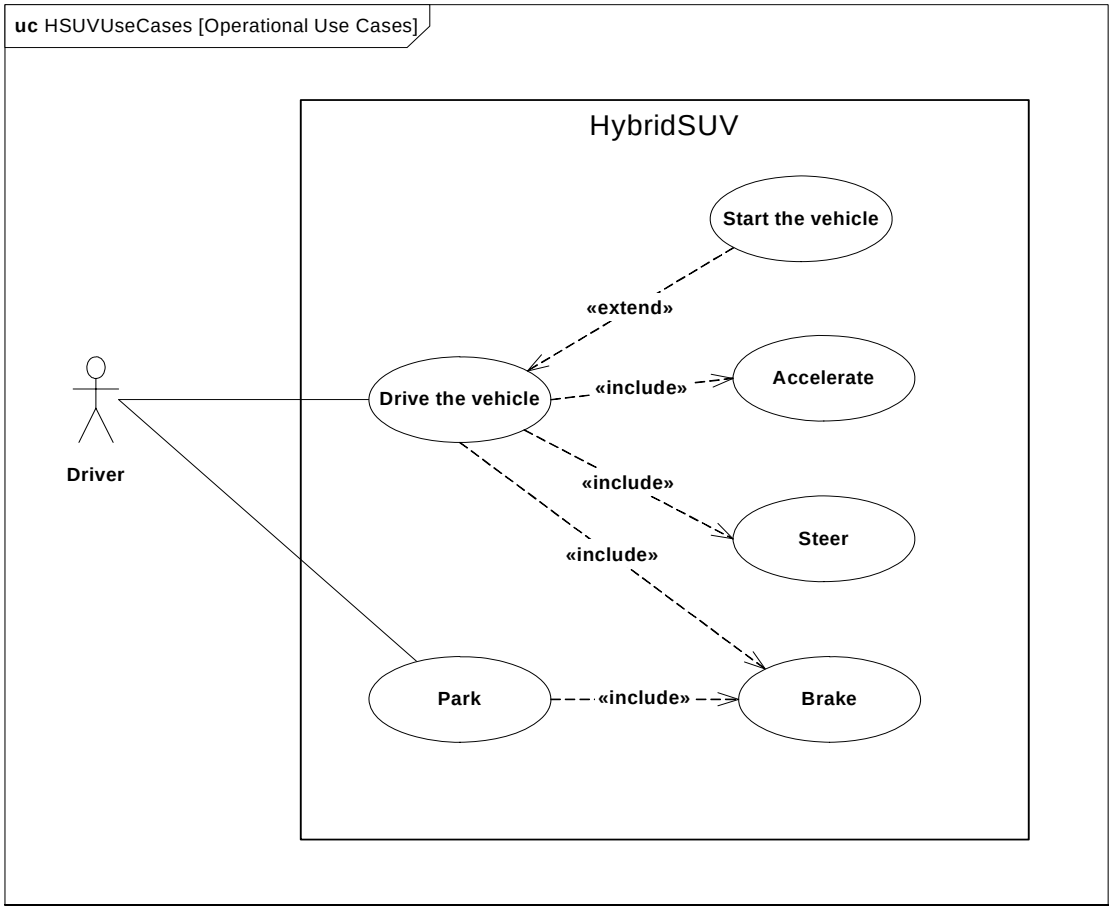


Figure C.6 - Establishing Operational Use Cases for "Drive the Vehicle" (Use Case Diagram)

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

C.4.3.1 Sequence Diagram - Drive Black Box

Figure C.7 shows the interactions between driver and vehicle that are necessary for the "Drive the Vehicle" Use Case. This diagram represents the "DriveBlackBox" interaction, with is owned by the AutomotiveDomain block. "BlackBox" for the purpose of this example, refers to how the subject system (HybridSUV block) interacts only with outside elements, without revealing any interior detail.

The conditions for each alternative in the alt controlSpeed sub clause are expressed in OCL, and relate to the states of the HybridSUV block, as shown in Figure C.8.

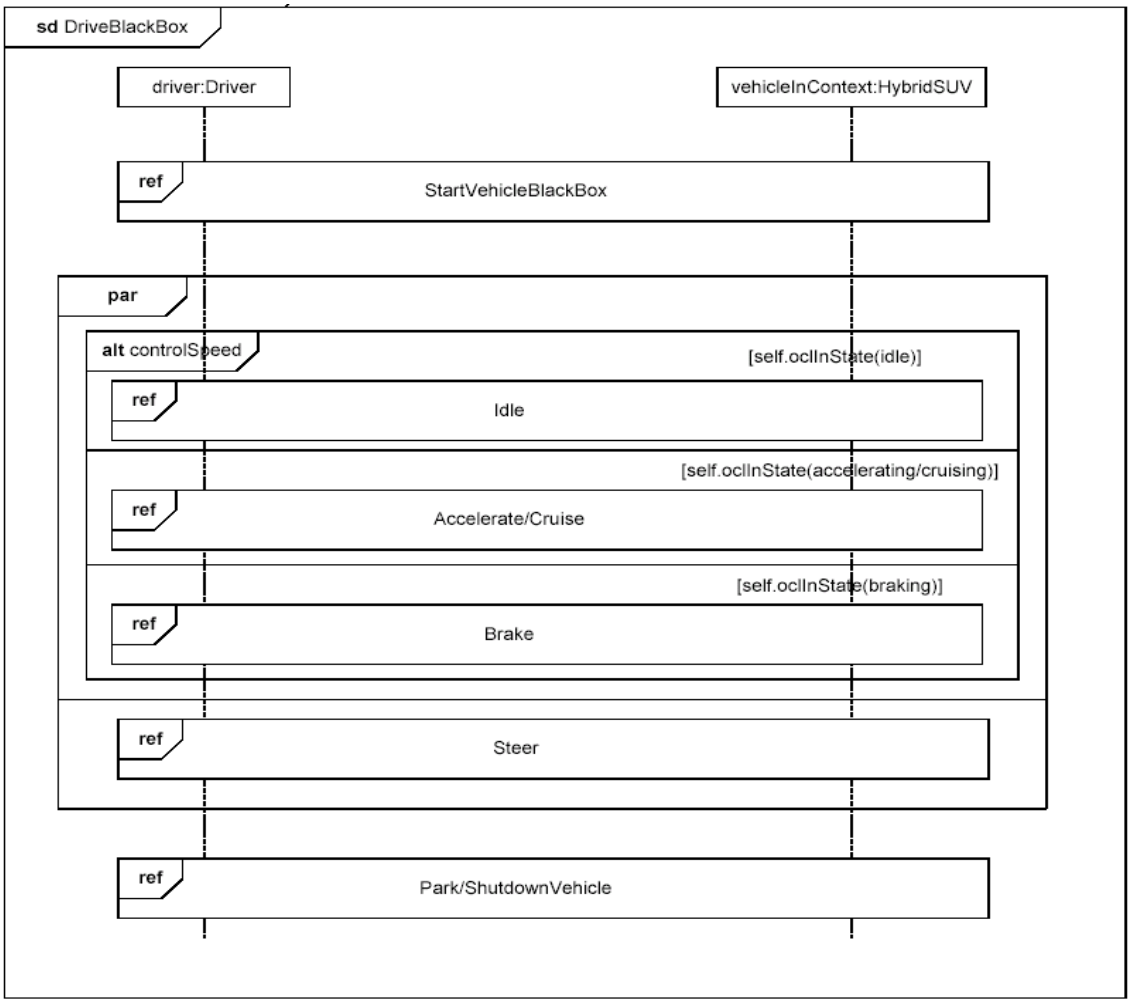


Figure C.7 - Elaborating Black Box Behavior for the "Drive the Vehicle" Use Case (Sequence Diagram)

C.4.3.2 State Machine Diagram - HSUV Operational States

Figure C.8 depicts the operational states of the HSUV block, via a State Machine named "HSUVOperationalStates." Note that this state machine was developed in conjunction with the DriveBlackBox interaction in Figure C.7. Also note that this state machine refines the requirement "PowerSourceManagment," which will be elaborated in the requirements sub clause of this sample problem. This diagram expresses only the nominal states. Exception states, like "acceleratorFailure," are not expressed on this diagram.

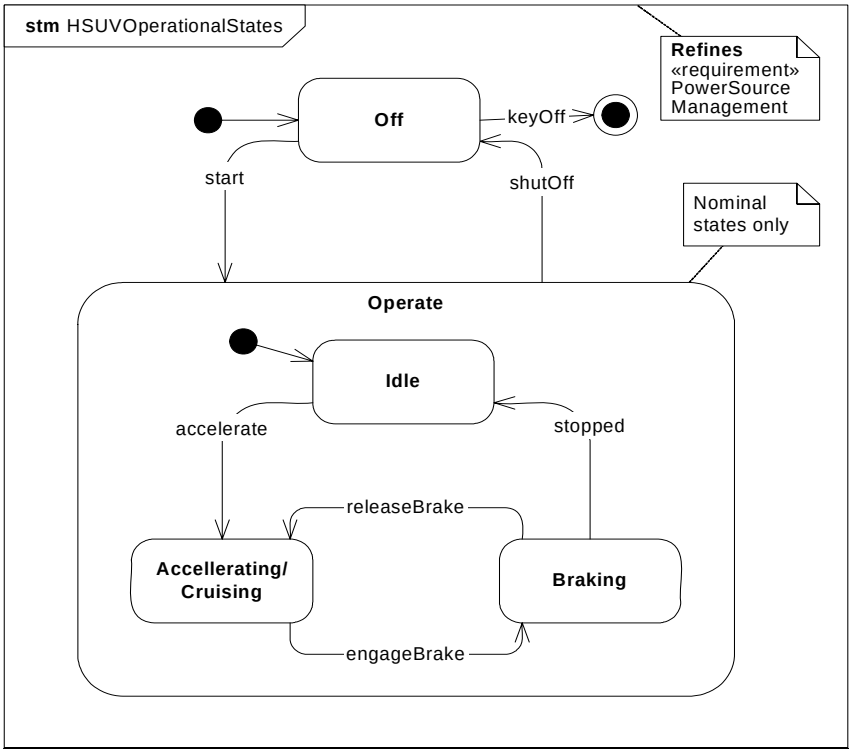


Figure C.8 - Finite State Machine Associated with "Drive the Vehicle" (State Machine Diagram)

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

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

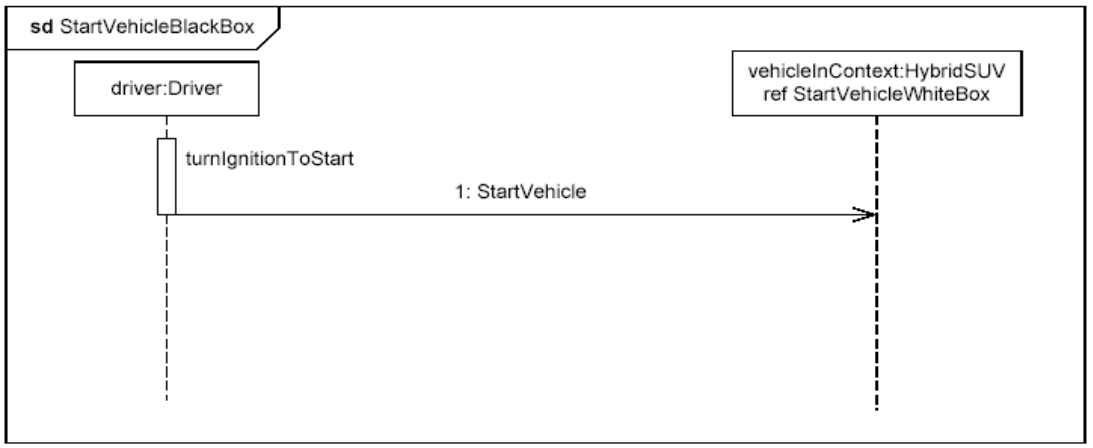


Figure C.9 - Black Box Interaction for "StartVehicle," referencing White Box Interaction (Sequence Diagram)

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

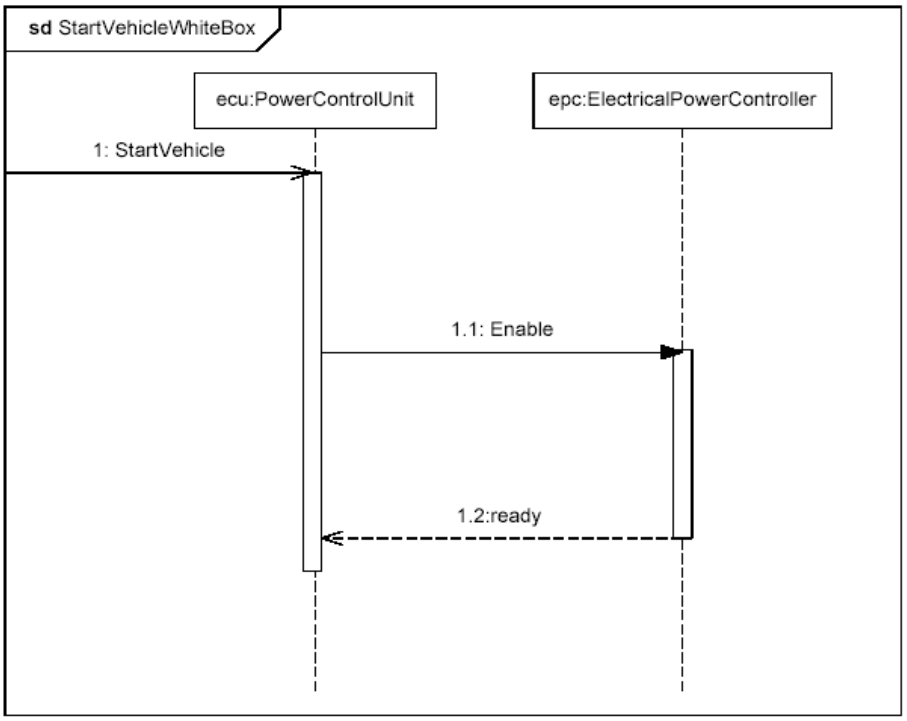


Figure C.10 - White Box Interaction for “StartVehicle” (Sequence Diagram)

C.4.4 Establishing Requirements (Requirements Diagrams and Tables)

C.4.4.1 Requirement Diagram - HSUV Requirement Hierarchy

The vehicle system specification contains many text based requirements. A few requirements are highlighted in Figure C.11, including the requirement for the vehicle to pass emissions standards, which is expanded for illustration purposes. The containment (cross hair) relationship, for purposes of this example, refers to the practice of decomposing a complex requirement into simpler, single requirements.

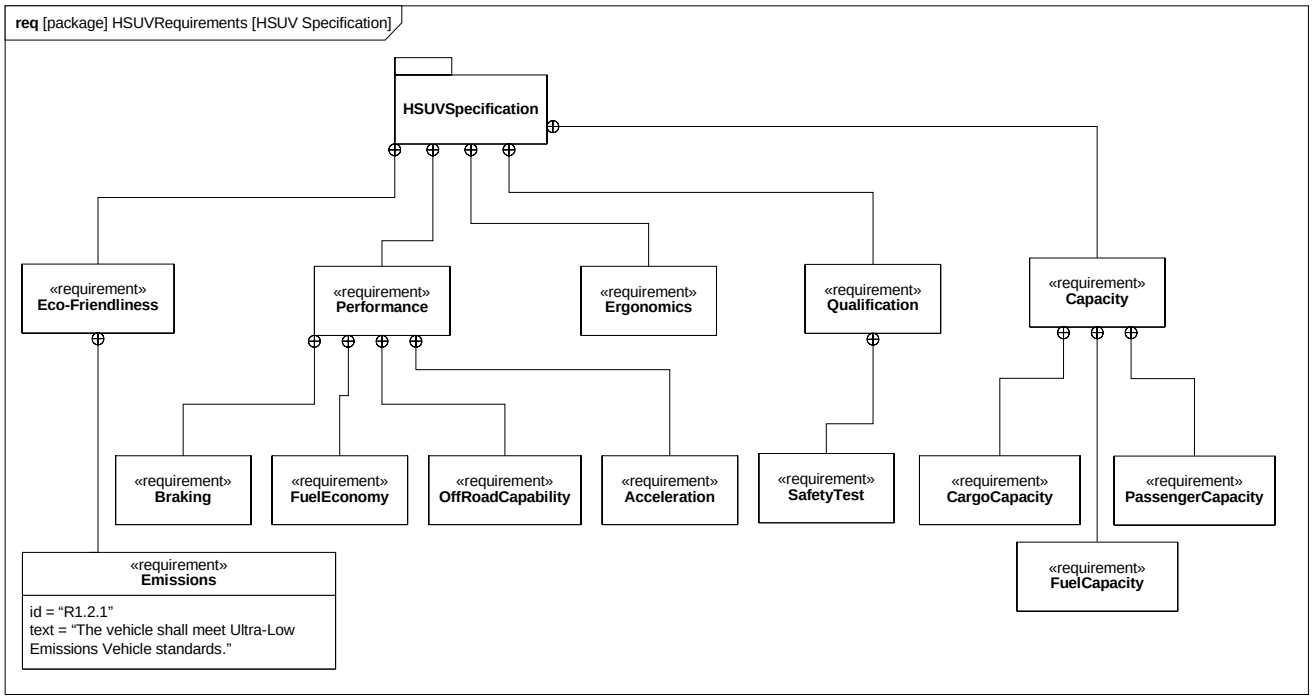


Figure C.11 - Establishing HSUV Requirements Hierarchy (containment) - (Requirements Diagram)

C.4.4.2 Requirement Diagram - Derived Requirements

Figure C.12 shows a set of requirements derived from the lowest tier requirements in the HSUV specification. Derived requirements, for the purpose of this example, express the concepts of requirements in the HSUVSpecification in a manner that specifically relates them to the HSUV system. Various other model elements may be necessary to help develop a derived requirement, and these model element may be related by a «refinedBy» relationship. Note how PowerSourceManagement is "RefinedBy" the HSUVOperationalStates model (Figure C.8). Note also that rationale can be attached to the «deriveReqt» relationship. In this case, rationale is provided by a referenced document "Hybrid Design Guidance."

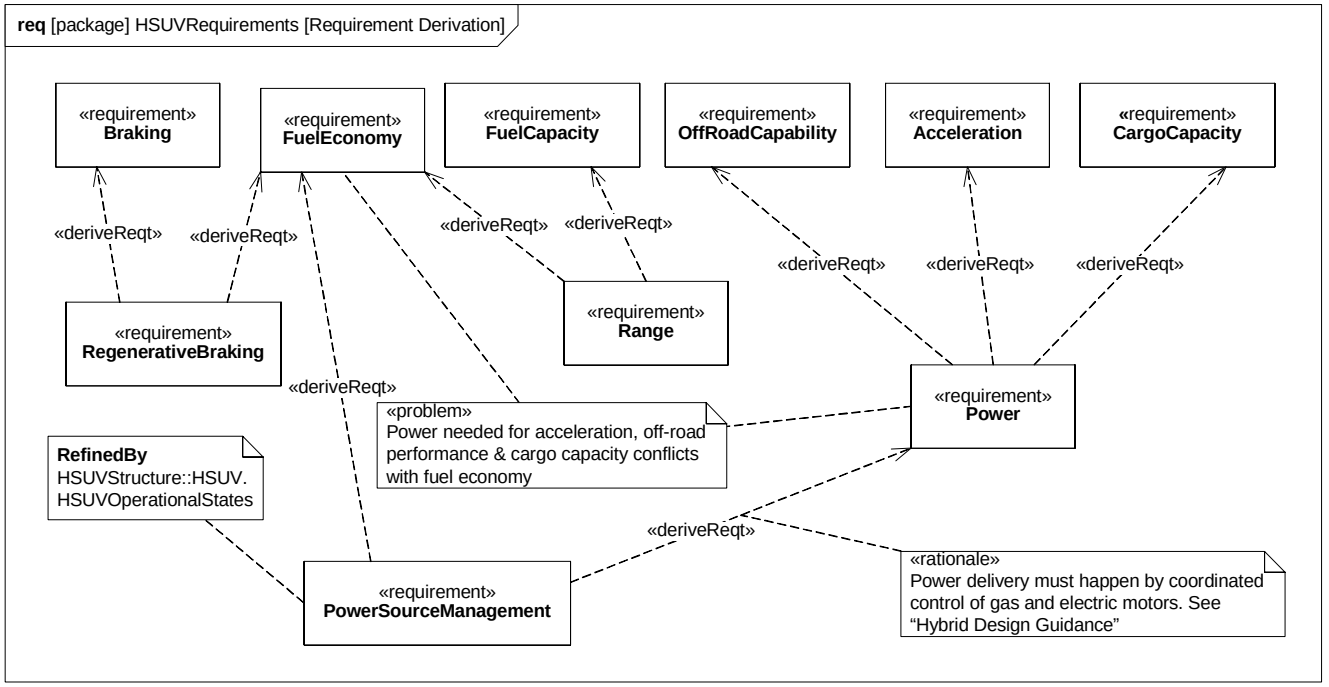


Figure C.12 - Establishing Derived Requirements and Rationale from Lowest Tier of Requirements Hierarchy (Requirements Diagram)

C.4.4.3 Requirement Diagram - Acceleration Requirement Relationships

Figure C.13 focuses on the Acceleration requirement, and relates it to other requirements and model elements. The "refine" relation, introduced in Figure C.12, shows how the Acceleration requirement is refined by a similarly named use case. The Power requirement is satisfied by the PowerSubsystem, and a Max Acceleration test case verifies the Acceleration requirement.

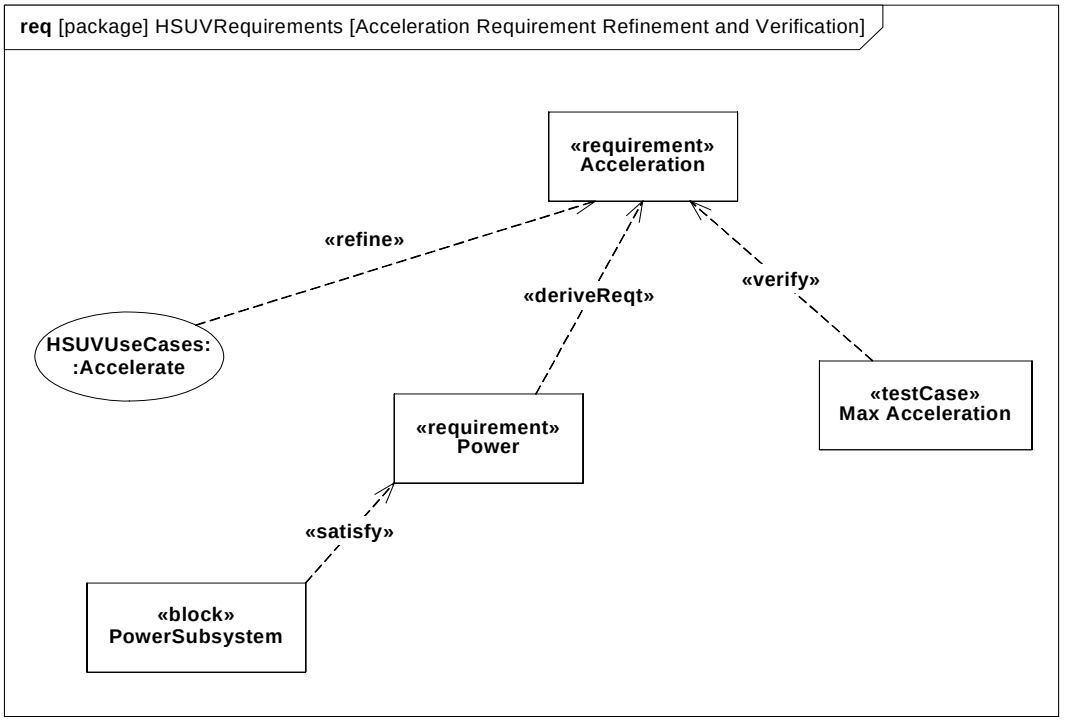


Figure C.13 - Acceleration Requirement Relationships (Requirements Diagram)

C.4.4.4 Table - Requirements Table

Figure C.14 contains two diagrams that show requirement containment (decomposition), and requirements derivation in tabular form. This is a more compact representation than the requirements diagrams shown previously.

table [requirement] Performance [Decomposition of Performance Requirement]

id	name	text
2	Performance	The Hybrid SUV shall have the braking, acceleration, and off-road capability of a typical SUV, but have dramatically better fuel economy.
2.1	Braking	The Hybrid SUV shall have the braking capability of a typical SUV.
2.2	FuelEconomy	The Hybrid SUV shall have dramatically better fuel economy than a typical SUV.
2.3	OffRoadCapability	The Hybrid SUV shall have the off-road capability of a typical SUV.
2.4	Acceleration	The Hybrid SUV shall have the acceleration of a typical SUV.

table [requirement] Performance [Tree of Performance Requirements]

id	name	relation	id	name	relation	id	name
2.1	Braking	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.2	Range			
4.2	FuelCapacity	deriveReqt	d.2	Range			
2.3	OffRoadCapability	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
2.4	Acceleration	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
4.1	CargoCapacity	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement

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

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

C.4.5.1 Block Definition Diagram - Automotive Domain

Figure C.15 provides definition for the concepts previously shown in the context diagram. Note that the interactions DriveBlackBox and Stac4rtVehicleBlackBox (described in Section C.4.3, “Elaborating Behavior (Sequence and State Machine Diagrams),” on page 186) are depicted as owned by the AutomotiveDomain block.

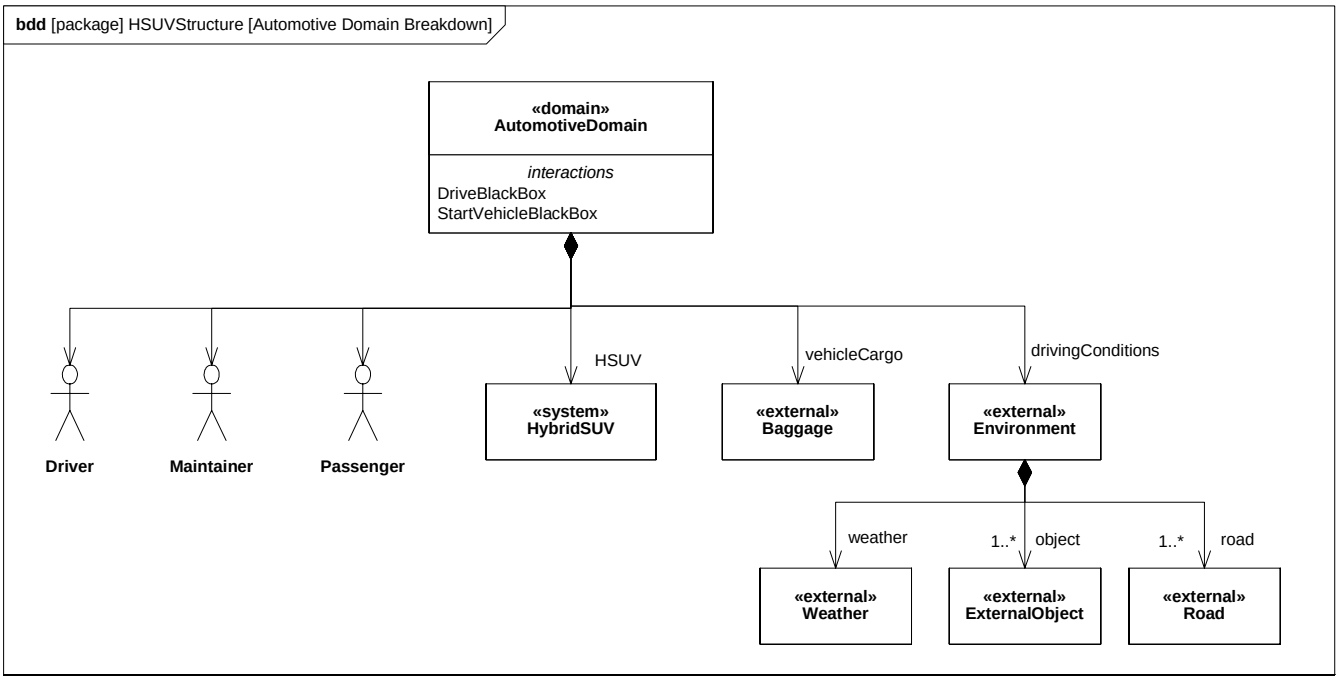


Figure C.15 - Defining the Automotive Domain (compare with Figure C.4) - (Block Definition Diagram)

C.4.5.2 Block Definition Diagram - Hybrid SUV

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

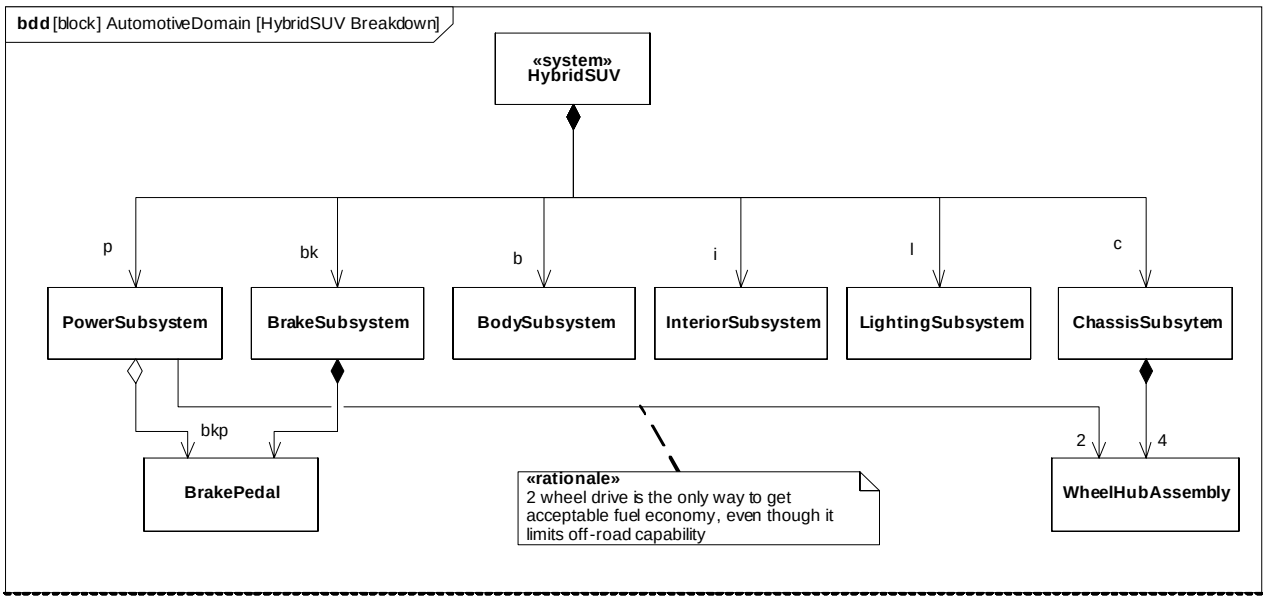


Figure C.16 - Defining Structure of the Hybrid SUV System (Block Definition Diagram)

C.4.5.3 Internal Block Diagram - Hybrid SUV

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

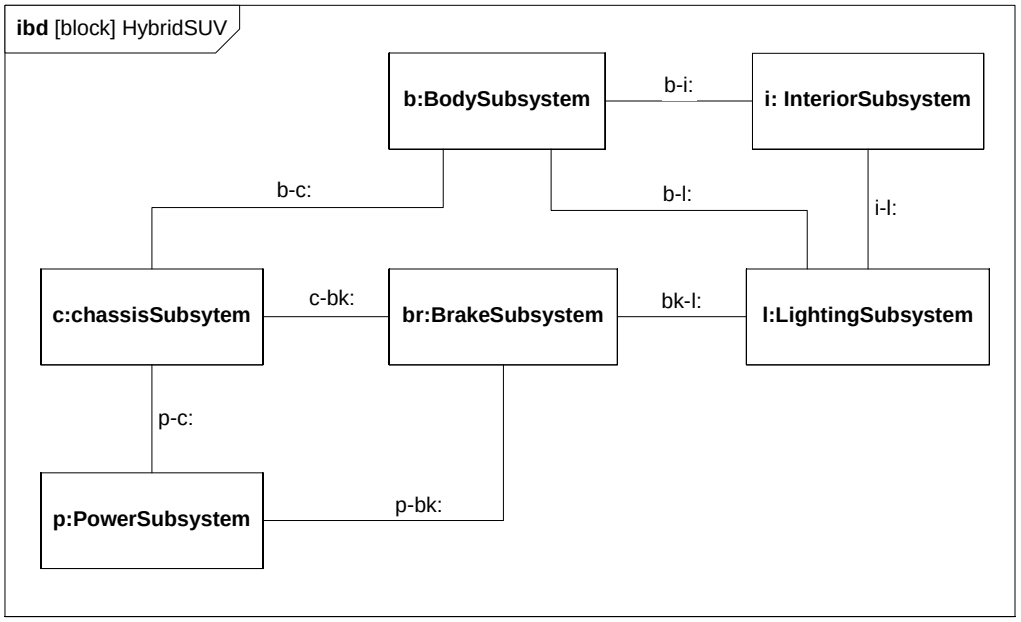


Figure C.17 - Internal Structure of Hybrid SUV (Internal Block Diagram)

C.4.5.4 Block Definition Diagram - Power Subsystem

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

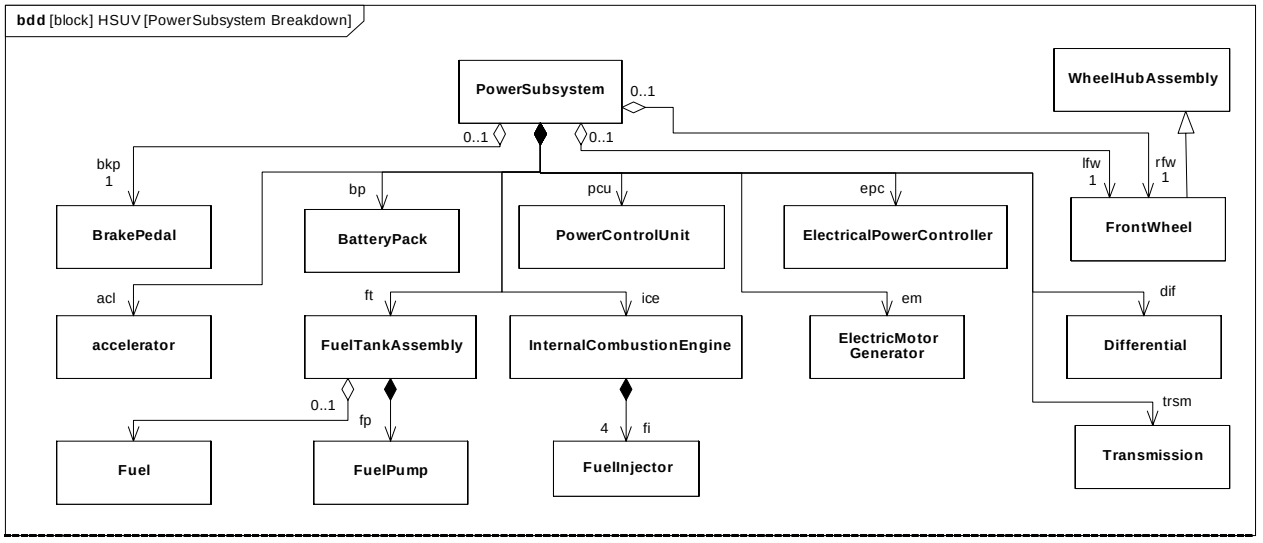


Figure C.18 - Defining Structure of Power Subsystem (Block Definition Diagram)

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

Figure C.19 shows how the parts of the PowerSubsystem block, as defined in the diagram above, are used. It shows connectors between parts, ports, and connectors with item flows. The dashed borders on FrontWheel and BrakePedal denote the “use-not-composition” relationship depicted elsewhere in Figure C.16 and Figure C.18. The dashed borders on Fuel denote a store, which keeps track of the amount and mass of fuel in the FuelTankAssy. This is also depicted in Figure C.18.

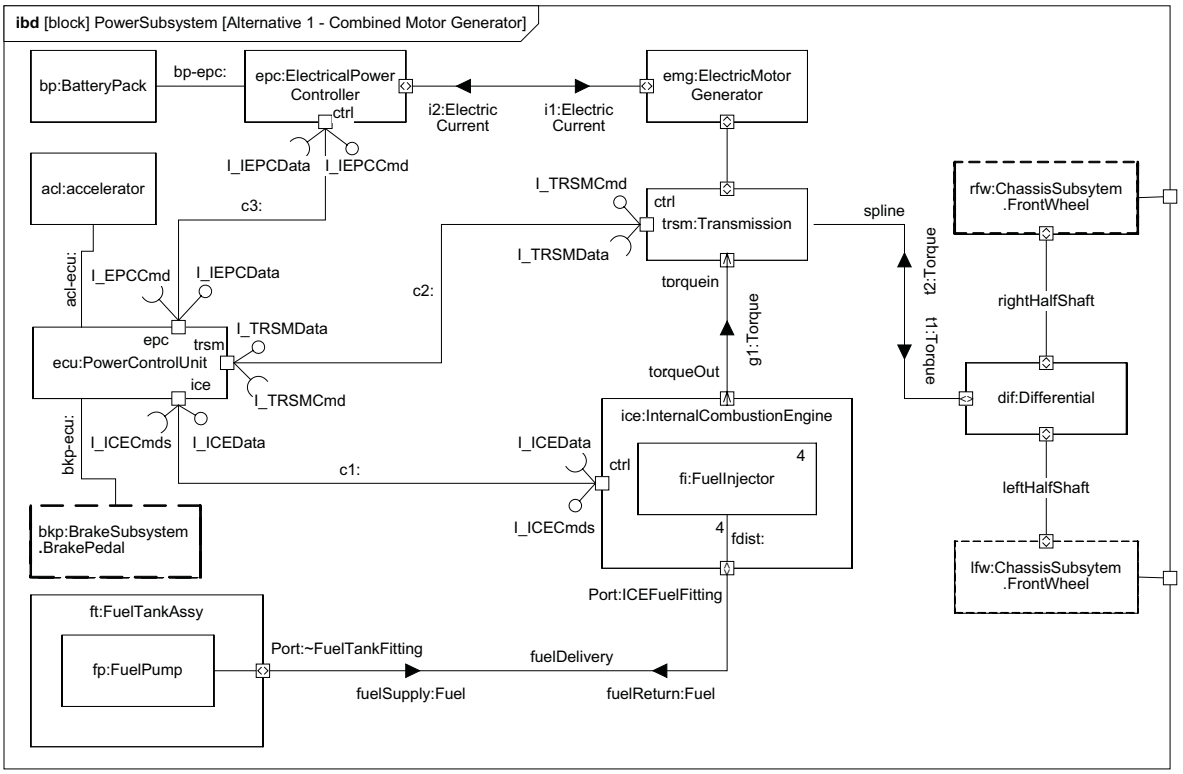


Figure C.19 - Internal Structure of the Power Subsystem (Internal Block Diagram)

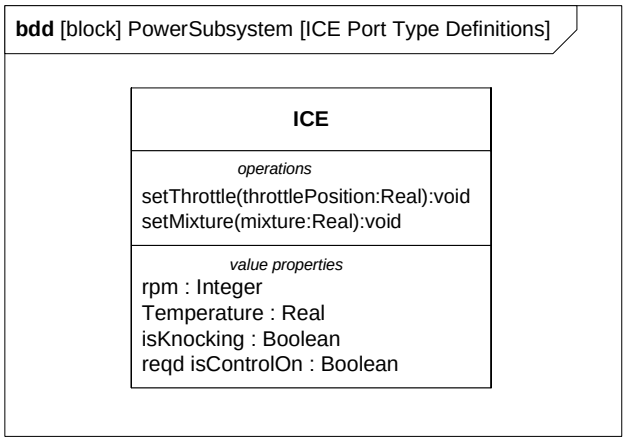


Figure C.20 - Blocks Typing Ports in the Power Subsystem (Block Definition Diagram)

Figure C.20 provides definition of the block that types the ports linked by connector c1 in Figure C.19.

C.4.6 Defining Ports and Flows

C.4.6.1 Block Definition Diagram - ICE Flow Properties

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

.

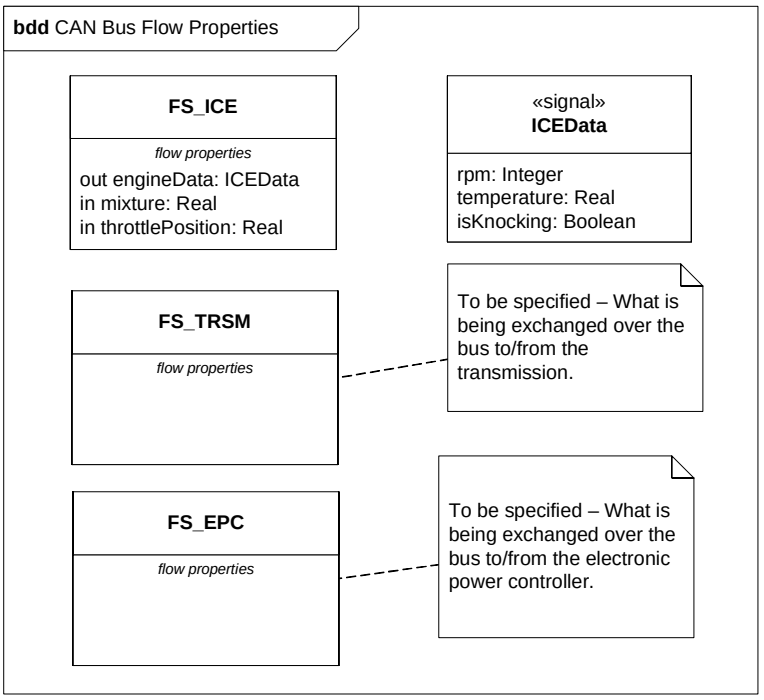


Figure C.21 - Initially Defining Port Types with Flow Properties for the CAN Bus (Block Definition Diagram)

C.4.6.2 Internal Block Diagram - CANbus

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

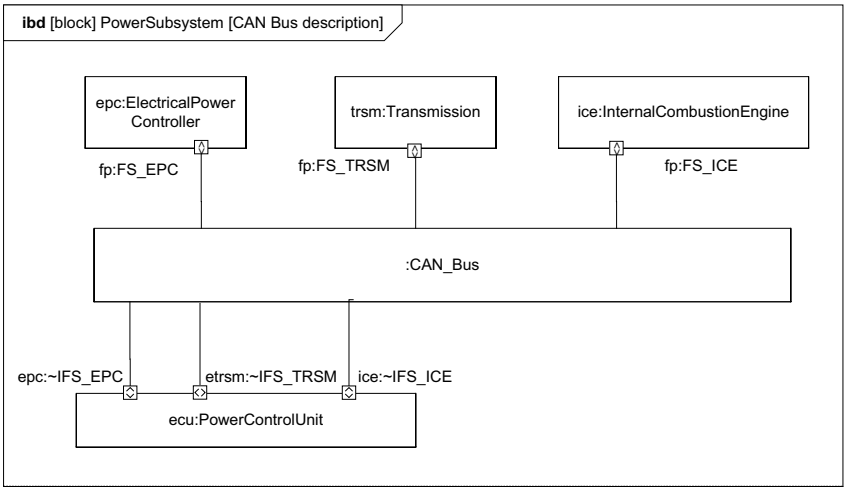


Figure C.22 - Consolidating Connectors into the CAN Bus. (Internal Block Diagram)

C.4.6.3 Block Definition Diagram - Fuel Flow Properties

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

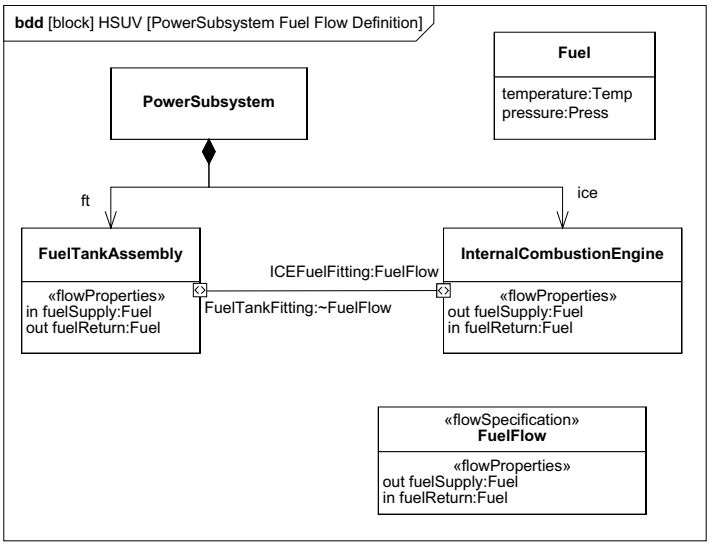


Figure C.23 - Elaborating Definition of Fuel Flow. (Block Definition Diagram)

C.4.6.4 Parametric Diagram - Fuel Flow

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

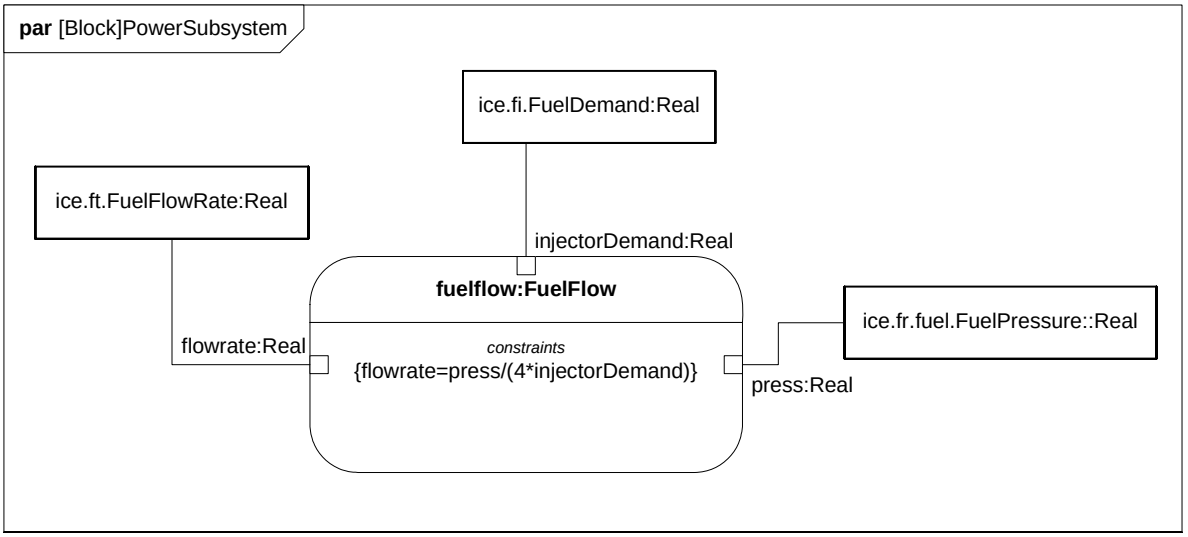


Figure C.24 - Defining Fuel Flow Constraints (Parametric Diagram)

C.4.6.5 Internal Block Diagram - Fuel Distribution

Figure C.25 shows how the connectors fuelDelivery and fdist on Figure C.19 have been expanded to include design detail. The fuelDelivery connector is actually two connectors, one carrying fuelSupply and the other carrying fuelReturn. The fdist connector inside the InternalCombustionEngine block has been expanded into the fuel regulator and fuel rail parts. These more detailed design elements are related to the original connectors using the allocation relationship. The Fuel store represents a quantity of fuel in the FuelTankAssy, which is drawn by the FuelPump for use in the engine, and is refreshed, to some degree, by fuel returning to the FuelTankAssy via the FuelReturnLine.

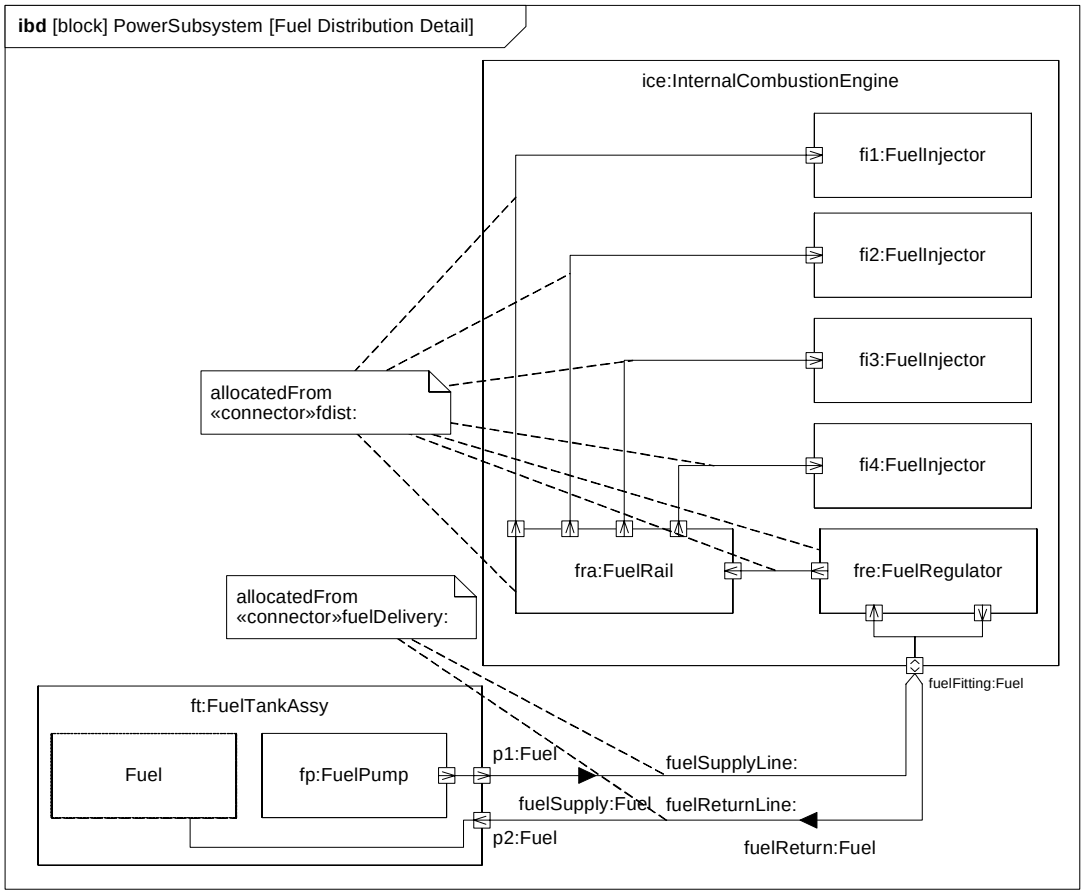


Figure C.25 - Detailed Internal Structure of Fuel Delivery Subsystem (Internal Block Diagram)

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

C.4.7.1 Block Definition Diagram - Analysis Context

Figure C.26 defines the various model elements that will be used to conduct analysis in this example. It depicts each of the constraint blocks/equations that will be used for the analysis, and key relationships between them.

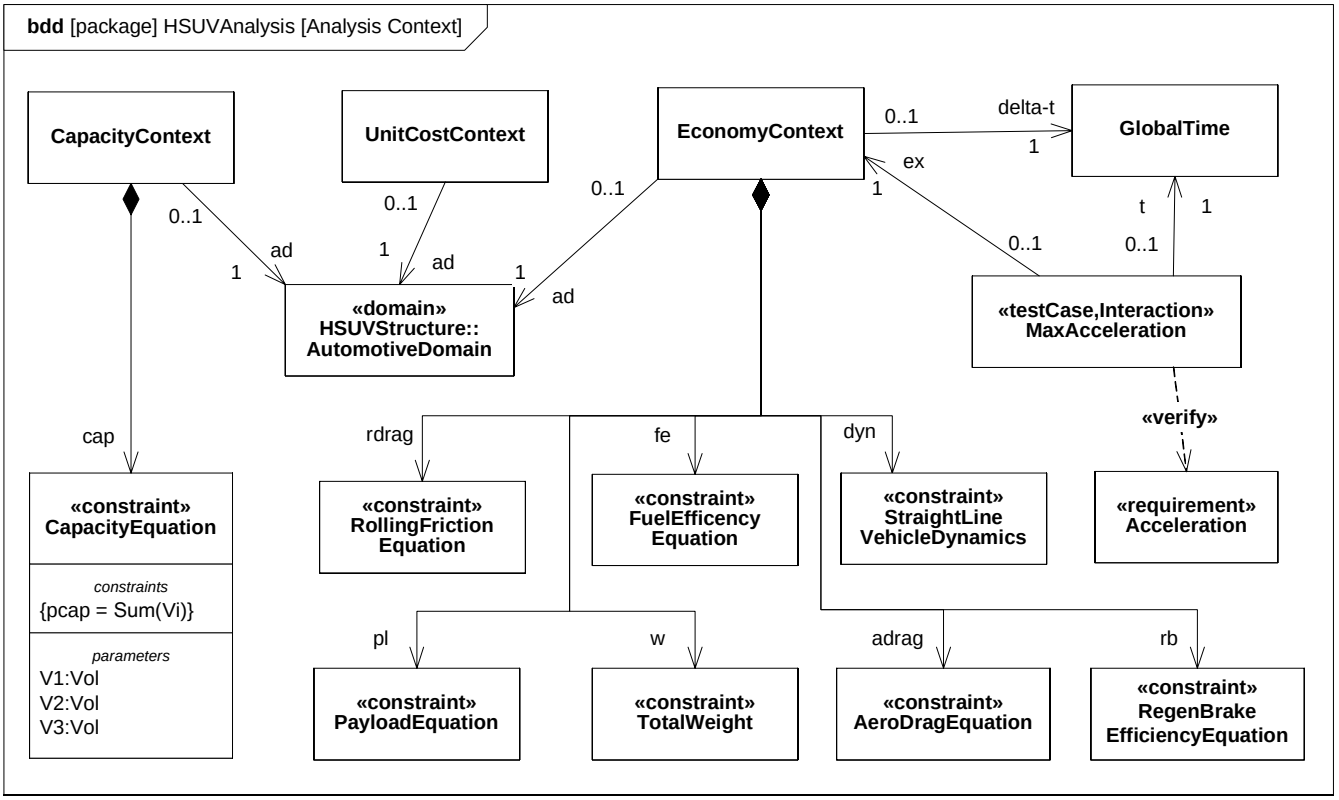


Figure C.26 - Defining Analyses for Hybrid SUV Engineering Development (Block Definition Diagram)

C.4.7.2 Package Diagram - Performance View Definition

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

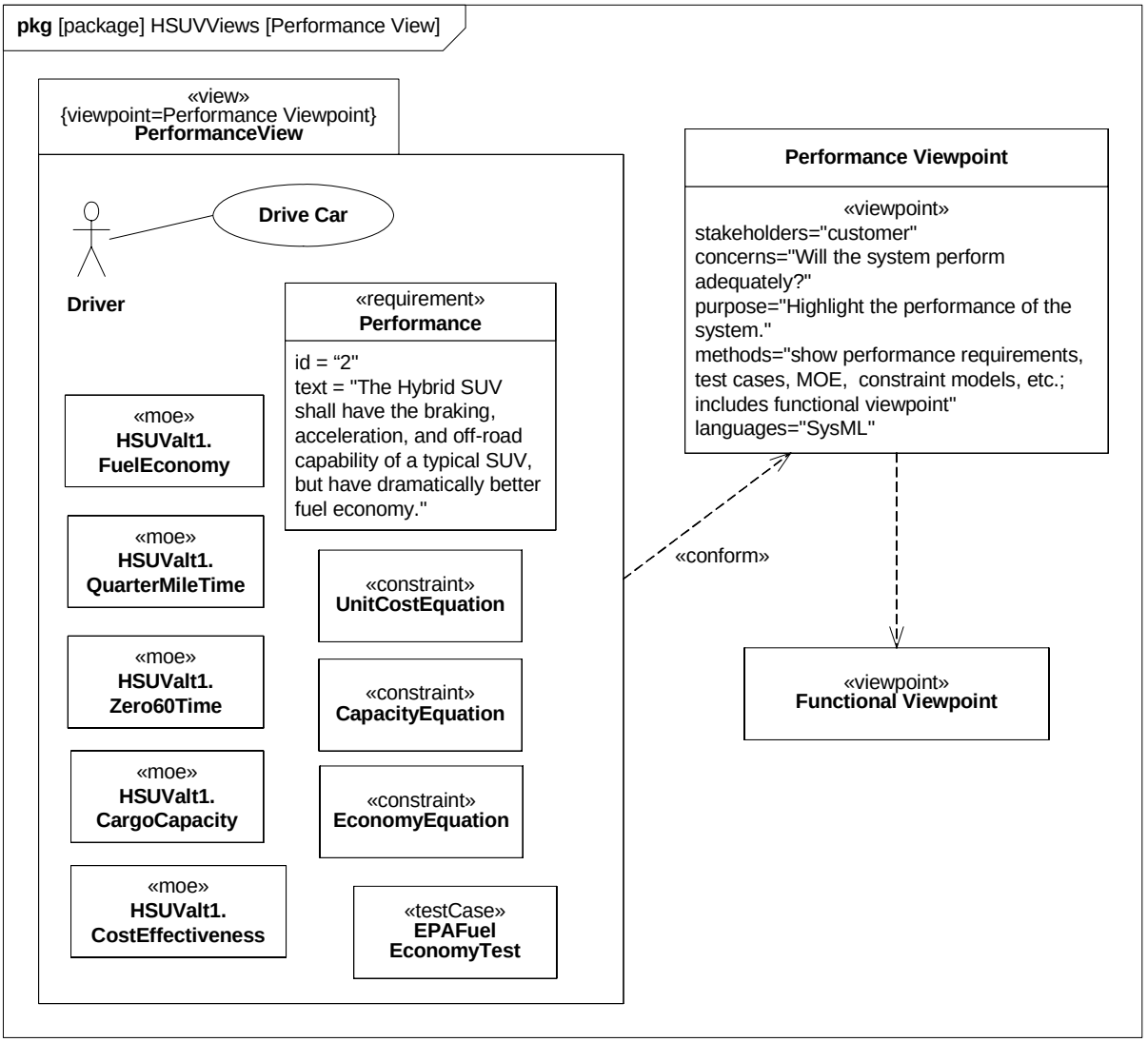


Figure C.27 - Establishing a Performance View of the User Model (Package Diagram)

C.4.7.3 Parametric Diagram - Measures of Effectiveness

Measure of Effectiveness is a user defined stereotype. Figure C.28 shows how the overall cost effectiveness of the HSUV will be evaluated. It shows the particular measures of effectiveness for one particular alternative for the HSUV design, and can be reused to evaluate other alternatives.

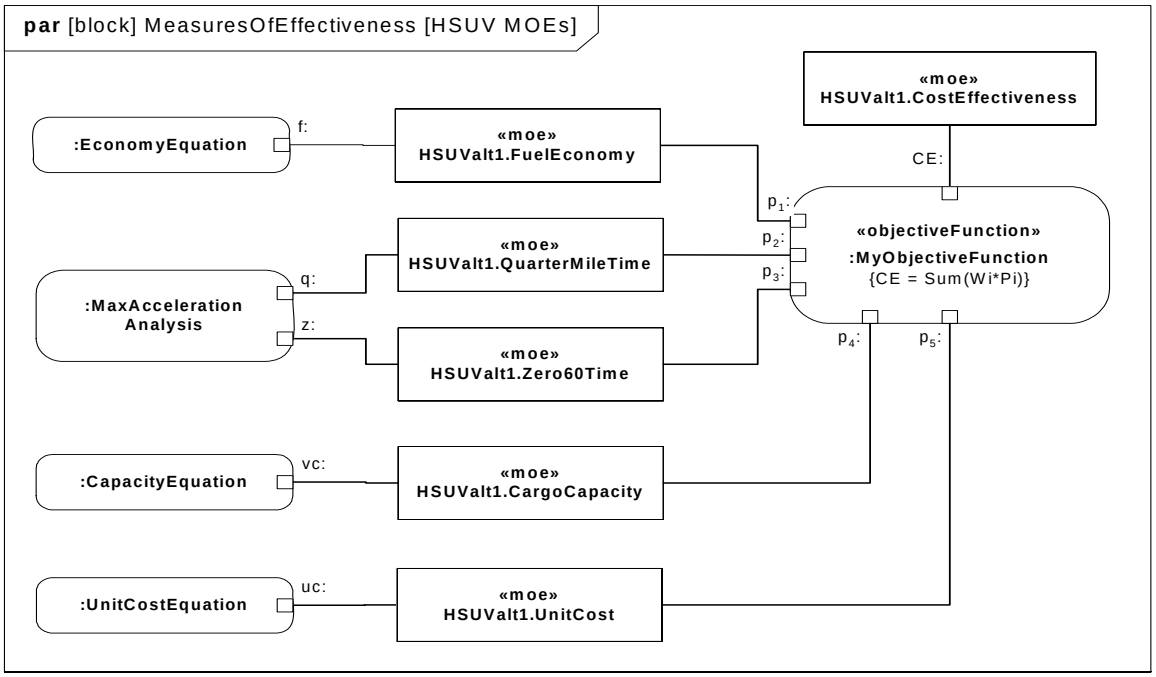


Figure C.28 - Defining Measures of Effectiveness and Key Relationships (Parametric Diagram)

C.4.7.4 Parametric Diagram - Economy

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

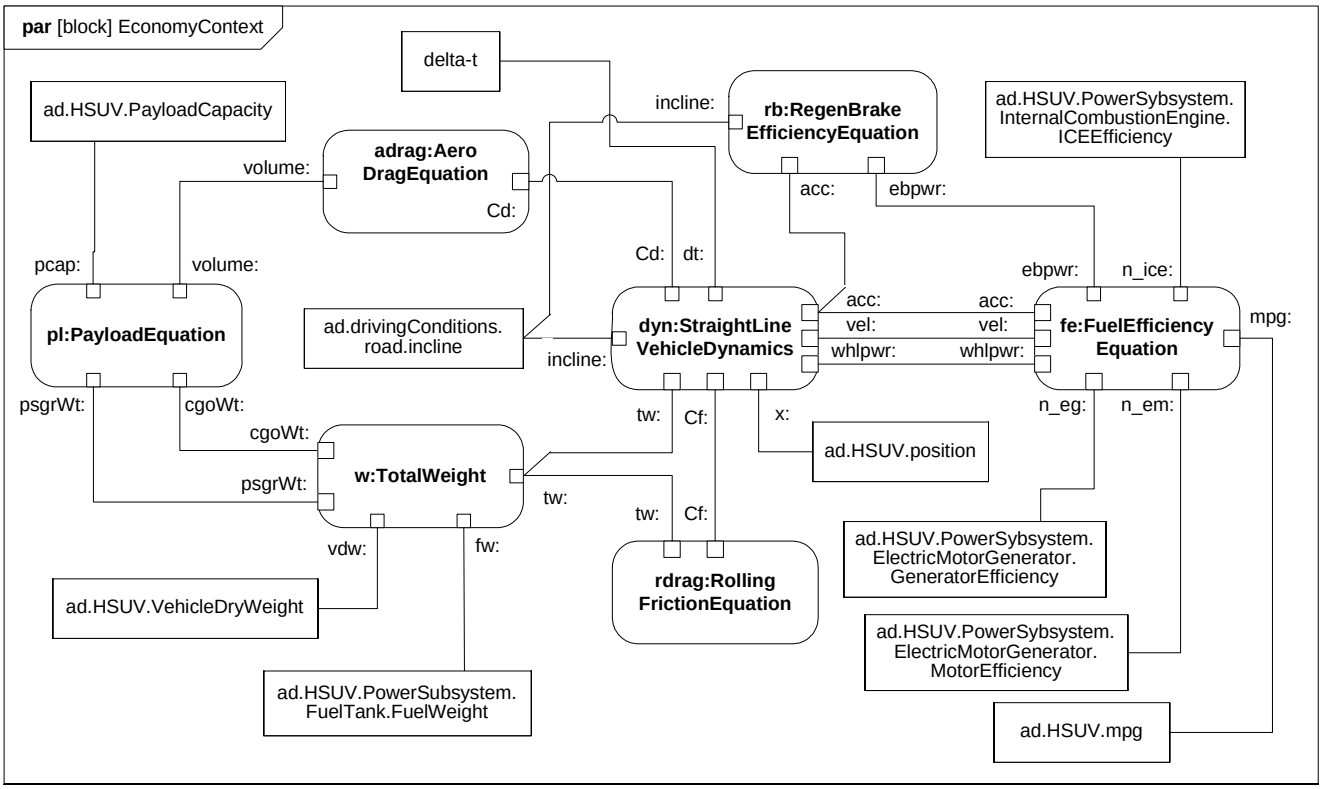


Figure C.29 - Establishing Mathematical Relationships for Fuel Economy Calculations (Parametric Diagram)

C.4.7.5 Parametric Diagram - Dynamics

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

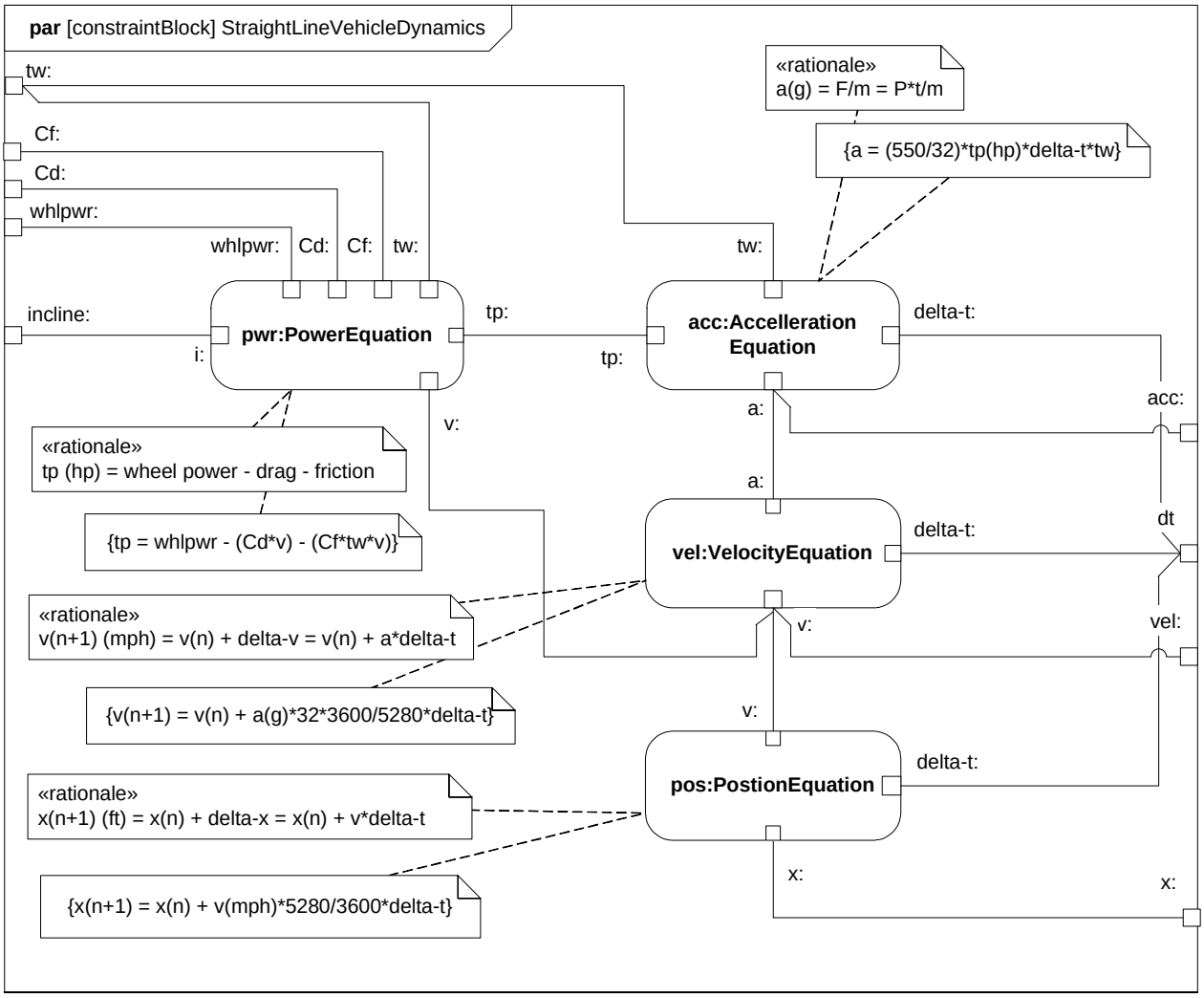


Figure C.30 - Straight Line Vehicle Dynamics Mathematical Model (Parametric Diagram)

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

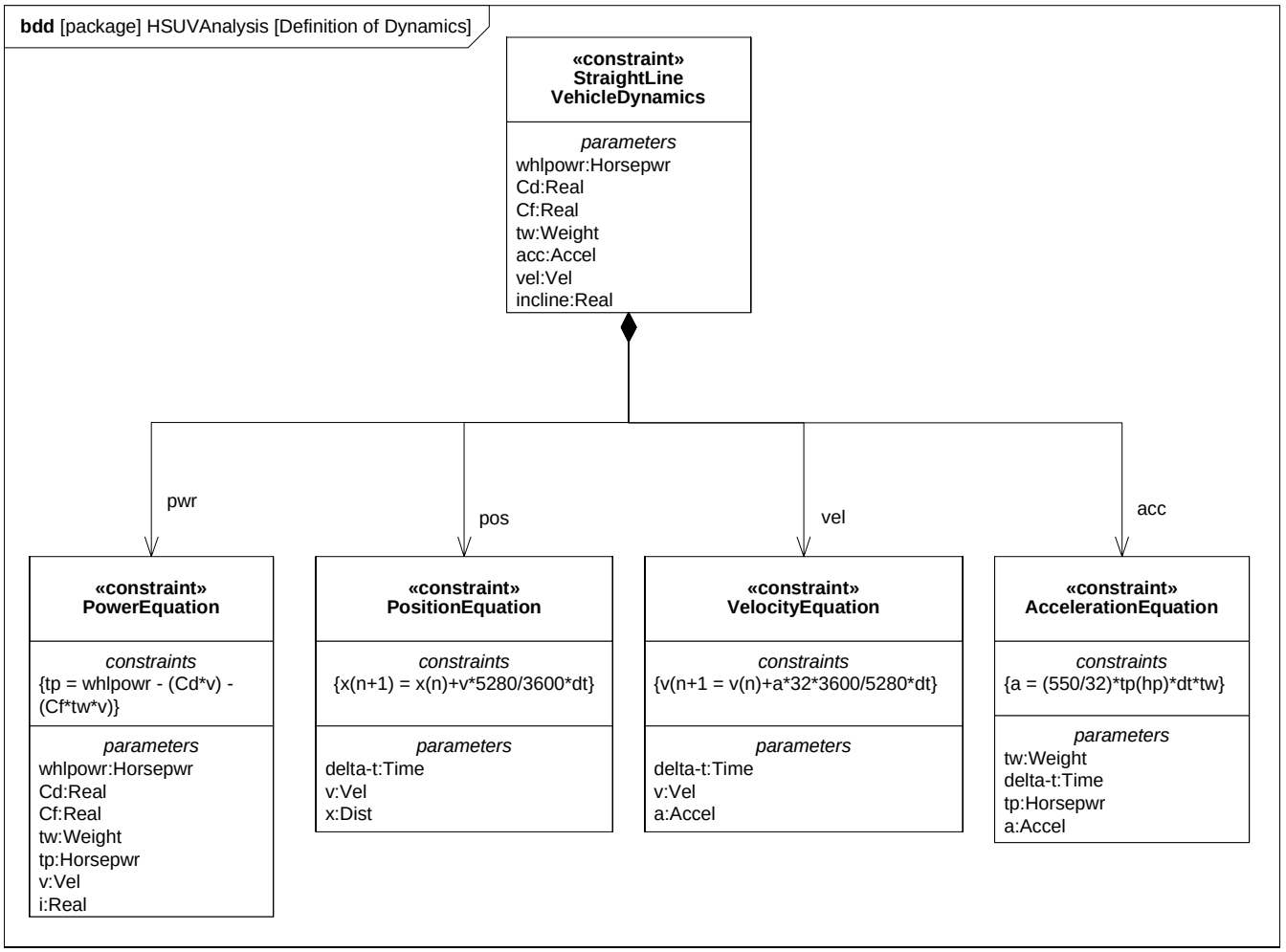


Figure C.31 - Defining Straight-Line Vehicle Dynamics Mathematical Constraints (Block Definition Diagram)

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

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

Timing diagrams, while included in UML 2, are not directly supported by SysML. For illustration purposes, however, the interaction shown in Figure C.32 was generated based on the constraints and parameters of the StraightLineVehicleDynamics constraintBlock, as described in the Figure C.30. It assumes a constant 100hp at the drive wheels, 4000lb gross vehicle weight, and constant values for Cd and Cf.

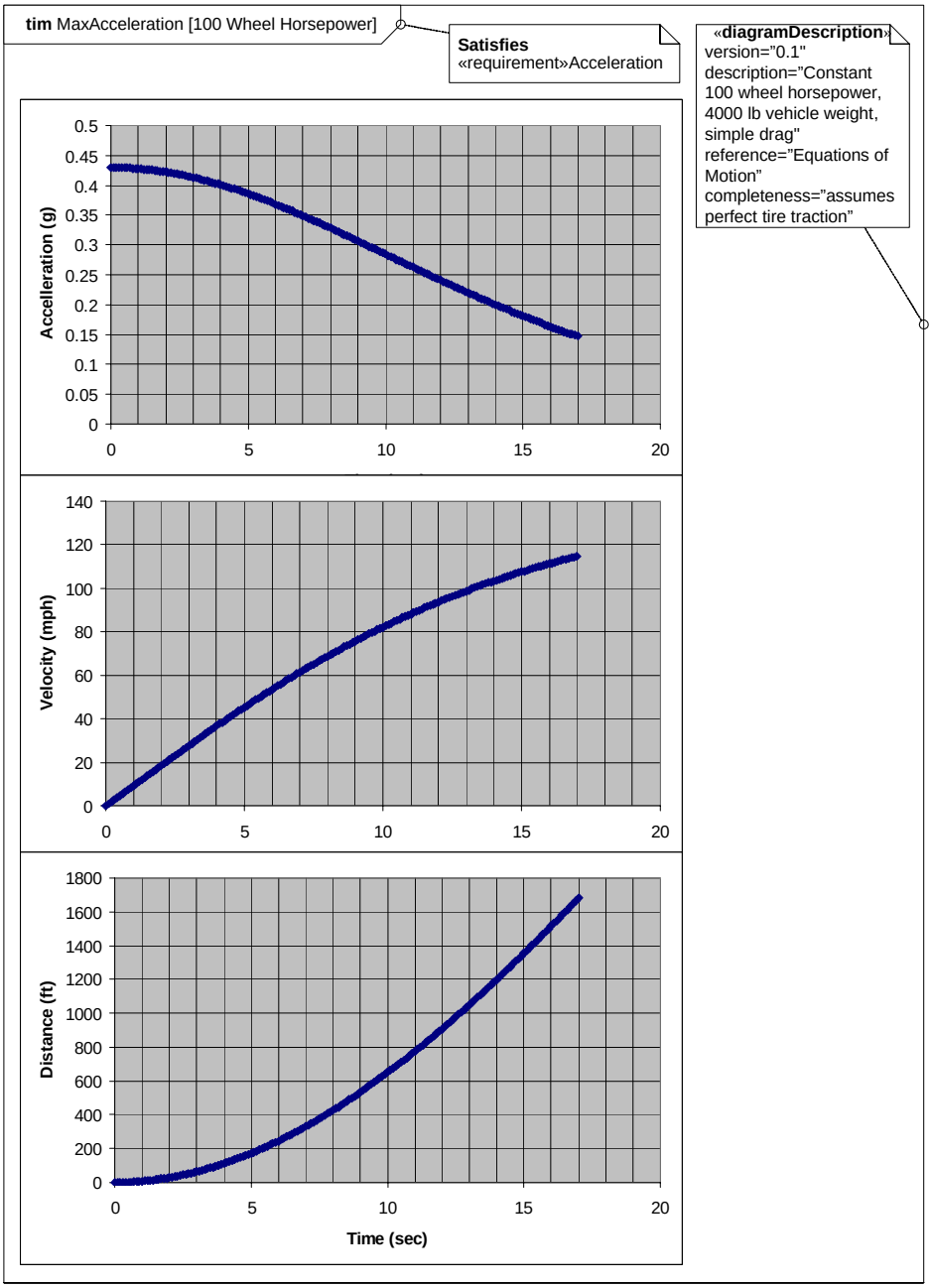


Figure C.32 - Results of Maximum Acceleration Analysis (Timing Diagram)

C.4.8 Defining, Decomposing, and Allocating Activities

C.4.8.1 Activity Diagram - Acceleration (top level)

Figure C.33 shows the top level behavior of an activity representing acceleration of the HSUV. It is the intent of the systems engineer in this example to allocate this behavior to parts of the PowerSubsystem. It is quickly found, however, that the behavior as depicted cannot be allocated, and must be further decomposed.

.

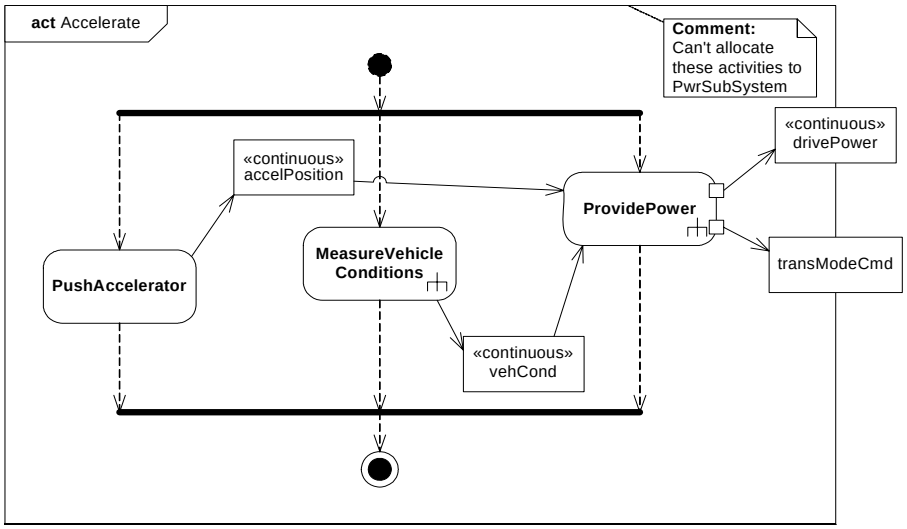


Figure C.33 - Behavior Model for "Accelerate" Function (Activity Diagram)

C.4.8.2 Block Definition Diagram - Acceleration

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

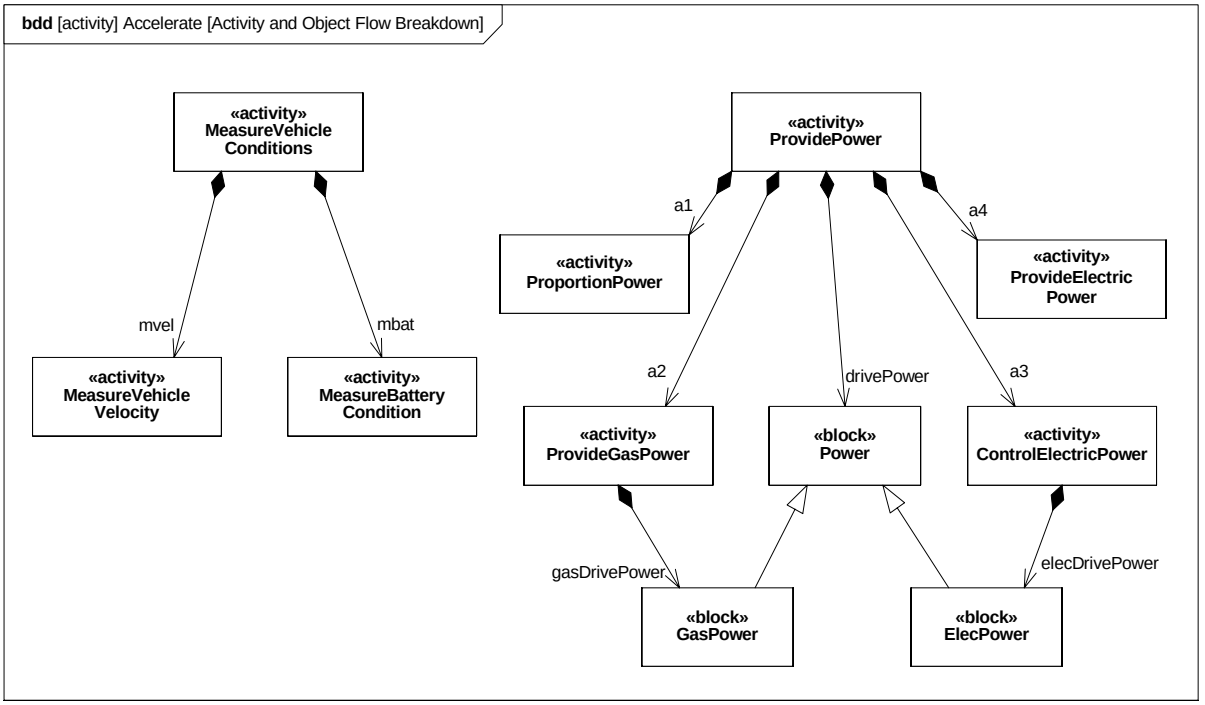


Figure C.34 - Decomposition of "Accelerate" Function (Block Definition diagram)

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

Figure C.35 shows the ProvidePower activity, which includes Actions invoking the decomposed Activities and ObjectNodes from Figure C.34. It also uses AllocateActivityPartitions and an allocation callout to explicitly allocate activities and an object flow to parts in the PowerSubsystem block.

Note that the incoming and outgoing object flows for the ProvidePower activity have been decomposed. This was done to distinguish the flow of electrically generated mechanical power and gas generated mechanical power, and to provide further insight into the specific vehicle conditions being monitored.

.

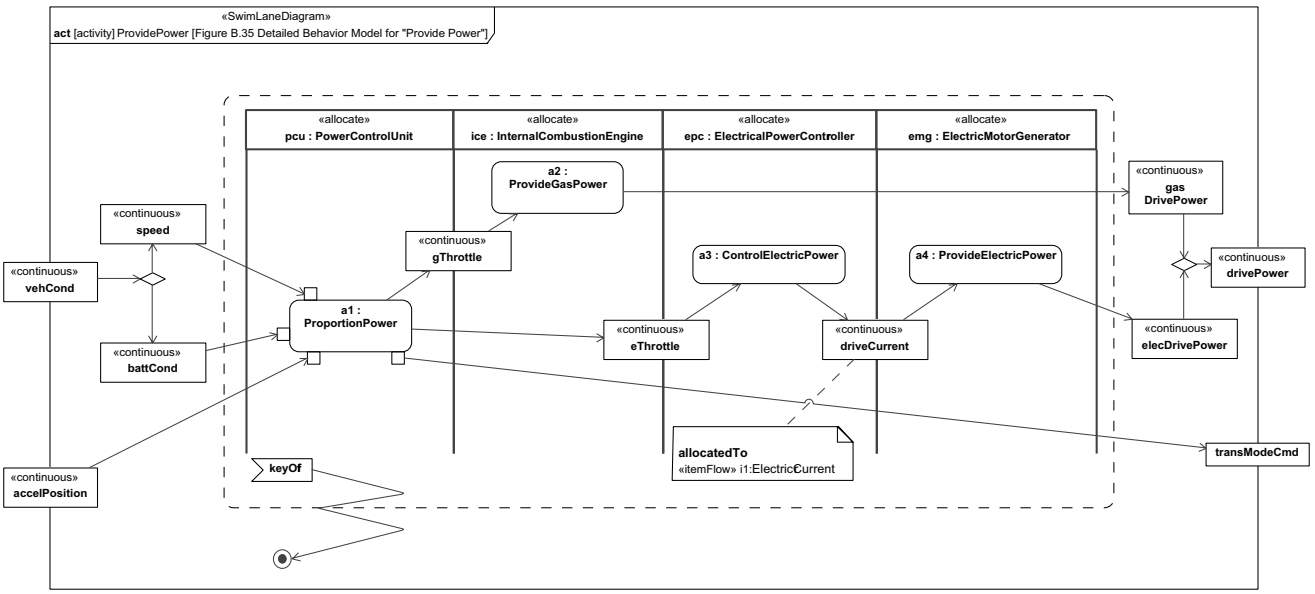


Figure C.35 - Detailed Behavior Model for “Provide Power” (Activity Diagram)
Note hierarchical consistency with Figure C.33.

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

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

.

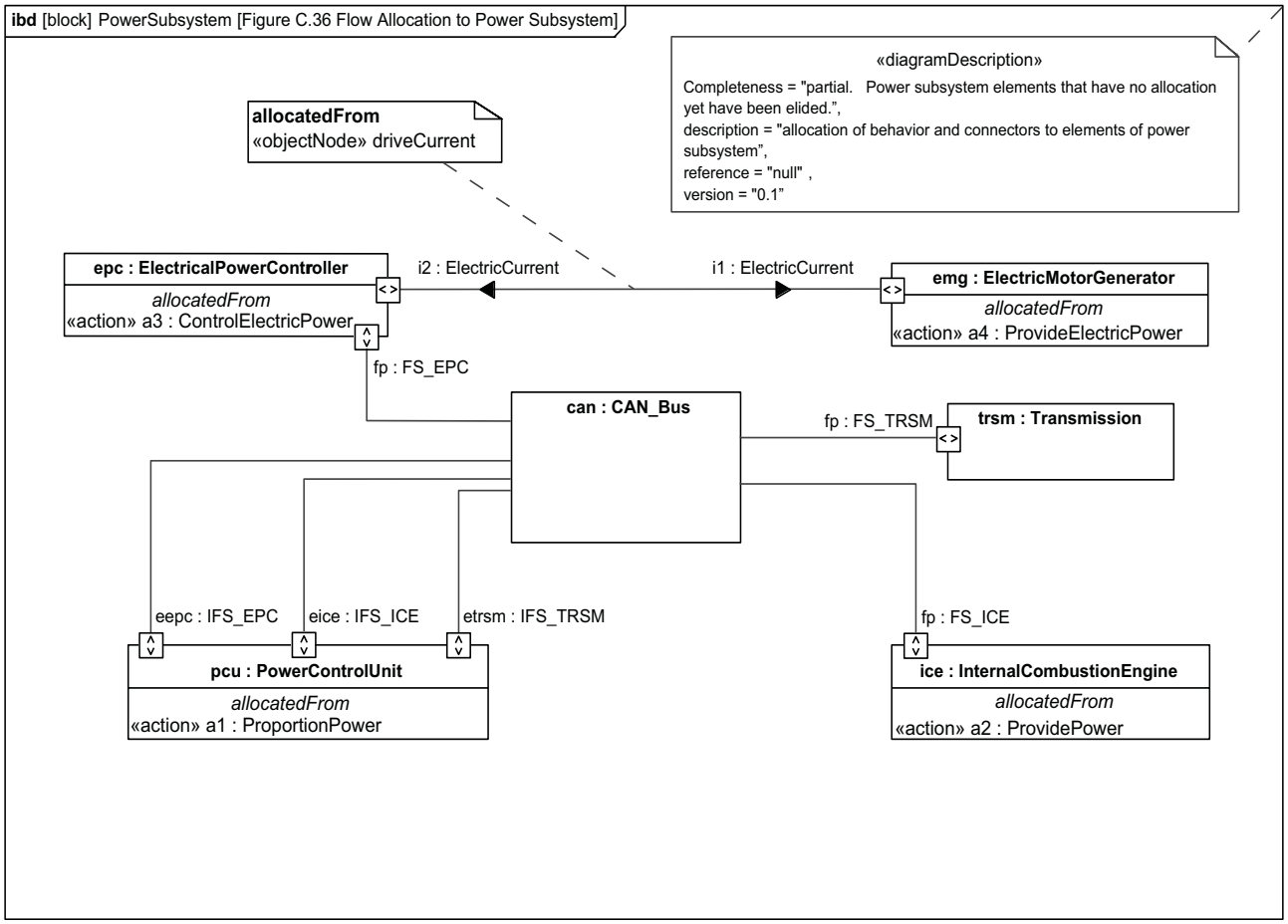


Figure C.36 - Flow Allocation to Power Subsystem (Internal Block Diagram)

C.4.8.5 Table - Acceleration Allocation

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

bdd [package] HSUV Behavior [Figure B.37 Tabular Representation of Allocation from"Accelerate" Behavior Model to Power Subsystem]

Type	Name	End	Relation	End	Type	Name
action	a1 : ProportionPower	from	allocate	to	part	ecu : PowerControlUnit
action	a2 : ProvideGasPower	from	allocate	to	part	ice : InternalCombustionEngine
action	a3 : ControlElectricPower	from	allocate	to	part	epc : ElectricPowerController
action	a4 : ProvideElectricPower	from	allocate	to	part	emg : ElectricMotorGenerator
objectFlow	o6	from	allocate	to	connector	epc-emg.1

Figure C.37 - Tabular Representation of Allocation from “Accelerate” Behavior Model to Power Subsystem (Table)

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

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

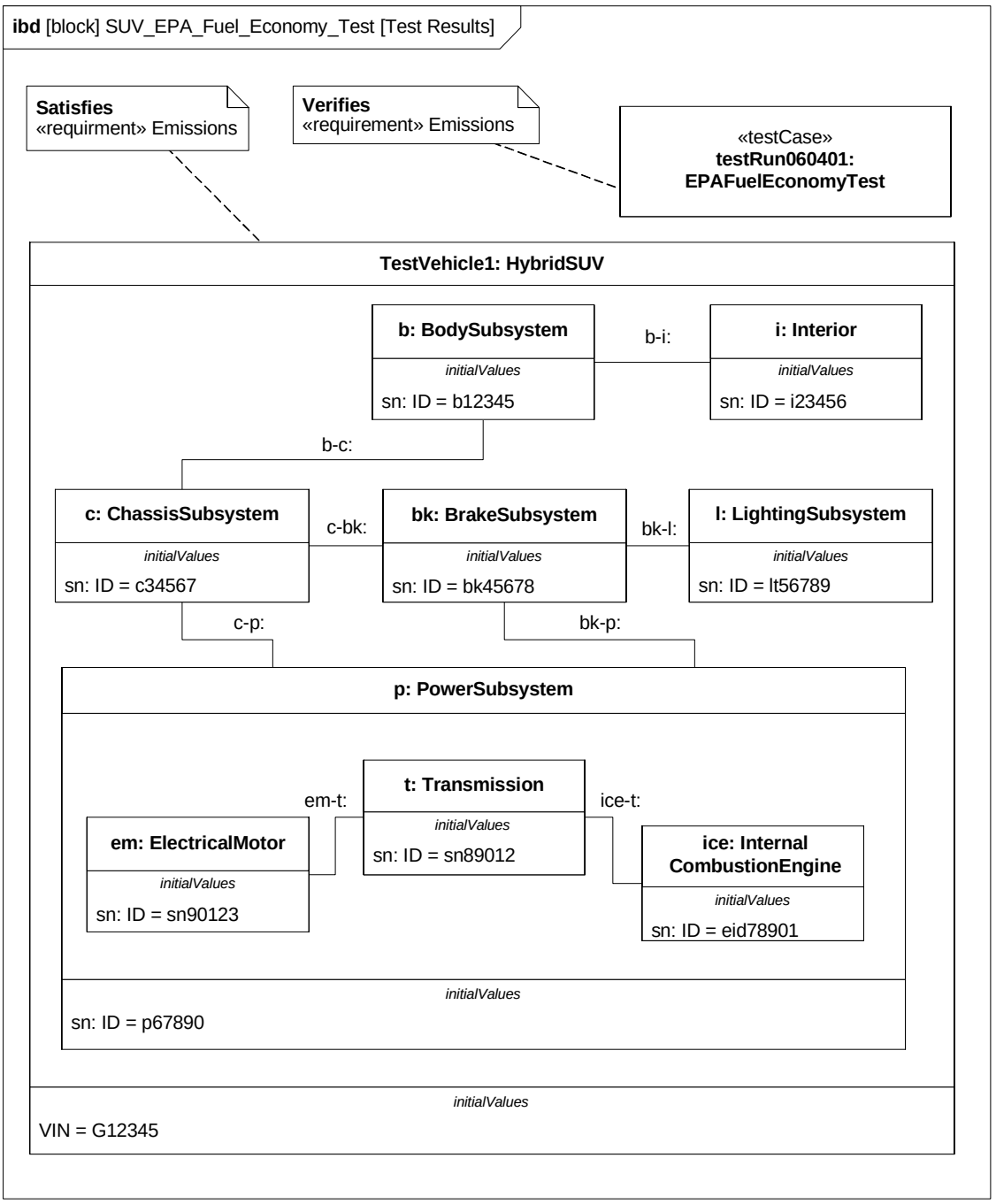


Figure C.38 - Special Case of Internal Block Diagram Showing Reference to Specific Properties (serial numbers)

Annex D: Non-normative Extensions

(informative)

This annex describes useful non-normative extensions to SysML that may be considered for standardization in future versions of the language.

Non-normative extensions consist of stereotypes and model libraries and are organized by major diagram type, consistent with how the main body of this specification is organized. Stereotypes in this sub clause are specified using a tabular format, consistent with how non-normative stereotypes are specified in the UML 2 Superstructure specification. Model libraries are specified using the guidelines provided in the Profiles & Model Libraries clause of this specification.

D.1 Activity Diagram Extensions

D.1.1 Overview

Two non-normative extensions to activities are described for:

- Enhanced Functional Flow Block Diagrams.
- Streaming activities that accept inputs and/or provide outputs while they are active.

More information on these extensions and the standard SysML extensions is available at [Bock. C., "SysML and UML 2.0 Support for Activity Modeling," vol. 9, no. 2, pp. 160-186, *Journal of the International Council of Systems Engineering*, 2006].

D.1.2 Stereotypes

Enhanced Functional Flow Block Diagrams (EFFBD) are a widely-used systems engineering diagram, also called a behavior diagram. Most of its functionality is a constrained use of UML activities, as described below. This extension does not address replication, resources, or kill branches. Kill branches can be translated to activities using interruptible regions and join specifications.

Table D.1 - Addition stereotypes for EFFBDs

Stereotype	Base class	Properties	Constraints	Description
«effbd»	UML4SysML::Activity (or subtype of «nonStreaming» below)	N/A	See below.	Specifies that the activity conforms to the constraints necessary for EFFBD.

When the «effbd» stereotype is applied to an activity, its contents must conform to the following constraints:

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

[2] (On Activity) All decisions, merges, joins, and forks are well-nested. In particular, each decision and merge are matched one-to-one, as are forks and joins, accounting for the output parameter sets acting as decisions, and input parameters and control acting as a join.

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

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

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

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

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

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

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

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

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

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

[13] (On ParameterSet) Parameter sets have exactly one parameter, and it must not be shared with other parameter sets.\

[14] (On ParameterSet) If one output parameter is in a parameter set, then all output parameters of the behavior or operation must be in parameter sets.

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

[16] The following SysML stereotypes cannot be applied: «rate», «controlOperator», «noBuffer», «overwrite».

A second extension distinguishes activities based on whether they can accept inputs or provide outputs after they start and before they finish (streaming), or only accept inputs when they start and provide outputs when they are finished (nonstreaming). EFFBD activities are nonstreaming. Streaming activities are often terminated by other activities, while nonstreaming activities usually terminate themselves.

Table D.2 - Streaming options for activities

Stereotype	Base Class	Properties	Constraints	Description
«streaming»	UML4SysML::Activity	N/A	The activity has at least one streaming parameter.	Used for activities that can accept inputs or provide outputs after they start and before they finish.
«nonStreaming»	UML4SysML::Activity	N/A	The activity has no streaming parameters.	Used for activities that accept inputs only when they start, and provide outputs only when they finish.

D.1.3 Stereotype Examples

Figure D.1 shows an example activity diagram with the «effbd» stereotype applied, translated from [Long. J., “Relationships between common graphical representations in system engineering,” 2002]. The stereotype applies the constraints specified in Section D.1.2, for example, that the data outputs on all functions are required and that queuing is FIF.

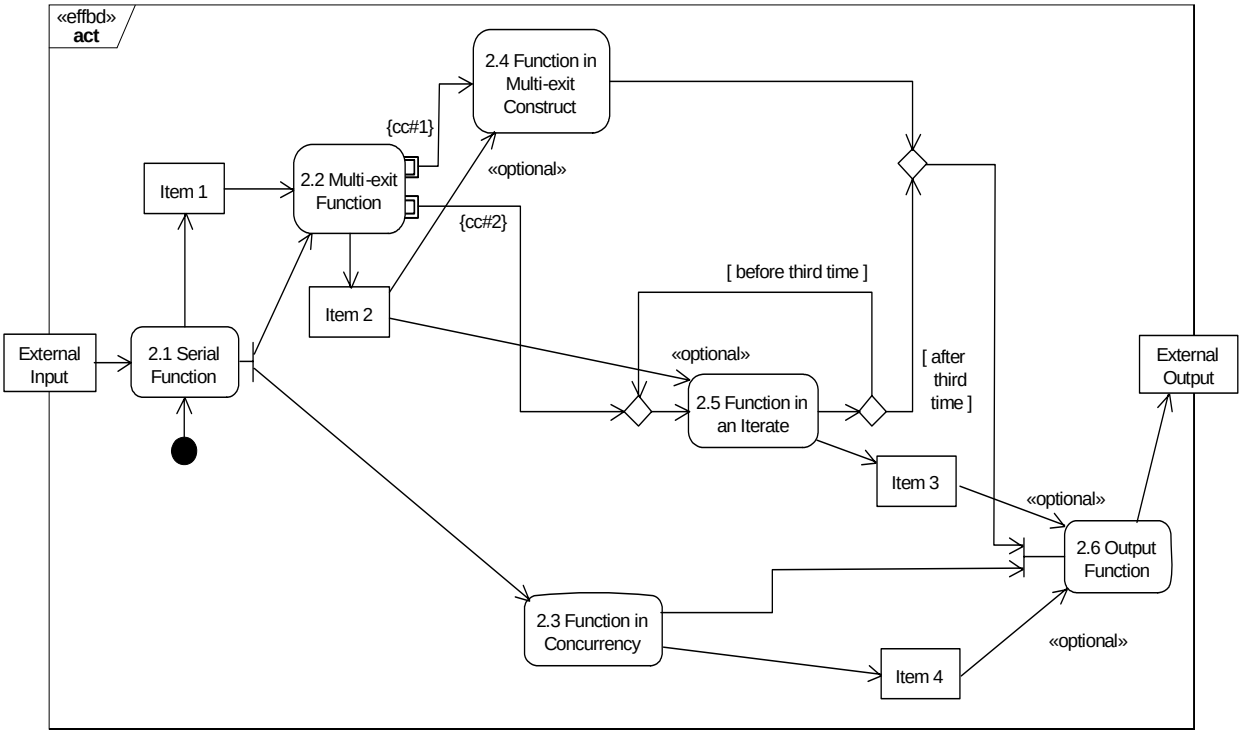


Figure D.1- Example activity with «effbd» stereotype applied

Figure D.2 shows an example activity diagram with the «streaming» and «nonStreaming» stereotypes applied, adapted from [MathWorks, "Using Simulink," 2004]. It is a numerical solution for the differential equation $x'(t) = -2x(t) + u(t)$. Item types are omitted brevity. The «streaming» and «nonStreaming» stereotypes indicate which subactivities take inputs and produce outputs while they are executing. They are simpler to use than the {stream} notation on streaming inputs and outputs.

The example assumes a default of zero for the lower input to Add, and that the entire activity is executed with clocked token flow, to ensure that actions with multiple inputs receive as many of them as possible before proceeding. See the article referenced in Section D.1.1.

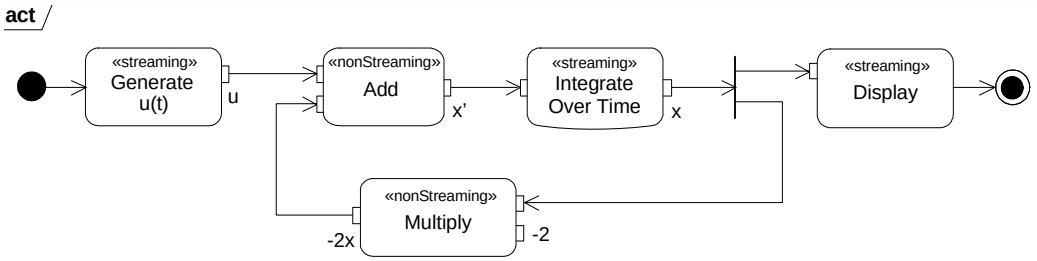


Figure D.2- Example activity with «streaming» and «nonStreaming» stereotypes applied to subactivities

D.2 Requirements Diagram Extensions

D.2.1 Overview

This sub clause describes an example of a non-normative extension for a requirements profile.

D.2.2 Stereotypes

This non-normative extension includes stereotypes for a simplified requirements taxonomy that is intended to be further adapted as required to support the particular needs of the application or organization. The requirements categories in this example include functional, interface, performance, physical requirements, and design constraints as shown in Table D.3. As shown in the table, each category is represented as a stereotype of the generic SysML «requirement». The table also includes a brief description of the category. The table does not include any stereotype properties or constraints, although they can be added as deemed appropriate for the application. For example, a constraint that could be applied to a functional requirement is that only SysML activities and operations can satisfy this category of requirement. Other examples of requirements categories may include operational, specialized requirements for reliability and maintainability, store requirements, activation, deactivation, and a high level category for stakeholder needs.

Some general guidance for applying a requirements profile is as follows:

- The categories should be adapted for the specific application or organization and reflected in the table. This includes agreement on the categories and their associated descriptions, stereotype properties, and constraints. Additional categories can be added by further subclassing the categories in the table below, or adding additional categories at the pier level of these categories.
- The default requirement category should be the generic «requirement».
- Apply the more specialized requirement stereotype (functional, interface, performance, physical, design constraint) as applicable and ensure consistency with the description, stereotype properties, and constraints.
- A specific text requirement can include the application of more than one requirement category, in which case, each stereotype should be shown in guillemets.

Table D.3 - Additional Requirement Stereotypes

Stereotype	Base Class	Properties	Constraints	Description
«extendedRequirement»	«requirement»	source: String risk: RiskKind verifyMethod: VerifyMethodKind	N/A	A mix-in stereotype that contains generally useful attributes for requirements
«functionalRequirement»	«extendedrequirement»	N/A	satisfied by an operation or behavior	Requirement that specifies an operation or behavior that a system, or part of a system, must perform.
«interfaceRequirement»	«extendedrequirement»	N/A	satisfied by a port, connector, item flow, and/ or constraint property	Requirement that specifies the ports for connecting systems and system parts and the optionally may include the item flows across the connector and/or Interface constraints.
«performanceRequirement»	«extendedrequirement»	N/A	satisfied by a value property	Requirement that quantitatively measures the extent to which a system, or a system part, satisfies a required capability or condition.

Table D.3 - Additional Requirement Stereotypes

Stereotype	Base Class	Properties	Constraints	Description
«physicalRequirement»	«extendedrequirement»	N/A	satisfied by a structural element.	Requirement that specifies physical characteristics and/or physical constraints of the system, or a system part.
«designConstraint»	«extendedrequirement»	N/A	satisfied by a block or part	Requirement that specifies a constraint on the implementation of the system or system part, such as the system must use a commercial off the shelf component.

Table D.4 provides the definition of the non-normative enumerations that are used to type properties of "extendedRequirement" stereotype of Figure D.3.

Table D.4 - Requirement property enumeration types

Enumeration	Enumeration Literals	Example Description
RiskKind	High	High indicates an unacceptable level of risk
	Medium	Medium indicates an acceptable level of risk
	Low	Low indicates a minimal level of risk or no risk
VerificationMethodKind	Analysis	Analysis indicates that verification will be performed by technical evaluation using mathematical representations, charts, graphs, circuit diagrams, data reduction, or representative data. Analysis also includes the verification of requirements under conditions, which are simulated or modeled; where the results are derived from the analysis of the results produced by the model.
	Demonstration	Demonstration indicates that verification will be performed by operation, movement or adjustment of the item under specific conditions to perform the design functions without recording of quantitative data.. Demonstration is typically considered the least restrictive of the verification types.
	Inspection	Inspection indicates that verification will be performed by examination of the item, reviewing descriptive documentation, and comparing the appropriate characteristics with a predetermined standard to determine conformance to requirements without the use of special laboratory equipment or procedures.
	Test	Test indicates that verification will be performed through systematic exercising of the applicable item under appropriate conditions with instrumentation to measure required parameters and the collection, analysis, and evaluation of quantitative data to show that measured parameters equal or exceed specified requirements.

D.2.3 Stereotype Examples

Figure D.3 shows the use of several subtypes of requirements extended to include the properties risk:RiskKind, verifyMethod:VerficationMethodKind, and a text attribute source:String, used to capture the source of the requirement.

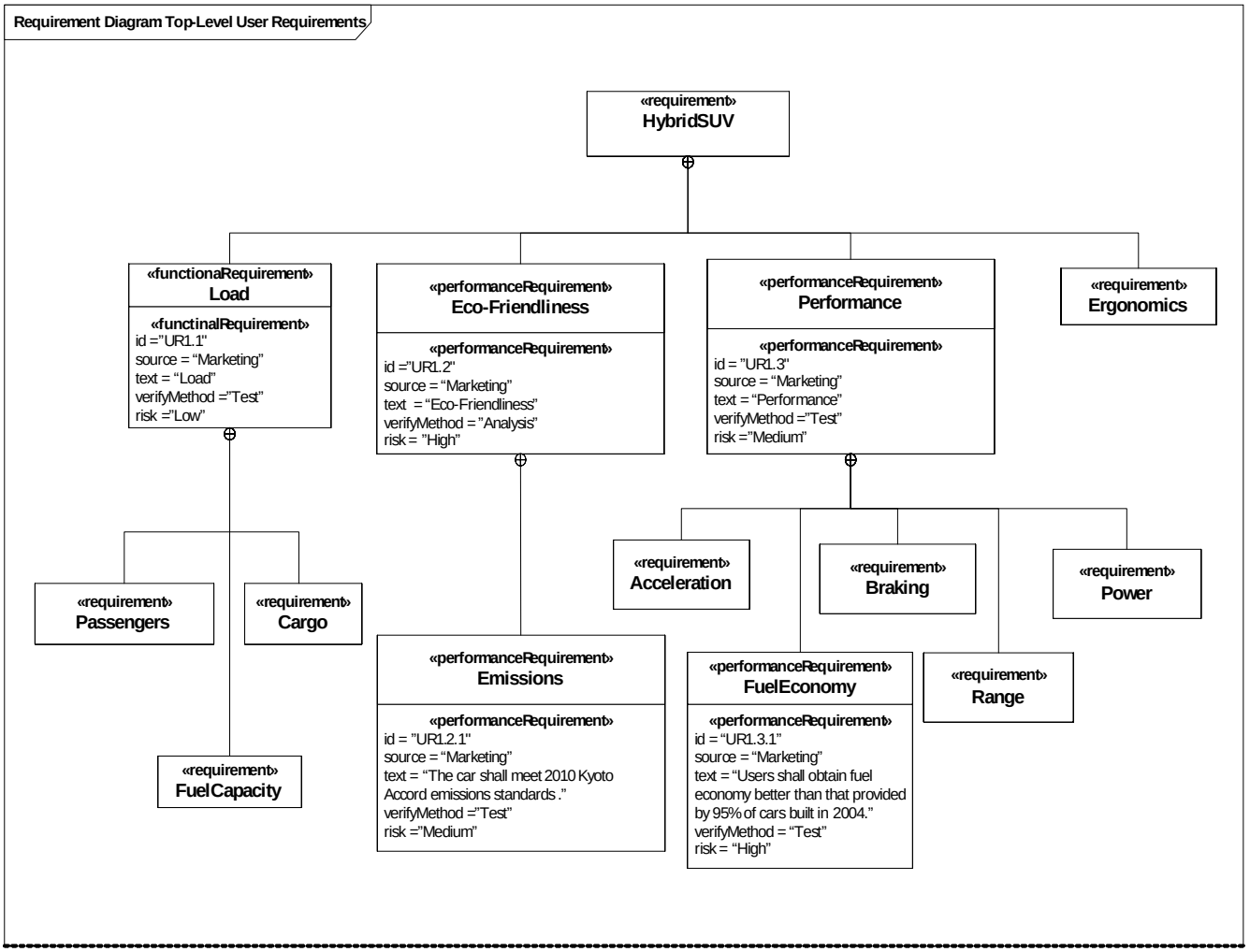


Figure D.3 - Example extensions to Requirement

D.3 Parametric Diagram Extensions for Trade Studies

D.3.1 Overview

This sub clause describes a non-normative extension of a parametric diagram (refer to the Constraint Blocks clause) to support trade studies and analysis, which are an essential aspect of any systems engineering effort. In particular, a trade study is used to evaluate a set of alternatives based on a defined set of criteria. The criteria may have a weighting to reflect their relative importance. An objective function (aka optimization or cost function) can be used to represent the weighted criteria and determine the overall value of each alternative. The objective function can be more complex than a simple linear weighting of the criteria and can include probability distribution functions and utility functions associated with each criteria. However, for this example, we will assume the simpler case.

A measure of effectiveness (moe) represents a parameter whose value is critical for achieving the desired mission cost effectiveness. It will also be assumed that the overall mission cost effectiveness can be determined by applying an objective function to a set of criteria, each of which is represented by a measure of effectiveness.

This non-normative extension includes stereotypes for an objective function and a measure of effectiveness. The objective function is a stereotype of a ConstraintBlock and the measure of effectiveness is a stereotype of a block property.

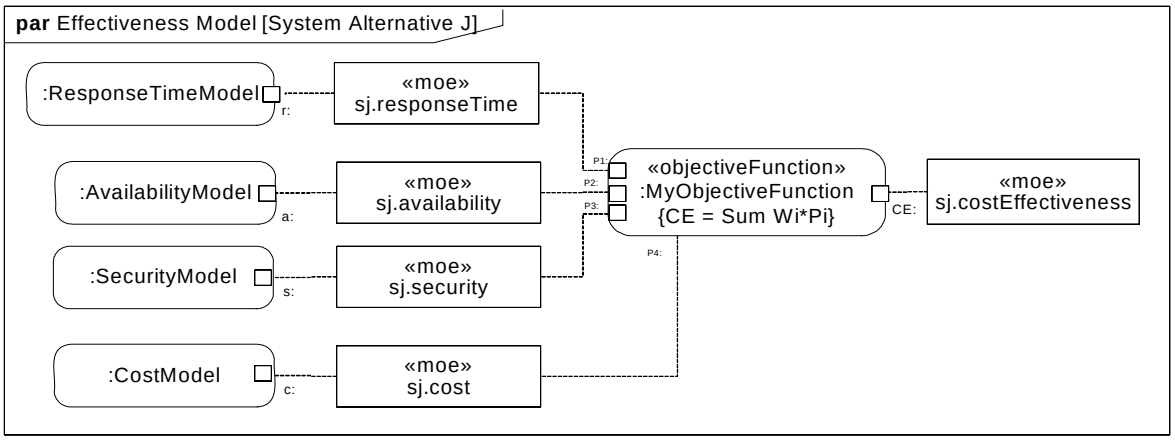
D.3.2 Stereotypes

Table D.5 - Stereotypes for Measures of Effectiveness

Stereotype	Base Class	Properties	Constraints	Description
«objectiveFunction»	«ConstraintBlock» or «ConstraintProperty»	N/A	N/A	An objective function (aka optimization or cost function) is used to determine the overall value of an alternative in terms of weighted criteria and/or moe's.
«moe»	UML4SysML::Property	N/A	N/A	A measure of effectiveness (moe) represents a parameter whose value is critical for achieving the desired mission cost effectiveness.

D.3.3 Stereotype Examples

In this example, operational availability, mission response time, and security effectiveness each represent moes along with life cycle cost. The overall cost effectiveness for each alternative may be defined by an objective function that represents a weighted sum of their moe values. For each moe, there is a separate parametric model to estimate the value of operational availability, mission response time, security effectiveness, and life cycle cost to determine an overall cost effectiveness for each alternative. It is assumed that the moes refer to the values for system alternative j (sj).



D.4 Model Library of SysML Quantity Kinds and Units for ISO 80000-1

This non-normative extension defines a model library of SysML QuantityKind and Unit definitions for a subset of quantities and units defined by the International System of Quantities (ISQ) and the International System of Units (SI). The specific quantities and units in this library are defined by *ISO 80000-1 Quantities and units - Part1: General*. ISO/IEC 80000 currently has fourteen parts that define many quantities and units for use within various fields of science and technology. Part 1 defines

base quantities and units used by other parts as well as a starting set of derived quantities and units with special names and symbols.

The model library defined in this sub clause contains SysML QuantityKind and Unit elements as defined by Clause 8, “Blocks” of this specification. Each QuantityKind or Unit element may optionally carry a “definitionURI” property to document each quantity kind and unit using additional information available from some external source. One option is for this definitionURI to identify an element of a QUDV model (see “Model Library for Quantities, Units, Dimensions, and Values (QUDV)” on page 222) that more fully describes the same quantities and units, including the systems of quantities and units they belong to, and the means by which they may be derived from each other. “Usage Examples” on page 239 contains examples of such QUDV definitions that could be referenced by these definitionURIs.

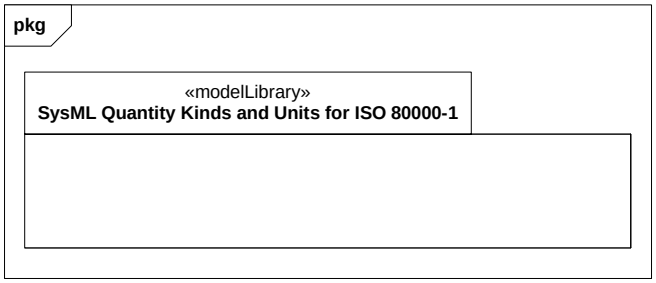


Figure D.4 - Model library of SysML Quantity Kinds and Units for ISO 80000-1

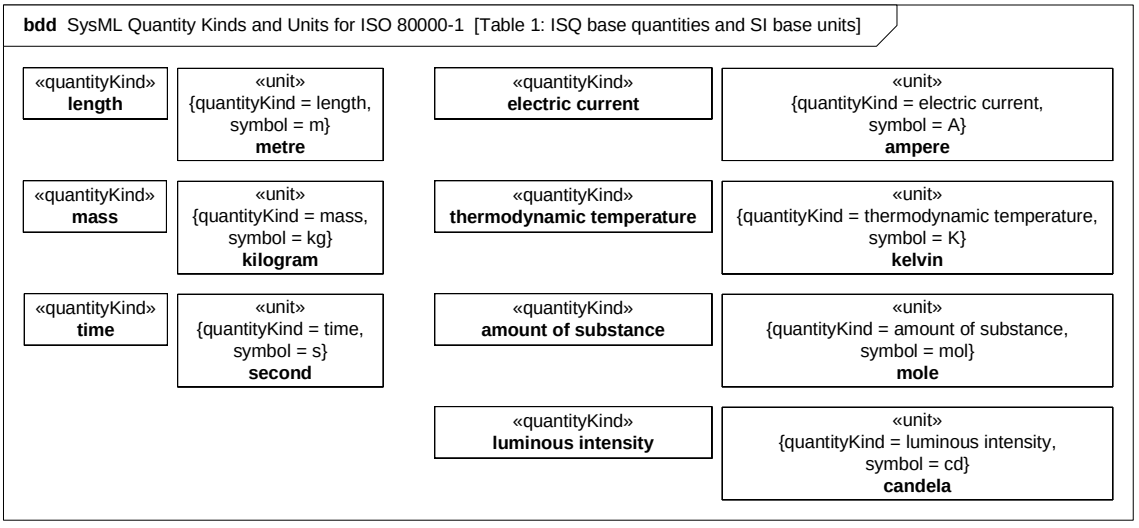


Figure D.5 - ISQ base quantities and SI base units

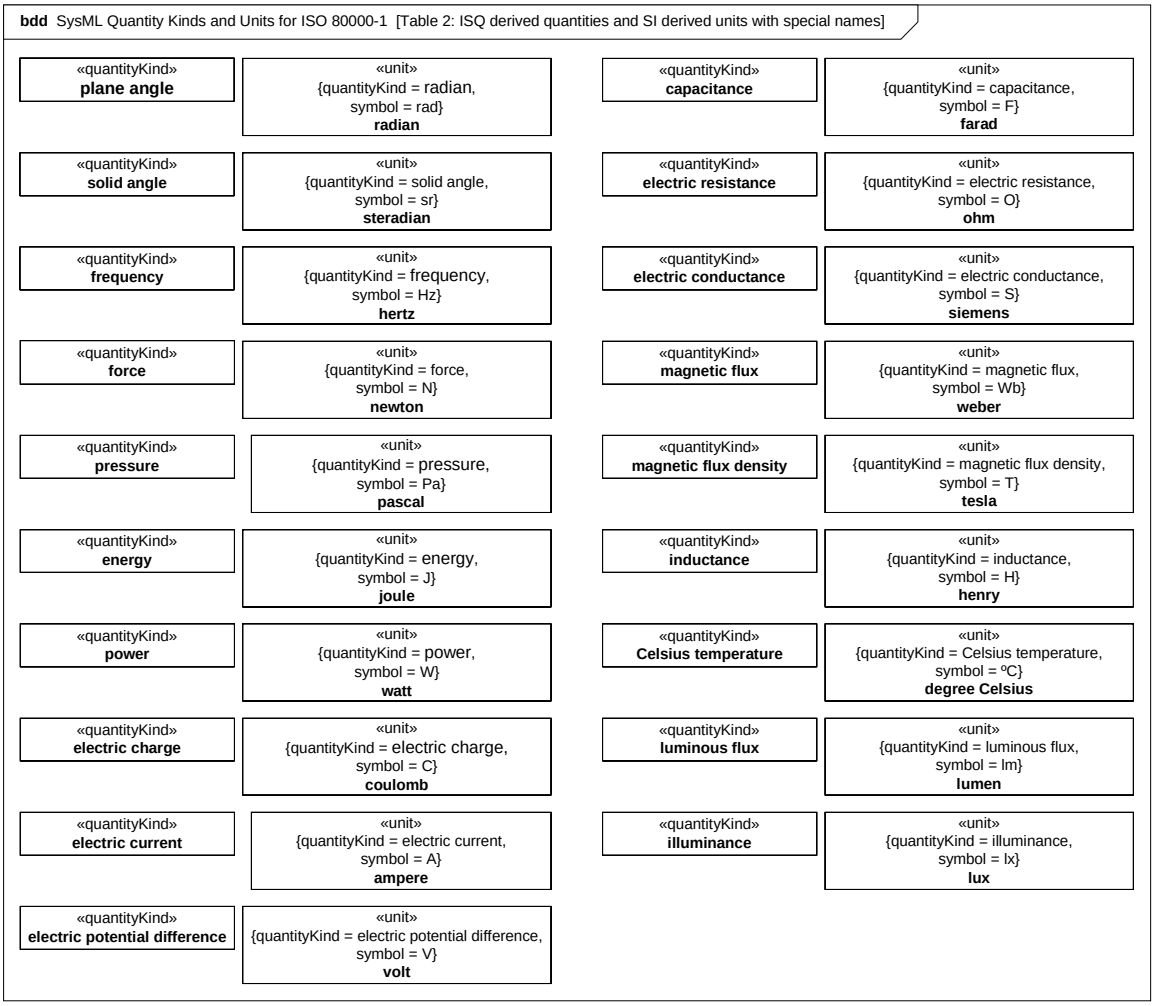


Figure D.6 - ISQ derived quantities and SI derived units with special names

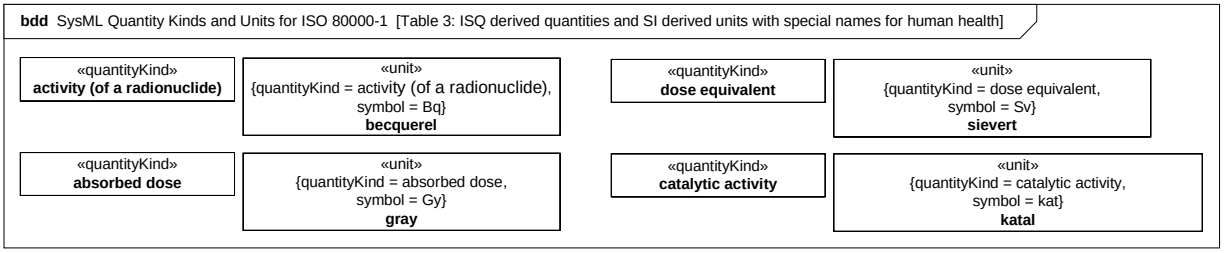


Figure D.7 - ISQ derived quantities and SI derived units with special names for human health

D.5 Model Library for Quantities, Units, Dimensions, and Values (QUDV)

D.5.1 Overview

For any system model, a solid foundation of well-defined quantities, units, and dimensions system is very important. Properties that describe many aspects of a system depend on it. At the same time, such a foundation should be a shareable resource that can be reused in many models within and across organizations and projects.

The most widely accepted, scrutinized, and globally used system of quantities and system of units are the International System of Quantities (ISQ) and the International System of Units (SI). They are formally standardized through [ISO31] and [IEC60027]. The harmonization of these two sets of standards into one new set [ISO/IEC80000] has been published by ISO in 2009 and 2010. The present QUDV model in SysML 1.3 is based on ISO/IEC 80000-1:2009, which refers normatively to the ISO/IEC Guide 99:2007. The ISO/IEC 80000-1:2009 document is also the baseline for the 2010 revision of the IEEE/ASTM American National Standards for Metric Practice SI-10. All the relevant concepts underlying ISQ and SI are publicly available in [VIM]. See Section D.5.3 for references to these documents.

At a minimum, SysML should provide the means to support the imminent international standard [ISO/IEC80000]. In addition, many other systems of quantities and units are still in use for particular applications and for historical reasons. A prime example is the system based on UK Imperial units, which is still widely used in North America. SysML should provide the means to support all such specific systems of quantities and units, including precise definitions of the relationships between different systems of units, and with explicit and unambiguous unit conversions to and from SI as well as other systems.

To provide a solid and stable foundation, the model for defining quantities, units, dimensions, and values in SysML is explicitly based on the concepts defined in [VIM], which have been written by the authoritative Working Group 2 of the Joint Committee for Guides in Metrology (JCGM/WG 2), in which the JCGM member organizations are represented: BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, and OIML. At the same time, the model library is designed in such a way that extensions to the ISQ and SI can be represented, as well as any alternative systems of quantities and units.

The model library can be used to support SysML user models in various ways. A simple approach is to define and document libraries of reusable systems of units and quantities for reuse across multiple projects, and to link units and quantity kinds from these libraries to Unit and QuantityKind stereotypes defined in SysML user models. The name of a Unit or QuantityKind stereotype, its definitionURI, or other means may be used to link it with definitions made using this library. Instances of blocks conforming to this model library may be created by instance specifications, as shown in Section D.5.4, or by other means.

Even though this model library is specified in terms of SysML blocks, its contents could equally be specified by UML classes without dependencies on any SysML extensions. This annex specifies the model library using SysML blocks to maintain compatibility with the SysML specification. UML and other forms of this same conceptual model are important and useful to align different standards with each other and with those of [VIM].

Separate forms of this model library, including a UML class model generated as a simple transformation from the model library specified in this annex, together with additional mappings and resources, example applications, and reference libraries of systems of units and quantities built using this model, are expected to be published via the SysML Project Portal wiki at http://www.omgwiki.org/OMGSysML/.

D.5.2 Abstract Syntax

Figures D.8-D.10 present the QUDV model library in a series of block definition diagrams.

The QUDV Concepts diagram in Figure D.8 presents the core concepts of System of Units, Unit, SystemOfQuantities, and QuantityKind. Distinguished unit values can be organized in a measurement scale for a particular QuantityKind. In SysML, a value property typed by a given ValueType, with stereotype properties that refer to a SysML Unit and/or QuantityKind, defines a quantity in the sense of ISO 80000-1, Sub clause 3.1. If specified, the unit of the ValueType designates the measurement unit assumed for the numerical value of such a quantity. Note that the combination of a unit and quantity kind defined in a

ValueType may be different than the combination of adoptedMeasurementUnit and primaryQuantityKind in the definition of the unit and quantity kind in a QUDV library.

In the QUDV Unit diagram in Figure D.9, SimpleUnit provides the basis for defining other units via conversion or derivation. Additionally, QUDV provides support for specifying a coherent derived unit as a product of the baseUnit(s) of a given SystemOfUnits. In a coherent SystemOfUnits, there is only one base unit for each base quantity kind.

In the QUDV QuantityKind diagram in Figure D.10, SimpleQuantityKind provides the basis for defining other quantity kinds via specialization or derivation. QUDV provides a declarative specification of dimensional analysis to assign to each QuantityKind an expression of its dependence on the baseQuantityKind(s) of a SystemOfQuantities. This dependence is expressed as a list of QuantityKindFactor(s) corresponding to a product of powers of the base quantities. Section D.5.2.18, “SystemOfQuantities” specifies the derivation of quantity dimensions using an algorithm specified in OCL.

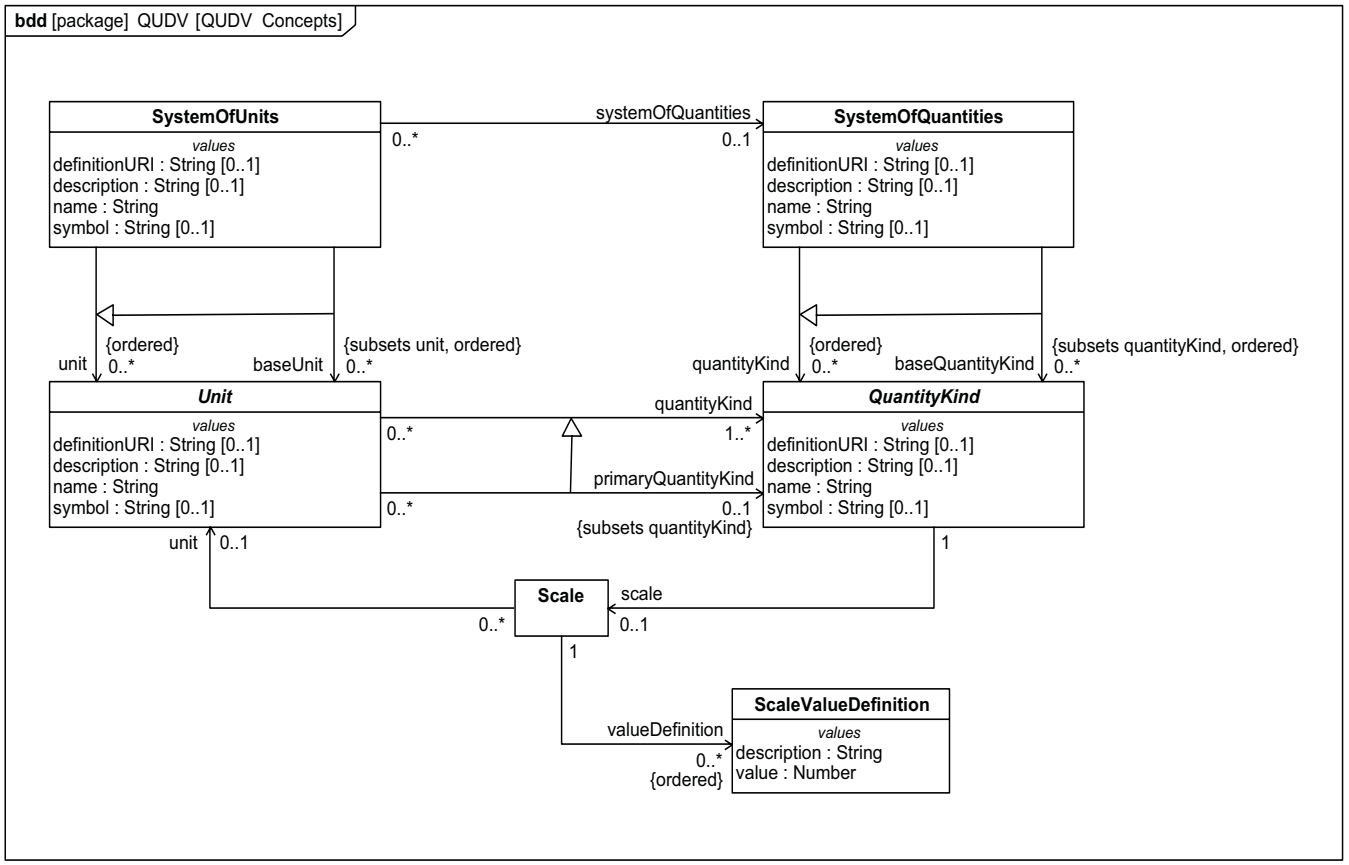


Figure D.8 - QUDV Concepts diagram

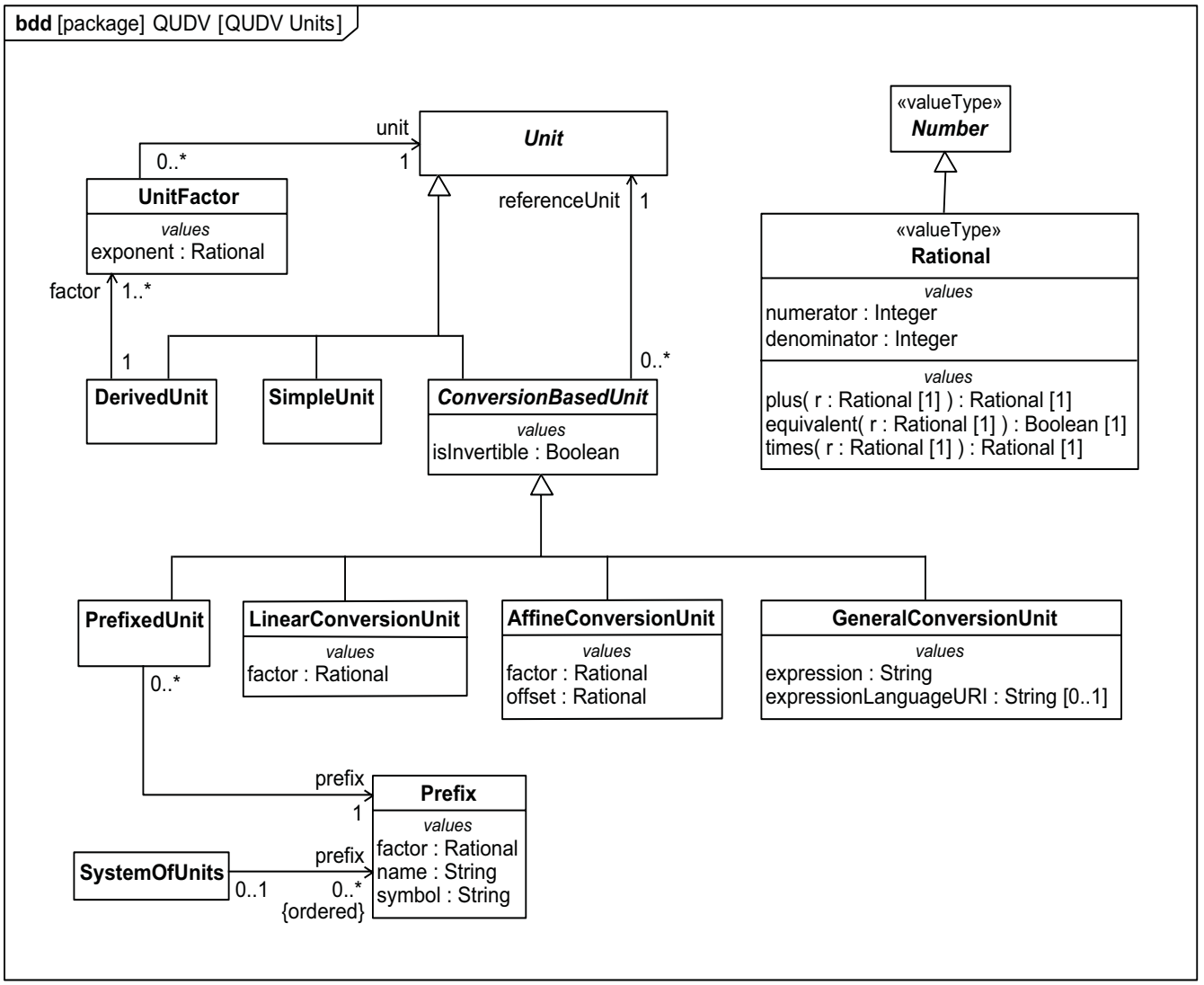


Figure D.9 - QUDV Units diagram

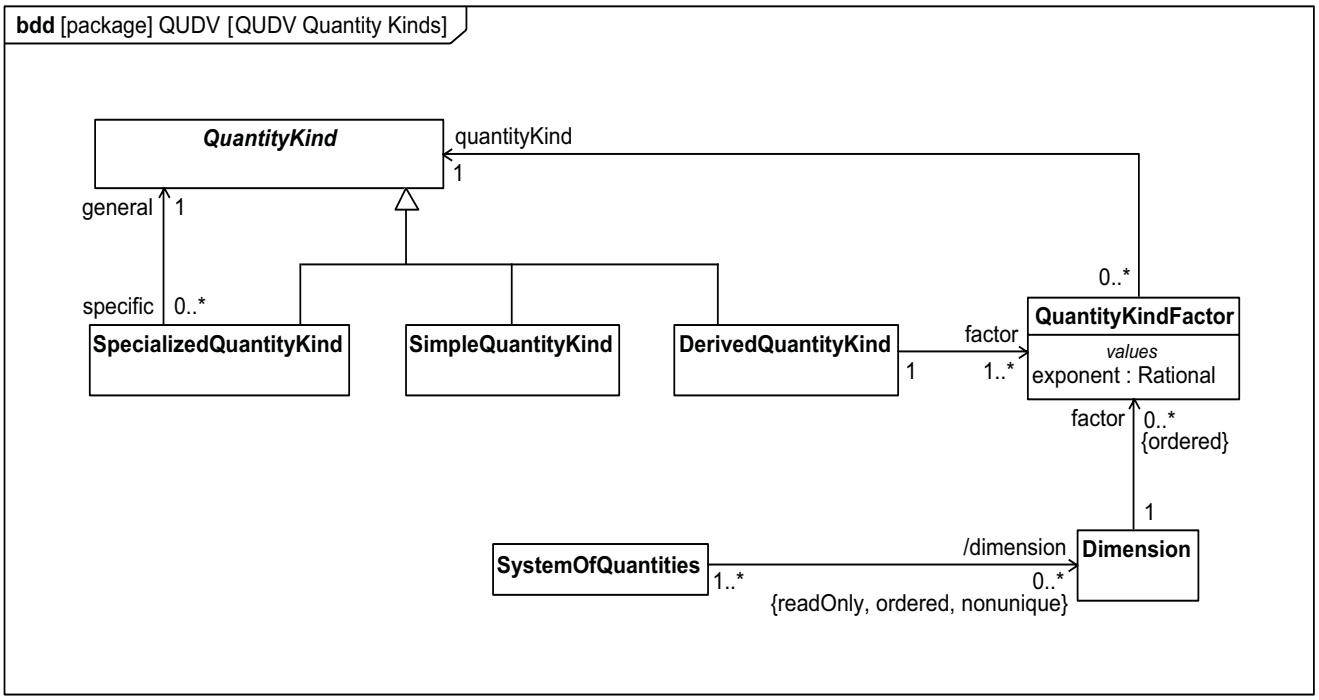


Figure D.10 - QUDV Quantity Kinds diagram

D.5.2.1 AffineConversionUnit

Description

An AffineConversionUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another reference measurement unit through an affine conversion relationship with a conversion factor and offset.

The unit conversion relationship is defined by the following equation:

$$value_{RU} = factor \cdot value_{CU} + offset$$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit, and,
$value_{CU}$ is the quantity value expressed in the AffineConversionUnit.

E.g., in the definition of the AffineConversionUnit for “degree Fahrenheit” with respect to the referenceUnit “degree Celsius,” the factor would be 5/9 and the offset would be -160/9, because

$T_{Celsius} = 5/9 \cdot T_{Fahrenheit} - 160/9$ which is equivalent with $T_{Fahrenheit} = 9/5 \cdot T_{Celsius} + 32/1$

Properties

- factor: Rational
  Rational number that specifies the factor in the unit conversion relationship.
- offset: Rational
  Rational number that specifies the offset in the unit conversion relationship.

D.5.2.2 ConversionBasedUnit

Description

A ConversionBasedUnit is an abstract classifier that is a Unit that represents a measurement unit that is defined with respect to another reference unit through an explicit conversion relationship.

Properties

- referenceUnit: Unit
  Specifies the unit with respect to which the ConversionBasedUnit is defined.
- isInvertible: Boolean
  Specifies whether the unit conversion relationship is invertible. For LinearConversionUnit and AffineConversionUnit this is always true.

D.5.2.3 DerivedQuantityKind

Description

A DerivedQuantityKind is a QuantityKind that represents a kind of quantity that is defined as a product of powers of one or more other kinds of quantity. A DerivedQuantityKind may also be used to define a synonym kind of quantity for another kind of quantity.

For example “velocity” can be specified as the product of “length” to the power one times “time” to the power minus one, and subsequently “speed” can be specified as “velocity” to the power one.

Properties

- factor: QuantityKindFactor [1..*]
  Set of QuantityKindFactor that specifies the product of powers of other kind(s) of quantity that define the DerivedQuantityKind.

D.5.2.4 DerivedUnit

Description

A DerivedUnit is a Unit that represents a measurement unit that is defined as a product of powers of one or more other measurement units.

For example the measurement unit “metre per second” for “velocity” is specified as the product of “metre” to the power one times “second” to the power minus one.

Properties

- factor: UnitFactor [1..*]
  Set of UnitFactor that specifies the product of powers of other measurement units that define the DerivedUnit.

D.5.2.5 Dimension

Description

A Dimension represents the [VIM] concept of “quantity dimension” that is defined as “expression of the dependence of a quantity on the base quantities of a system of quantities as a product of powers of factors corresponding to the base quantities, omitting any numerical factor.”

For example in the ISQ the quantity dimension of “force” is denoted by $\dim F = L \cdot M \cdot T^{-2}$, where “F” is the symbol for “force,” and “L,” “M,” and “T” are the symbols for the ISQ base quantities “length,” “mass,” and “time” respectively.

The Dimension of any QuantityKind can be derived through the algorithm that is defined in Section D.5.2.18 with SystemOfQuantities. The actual Dimension for a given QuantityKind depends on the choice of baseQuantityKind specified in a SystemOfQuantities.

Properties

- symbolicExpression: String [0..1]
  Symbolic expression of the quantity dimension's product of powers, in terms of symbols of the kinds of quantity that represent the base kinds of quantity and their exponents. In tool implementations, the symbolicExpression may automatically derived from the associated factor set.
- factor: QuantityKindFactor [0..*] {ordered}
  Ordered set of QuantityKindFactor that specifies the product of powers of base dimensions that define the Dimension. The possible base dimensions are represented by the ordered set of baseQuantityKind defined in the SystemOfQuantities for which the Dimension is specified. The order of the factors should follow the ordered set of baseQuantityKind in SystemOfQuantities.

D.5.2.6 GeneralConversionUnit

Description

A GeneralConversionUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another reference measurement unit through a conversion relationship expressed in some syntax through a general mathematical expression.

The unit conversion relationship is defined by the following equation:

$$value_{RU} / value_{CU} = f(value_{RU}, value_{CU})$$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit and
$value_{CU}$ is the quantity value expressed in the GeneralConversionUnit and
$f(value_{RU}, value_{CU})$ is a mathematical expression that includes $value_{RU}$ and $value_{CU}$

Properties

- expression: String
  Specifies the unit conversion relationship in some expression syntax.
- expressionLanguageURI: String [0..1]
  URI that specifies the language for the expression syntax.

D.5.2.7 LinearConversionUnit

Description

A LinearConversionUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another measurement reference unit through a linear conversion relationship with a conversion factor.

The unit conversion relationship is defined by the following equation:

$$value_{RU} = factor \cdot value_{CU}$$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit, and,
$value_{CU}$ is the quantity value expressed in the LinearConversionUnit.

For example, in the definition of the LinearConversionUnit for “inch” with respect to the referenceUnit “metre,” the factor would be 254/10000, because 0.0254 metre = 1 inch.

Properties

- factor: Rational
  Rational number that specifies the factor in the unit conversion relationship.

D.5.2.8 Prefix

Description

A Prefix represents a named multiple or submultiple multiplication factor used in the specification of a PrefixedUnit. A SystemOfUnits may specify a set of prefixes.

Properties

- name: String
  Name of the prefix.
- symbol: String [0..1]
  Short symbolic name of the prefix.
- factor: Real [1]
  Specifies the multiple or submultiple multiplication factor.

D.5.2.9 PrefixedUnit

Description

A PrefixedUnit is a ConversionBasedUnit that represents a measurement unit that is defined with respect to another measurement reference unit through a linear conversion relationship with a named prefix that represents a multiple or submultiple of a unit.

[VIM] defines “multiple of a unit” as “measurement obtained by multiplying a given measurement unit by an integer greater than one” and “submultiple of a unit” as “measurement unit obtained by dividing a given measurement unit by an integer greater than one.”

The unit conversion relationship is defined by the following equation:

$$value_{RU} = factor \cdot value_{CU}$$

where:

$value_{RU}$ is the quantity value expressed in the referenceUnit and
$value_{CU}$ is the quantity value expressed in the PrefixedUnit.

For example, in the definition of the PrefixedUnit for “megabyte” with respect to the referenceUnit “byte,” the prefix would be the Prefix for “mega” with a factor 10^6, because 10^6 byte = 1 megabyte.

See [VIM] for all decimal and binary multiples and decimal submultiples defined in SI.

Properties

- prefix: Prefix
  Specifies the prefix that defines the name, symbol, and factor of the multiple or submultiple.

Constraints

[1] The referenceUnit shall not be a PrefixedUnit, i.e., it is not allowed to prefix an already prefixed measurement unit. In general the referenceUnit should be a SimpleUnit.

```
package QUDV
context PrefixedUnit
inv: not referenceUnit.oclIsTypeOf(PrefixedUnit)
endpackage
```

D.5.2.10 QuantityKind

Description

A QuantityKind is an abstract classifier that represents the [VIM] concept of “kind of quantity” that is defined as “aspect common to mutually comparable quantities.” A QuantityKind represents the essence of a quantity without any numerical value or unit. Quantities of the same kind within a given system of quantities have the same quantity dimension. However, quantities of the same dimension are not necessarily of the same kind.

Properties

- name: String
  Name of the kind of quantity.
- symbol: String [0..1]
  Short symbolic name of the kind of quantity.
- description: String [0..1]
  Textual description of the kind of quantity.
- definitionURI: String [0..1]
  URI that references an external definition of the kind of quantity.
- scale: Scale [0..1]
  Specification of a Scale that is associated to the QuantityKind.

D.5.2.11 QuantityKindFactor

Description

A QuantityKindFactor represents a factor in the product of powers that defines a DerivedQuantityKind.

Properties

- exponent: Rational
  Rational number that specifies the exponent of the power to which the quantityKind is raised.
- quantityKind: QuantityKind
  Reference to the QuantityKind that participates in the factor.

D.5.2.12 Rational

Description

A Rational value type represents the mathematical concept of a number that can be expressed as a quotient of two integers. It may be used to express the exact value of such values, without issues of rounding or other approximations if the result of the division were used instead.

Properties

- numerator: Integer
  An integer number used to express the numerator of a rational number.
- denominator: Integer
  An integer number used to express the denominator of a rational number.

Operations

```
package QUDV
context Rational
def: plus(r : Rational[1]) : Rational[1]
       = result.numerator = self.numerator * r.demonimator
+ r.numerator * self.denominator
and result.denominator = self.denominator * r.denominator

context Rational
def: equivalent(r : Rational[1]) : Boolean[1]
       = result = (  self.numerator * r.demonimator
= r.numerator * self.denominator)

context Rational
def: times(r : Rational[1]) : Rational[1]
       = result.numerator = self.numerator * r.numerator
and result.denominator = self.denominator * r.denominator
endpackage
```

Constraints

[1] The denominator of a rational number must not be zero.

```
package QUDV
context Rational
inv: denominator <> 0
endpackage
```

D.5.2.13 Scale

Description

A Scale represents the [VIM] concept of a “measurement scale” that is defined as an “ordered set of quantity values of quantities of a given kind of quantity used in ranking, according to magnitude, quantities of that kind.” A Scale specifies one or more fixed values that have a specific significance in the definition of the associating QuantityKind.

For example the “thermodynamic temperature” kind of quantity is defined by specifying the values of 0 and 273.16 kelvin as the temperatures of absolute zero and the triple point of water respectively.

A Scale does not always need to specify a unit. For example the “Rockwell C Hardness Scale” or the “Beaufort Wind Force Scale” are ordinal scales that do not have a particular associated unit. Similarly, subjective scales for a “priority” or “risk” kind of quantity with e.g., value definitions 0 for “low,” 1 for “medium,” and 3 for “high” do not have a particular associated unit.

Properties

- valueDefinition: ScaleValueDefinition [1..*] {ordered}
  Ordered set of ScaleValueDefinition that specifies the defined numerical value(s) and textual definition(s) for the measurement scale.
- unit: Unit [0..1]
  Optionally specifies the unit in which the value of each valueDefinition is expressed.

Constraints

[1] A scale must be the same as the scale of the unit's primaryQuantityKind, if any.

```
package QUDV
context Scale
inv: unit = null
       or unit.primaryQuantityKind = null
       or unit.primaryQuantityKind.scale = null
       or unit.primaryQuantityKind.scale = self
endpackage
```

D.5.2.14 ScaleValueDefinition

Description

A ScaleValueDefinition represents a specific value for a measurement scale.

Properties

- value: Number
  Specifies the numerical value.
- definition: String
  Specifies the textual definition for the value.

D.5.2.15 SimpleQuantityKind

Description

A SimpleQuantityKind is a QuantityKind that represents a kind of quantity that does not depend on any other QuantityKind. Typically a base quantity would be specified as a SimpleQuantityKind.

D.5.2.16 SimpleUnit

Description

A SimpleUnit is a Unit that represents a measurement unit that does not depend on any other Unit. Typically a base unit would be specified as a SimpleUnit.

D.5.2.17 SpecializedQuantityKind

Description

A SpecializedQuantityKind is a QuantityKind that represents a kind of quantity that is a specialization of another kind of quantity.

For example, “distance,” “width,” “depth,” “radius,” and “wavelength” can all be specified as specializations of the “length” SimpleQuantityKind.

Properties

- general: QuantityKind
  Specifies the QuantityKind that is specialized.

D.5.2.18 SystemOfQuantities

Description

A SystemOfQuantities represents the [VIM] concept of “system of quantities” that is defined as a “set of quantities together with a set of non-contradictory equations relating those quantities.” It collects a list of QuantityKind that specifies the kinds of quantity that are known in the system.

The International System of Quantities (ISQ) is an example of a SystemOfQuantities, defined in [ISO31] and [ISO/IEC80000].

Properties

- name: String
  Name of the system of quantities.
- symbol: String [0..1]
  Short symbolic name of the system of quantities.
- description: String [0..1]
  Textual description of the system of quantities.
- definitionURI: String [0..1]
  URI that references an external definition of the system of quantities. Note that as part of [ISO/IEC80000] normative URIs for each of the ISQ quantities and SI units are being defined.
- quantityKind: QuantityKind [0..*] {ordered}
  Ordered set of QuantityKind that specifies the kinds of quantity that are known in the system.
- baseQuantityKind: QuantityKind [0..*] {ordered, subsets quantityKind}
  Ordered set of QuantityKind that specifies the base quantities of the system of quantities. This is a subset of the complete quantityKind list. The base quantities define the basis for the quantity dimension of a kind of quantity.
- /dimension: Dimension [0..*] {ordered, readOnly, nonunique}
  Derived ordered set of Dimension. The actual dimension of a QuantityKind depends on the list of baseQuantityKind that are specified in an actual SystemOfQuantities, see the DerivedDimensions constraint.

Constraints

[1] All quantity dimensions are derived through the following algorithm specified in OCL.

```
package QUDV
-- get the set of units, if any, that a given unit directly depends on
```

```
context Unit
def: directUnitDependencies : Set(Unit) =
       if oclIsKindOf(ConversionBasedUnit)
       then oclAsType(ConversionBasedUnit).referenceUnit
       else
              if oclIsKindOf(DerivedUnit)
              then oclAsType(DerivedUnit).factor->collect(unit)->asSet()
              else Set{}
       endif
endif

-- get the set of units, if any, that a given unit directly or indirectly depends on
context Unit
def: allUnitDependencies : Set(Unit)
       = self->closure(directUnitDependencies)

context Unit
inv acyclic_unit_dependencies
       : not allUnitDependencies->excludes(self)

-- get the set of quantityKinds, if any, that a given quantityKind directly depends on
context QuantityKind
def: directQKindDependencies : Set(QuantityKind)
       = if oclIsKindOf(DerivedQuantityKind)
              then oclAsType(DerivedQuantityKind).factor
->collect(quantityKind)->asSet()
       else
              if oclIsKindOf(SpecializedQuantityKind)
              then oclAsType(SpecializedQuantityKind).general
              else Set{}
              endif
       endif

context QuantityKind
def: allQuantityKindDependencies : Set(QuantityKind)
       = self->closure(directQKindDependencies)

context QuantityKind
inv acyclic_quantity_kind_dependencies
       : allQuantityKindDependencies->excludes(self)

--context SystemOfQuantities::deriveQuantityKindDimensions() :
--post: quantityKind->forAll(qK|qK.hasProperDimension(self))
-- The derived dimension of a simple quantity kind must
-- have exactly one factor
-- whose numerator and denominator are equal to 1.

context SimpleQuantityKind
def: hasProperDimension(sq:SystemOfQuantities) : Boolean
       = let d:Dimension=sq.getDimension(self)
       in d.factor->size()=1
       and d.factor->forAll(exponent->forAll(numerator=1 and denominator=1))

-- The derived dimension of a specialized quantity kind is
-- the dimension of its general quantity kind.
```

```
context SpecializedQuantityKind
def: hasProperDimension(sq:SystemOfQuantities) : Boolean
       = sq.getDimension(self) = sq.getDimension(general)

-- A helper function to produce the factor/quantityKind tuples
-- for a given Dimension.
context Dimension
def: dimFactors : Bag(Tuple(factor:Rational,qKind:QuantityKind))
       = self.factor->collect(qkf | Tuple{factor=qkf.exponent,qKind=qkf.quantityKind})

-- A helper function to get all the factor/quantityKind tuples
-- for the dimension factors of a derived quantity kind.
context DerivedQuantityKind
def: derQFactors(sq:SystemOfQuantities) : Bag(Tuple(factor:Rational,qKind:QuantityKind))
       = self.factor->collect(qkf |
              let qd:Dimension = sq.getDimension(qkf.quantityKind) in
              qd.factor->collect(qkf |
              Tuple{factor=qkf.exponent.plus(df.exponent),qKind=qkf.quantityKind}))

-- Reduce a bag of factor/quantityKind tuples by combining
-- all factors for the same quantity kind
-- and eliminating the zero-factor/quantityKind tuples
context DerivedQuantityKind
def: reducetoNonZeroUniqueFactors(
       qFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind)),
       qKinds:Set(QuantityKind))
: Bag(Tuple(factor:Rational,qKind:QuantityKind))
= let uqFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
= qKinds->collect(
       -- for each unique quantity kind, qKind1,
       -- from the set of unique quantity kinds, qKinds...
       qKind1:QuantityKind|
       -- get the sequence of factors from the set of
       -- qFactors tuples whose quantity kind is qKind1...
       let factor1s:Sequence(Rational)
              = qFactors->select(qKind=qKind1)
                     ->collect(factor)->asSequence()
              -- start with the first factor, factor1,
              -- from all the factor1s associated to qKind1...
              in let factor1:Rational=factor1s->first()
                     -- construct the factor/quantityKind tuple
                     -- for qKind1 where
                     -- the factor is the product of factor1 with
                     -- all remaining factors1s
                     in Tuple{
                            factor=factor1s->excluding(factor1)->iterate(
                                   factorI:Rational;
                                   factorN:Rational=factorI |
                                          factorN.plus(factorI)),
                            qKind=qKind1})
-- eliminate the factor/quantityKind tuples where
-- the factor is zero
in let nqFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
       = uqFactors->select(factor.numerator<>0)
in nqFactors
```

```
-- The derived dimension of a derived quantity kind is
-- the simplified set of factor/quantityKind tuples
-- for the derived quantity kind. The simplified set
-- of factor/quantityKind tuples has
-- one factor/quantityKind tuple for each quantityKind where
-- the simplified factor is a non-zero product of
-- all the factors in the factor/quantityKind tuples.
context DerivedQuantityKind
def: hasProperDimension(sq:SystemOfQuantities) : Boolean
       = let d:Dimension = sq.getDimension(self)
       in let resFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
       = d.dimFactors
       -- the unique quantityKinds from the result...
       in let resKinds:Set(QuantityKind)
              =resFactors->collect(qKind)->asSet()

       -- the factor/quantityKind tuples from the derived quantity...
       in let qFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
       = self.derQFactors(sq)

       -- the unique quantityKinds from the derived quantity...
       in let qKinds:Set(QuantityKind)=qFactors->collect(qKind)->asSet()

       -- get the reduced non-zero factor/quantityKinds...
       in let nqFactors:Bag(Tuple(factor:Rational,qKind:QuantityKind))
       = self.reducetoNonZeroUniqueFactors(qFactors, qKinds)

       -- condition1: there should be the same number
       -- of factor/quantityKind tuples in the result
       -- compared to the non-zero unique factor/quantityKind
       -- tuples for the derivedQuantityKind
       in nqFactors->size() = resFactors->size()

       -- condition2: there should be the same set of
       -- quantity kinds in the result
       -- and in the non-zero unique factor/quantityKind tuples
       -- and qKinds->symmetricDifference(resKinds)->isEmpty()

       -- condition3: for each quantity kind,
       -- the factors in the result and
       -- in the reduced non-zero unique factor/quantityKind
       -- tuples should be equivalent rationals
       and qKinds->forAll(qk:QuantityKind|
       let nFactor:Rational
              =nqFactors->select(qKind=qk)
              ->collect(factor)->asSequence()->first()
       in let rFactor:Rational
              =resFactors->select(qKind=qk)
              ->collect(factor)->asSequence()->first()
       in nFactor.equivalent(rFactor))
endpackage
```

[2] In a well-formed SystemOfQuantities, all the general quantityKinds of every SpecializedQuantityKind are defined in the SystemOfQuantities (SoQ1) and each QuantityKindFactor of any DerivedQuantityKind refers to a QuantityKind in that SystemOfQuantities.

```
package QUDV
inv SoQ1:
       quantityKind->includesAll(
              quantityKind->select(oclIsKindOf(SpecializedQuantityKind))
              ->collect(oclAsType(SpecializedQuantityKind).general))
inv SoQ2:
       quantityKind->includesAll(
              quantityKind->select(oclIsKindOf(DerivedQuantityKind))
              ->collect(oclAsType(DerivedQuantityKind).factor
                     ->collect(quantityKind)))
endpackage
```

D.5.2.19 SystemOfUnits

Description

A SystemOfUnits represents the [VIM] concept of “system of units” that is defined as “set of base units and derived units, together with their multiples and submultiples, defined in accordance with given rules, for a given system of quantities.” It collects a list of Unit that are known in the system. A QUDV SystemOfUnits only optionally defines multiples and submultiples.

Properties

- name: String
  Name of the system of units.
- symbol: String [0..1]
  Short symbolic name of the system of units.
- description: String [0..1]
  Textual description of the system of units.
- definitionURI: String [0..1]
  URI that references an external definition of the system of units. Note that as part of [ISO/IEC80000] normative URIs for each of the quantities in the ISQ and units in the SI are being defined.
- unit: Unit [0..*] {ordered}
  Ordered set of Unit that specifies the units that are known in the system.
- baseUnit: Unit [0..*] {ordered, subsets unit}
  Ordered set of Unit that specifies the base units of the system of units. A “base unit” is defined in [VIM] as a “measurement unit that is adopted by convention for a base quantity.” It is the (preferred) unit in which base quantities of the associated systemOfQuantities are expressed.
- prefix: Prefix [0..*] {ordered}
  Ordered set of Prefix that specifies the prefixes for multiples and submultiples of units in the system.
- systemOfQuantities: SystemOfQuantities [0..1]
  Reference to the SystemOfQuantities for which the units are specified.

Constraints

[1] In a coherent system of units, there is only one base unit for each base quantity.

```
package QUDV
context SystemOfUnits
def: isCoherent() : Boolean =
```

```
        baseUnit->size() = systemOfQuantities.baseQuantityKind->size()
        and baseUnit
        ->forAll(bU|systemOfQuantities.baseQuantityKind
                ->one(bQK|bU.primaryQuantityKind=bQK))
        and systemOfQuantities.baseQuantityKind
        ->forAll(bQK|baseUnit->one(bU|bQK=bU.primaryQuantityKind))
endpackage
```

[2] A coherent derived unit is a derived unit that, for a given system of quantities and for a chosen set of base units, is a product of powers of base units with no other proportionality factor than one.

```
package QUDV
context SystemOfUnits
def: isCoherent(du : DerivedUnit) : Boolean =
        baseUnit->includesAll(du.factor->collect(unit))
        and du.factor->collect(exponent)
                ->forAll(numerator=1 and denominator=1)
endpackage
```

[3] In a well-formed SystemOfUnits, all of the prefixes of PrefixedUnits are defined in the SystemOfUnits (SoU1); each UnitFactor of any DerivedUnit refers to a Unit in the SystemofUnits (SoU2); all ConversionBasedUnits refer to units in the SystemOfUnits (SoU3); and all of the quantityKinds that are measurementUnits of Units in the SystemOfUnits are defined in the systemOfQuantities of that SystemOfUnit (SoU4).

```
package QUDV
context SystemOfUnits
context SystemOfUnits
inv SoU3_1:
        prefix->includesAll(
        unit->select(oclIsTypeOf(PrefixedUnit))
                ->collect(oclAsType(PrefixedUnit).prefix))
inv SoU3_2:
        unit->includesAll(
        unit->select(oclIsTypeOf(DerivedUnit))
                ->collect(oclAsType(DerivedUnit).factor->collect(unit)))
inv SoU3_3:
        unit->includesAll(
                unit->select(oclIsTypeOf(ConversionBasedUnit))
                ->collect(oclAsType(ConversionBasedUnit).referenceUnit))
inv SoU3_4:
        systemOfQuantities = null or
        systemOfQuantities.quantityKind->includesAll(
                unit->collect(quantityKind))
endpackage
```

D.5.2.20 Unit

Description

A Unit is an abstract classifier that represents the [VIM] concept of “measurement unit” that is defined as “real scalar quantity, defined and adopted by convention, with which any other quantity of the same kind can be compared to express the ratio of the two quantities as a number.”

Properties

- name: String
  Name of the unit.
- symbol: String [0..1]
  Short symbolic name of the unit.
- description: String [0..1]
  Textual description of the unit.
- definitionURI: String [0..1]
  URI that references an external definition of the unit.
- quantityKind : QuantityKind[0..*]
  Since a Unit is a particular value for a quantity of a given kind, every Unit must be the measurementUnit of at least one QuantityKind.
- primaryQuantityKind : QuantityKind[0..1] {subsets quantityKind}
  For a given SystemOfUnits, each chosen baseUnit must have an adopted primaryQuantityKind which is its adoptedMeasurementUnit. Here the notion of "primary" is meant to designate the particular quantity kind that is used in the formal definition of a measurement unit, as is done in ISO/IEC 80000 (see Figure D.11).

D.5.2.21 UnitFactor

Description

A UnitFactor represents a factor in the product of powers that defines a DerivedUnit.

Properties

- exponent: Rational
  Rational number that specifies the exponent of the power to which the unit is raised.
- unit: Unit
  Reference to the Unit that participates in the factor.

D.5.3 References

[VIM]
JCGM 200:2008, International Vocabulary of Metrology - Basic and General Concepts and Associated Terms (VIM), 3rd edition, 2008, BIPM, Paris, France. Available for download in PDF format from BIPM's website http://www.bipm.org. This third edition is also published on paper by ISO (ISO/IEC Guide 99-12:2007, International Vocabulary of Metrology – Basic and General Concepts and Associated Terms, VIM).

[ISO/IEC80000]
ISO/IEC 80000, Quantities and units. 15 parts, some published, some still in progress, harmonized replacement of [ISO31] and [IEC60027], the new international system of quantities and units.

[ISO31]
ISO 31, Quantities and units (Third edition 1992-08-01). Specifies the international system of units - SI - in 14 parts.

[IEC60027]
IEC 60027-2:2005, Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics (Third edition 2005-08).

[SI-Brochure]
Le Système international d'unités (SI) / The International System of Units (SI), 8th edition 2006, BIPM, (French and English). Available for download in PDF format from http://www.bipm.org/en/si/si_brochure.

[NIST330]
The International System of Units (SI), NIST Special Publication 330, 2008 Edition. NOTE: U.S. version of the English language text of [SI-Brochure].
Available for download in PDF format from http://physics.nist.gov/cuu/Units/bibliography.html.

[NIST822]
Guide for the Use of the International System of Units (SI), NIST Special Publication 811, 2008 Edition.
Available for download in PDF format from http://physics.nist.gov/cuu/Units/bibliography.html

D.5.4 Usage Examples

D.5.4.1 SI Unit and QuantityKind examples

Figure D.11 shows an approach for defining base units of the System International of Units defined in http://www.bipm.org/en/si/si_brochure/chapter2/2-1/ and http://physics.nist.gov/cuu/Units/units.html. This approach involves instantiating the concrete classes of Unit shown in Figure D.9.

Figure D.12 diagram shows the definition of “newton” as a DerivedUnit (D.5.2.4) corresponding to the “force” DerivedQuantityKind (D.5.2.3). Derived units and quantity kinds are defined as products of factors on other units and quantity kinds respectively. In the QUDV, the product factors of a DerivedUnit (resp. DerivedQuantityKind) are all of the UnitFactor (resp. DerivedUnitFactor) at the “factor” ends of association link instances.

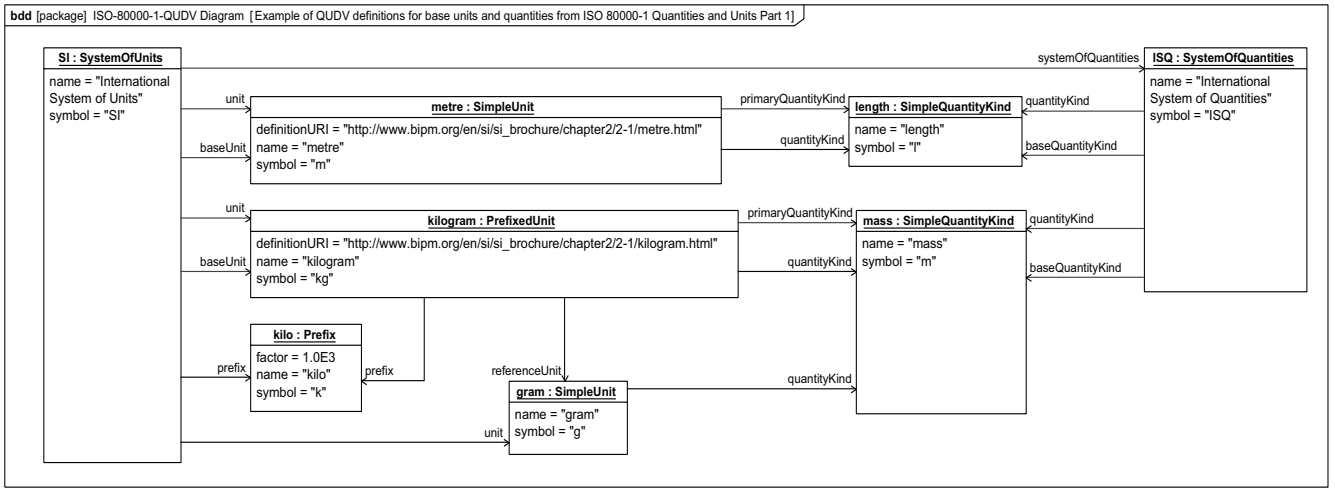


Figure D.11 - Base Unit and Quantity Kinds of the SI and ISQ respectively

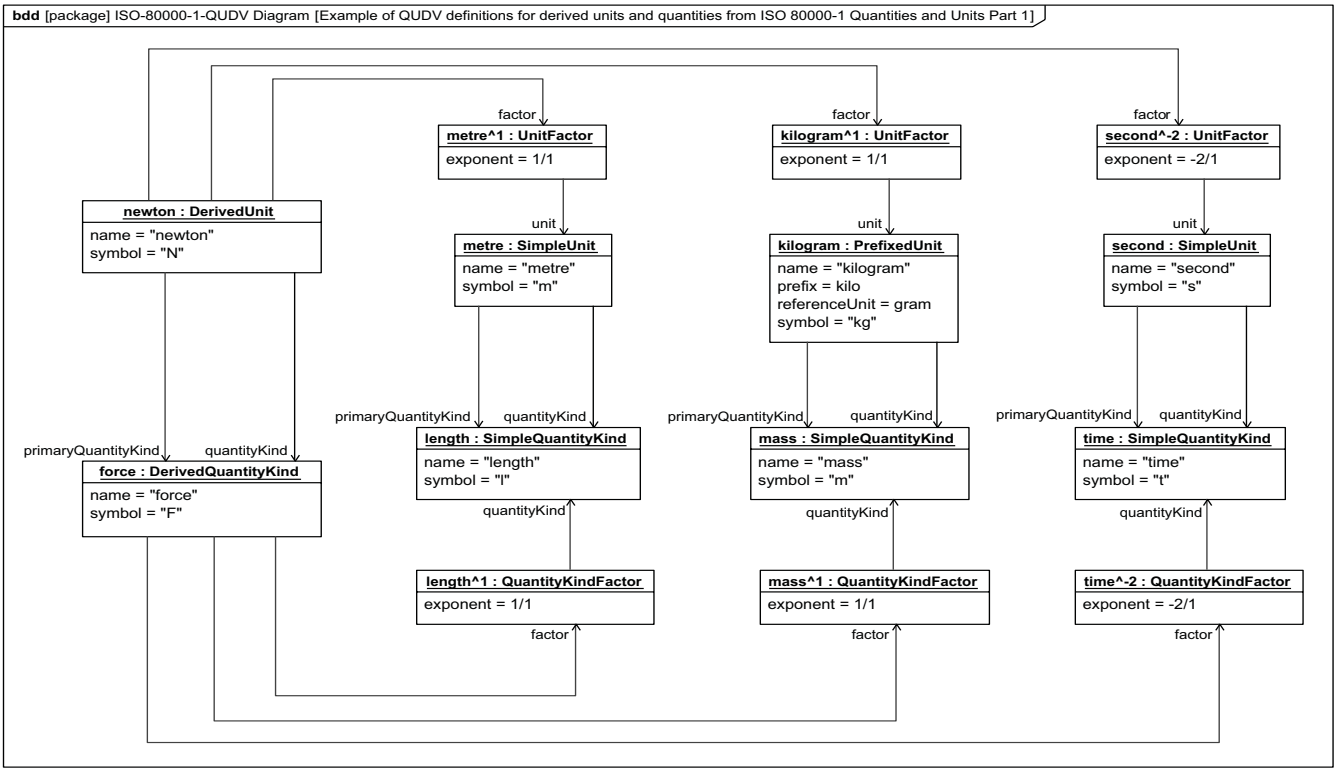


Figure D.12 - Example of a derived unit and derived quantity kind

D.5.4.2 Spring example

Figure D.13 shows a simple model of the length of a spring defined as the linear distance between the linear position of its two flange ends. QUDV supports defining arbitrary systems of units and quantities. Although this example uses only one unit, “metre” and one quantity kind, “lengthQK;” this example illustrates specialized value types to make additional distinctions such as “LinearPosition” vs. “LinearDistance,” two distinct quantities that have the same unit and quantity kind. This example illustrates an instance of a spring and uses the dot pathname property notation defined for IBDs (Section 8.3.1.2) to clearly indicate the role of each instance specification.

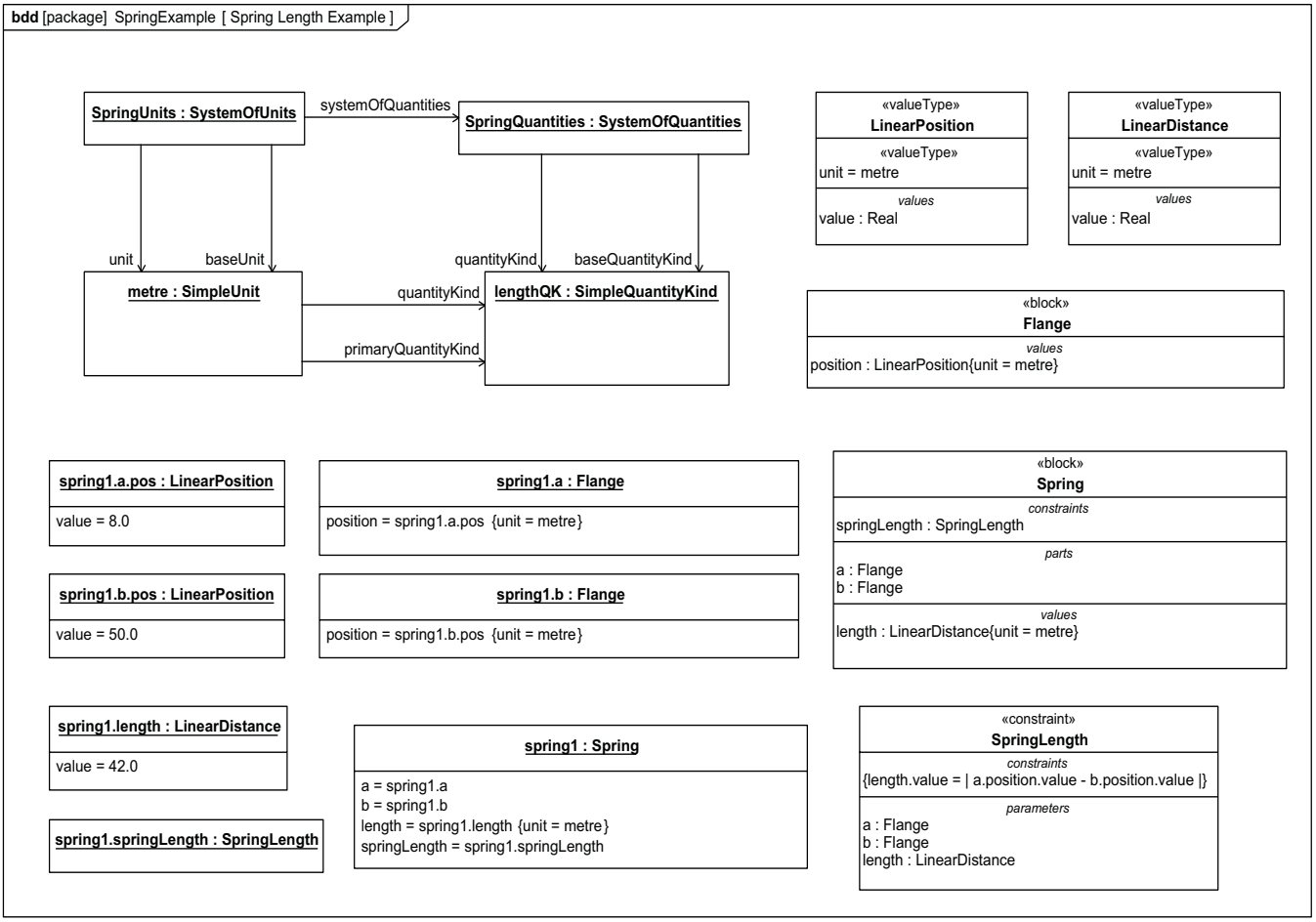


Figure D.13 - Spring Length Example

D.6 Distribution Extensions

D.6.1 Overview

This sub clause describes a non-normative extension to provide a candidate set of distributions (see “DistributedProperty” on page 47). It consists of a profile containing stereotypes that can be used to specify distributions for properties of blocks.

D.6.2 Stereotypes

D.6.2.1 Package Distributions

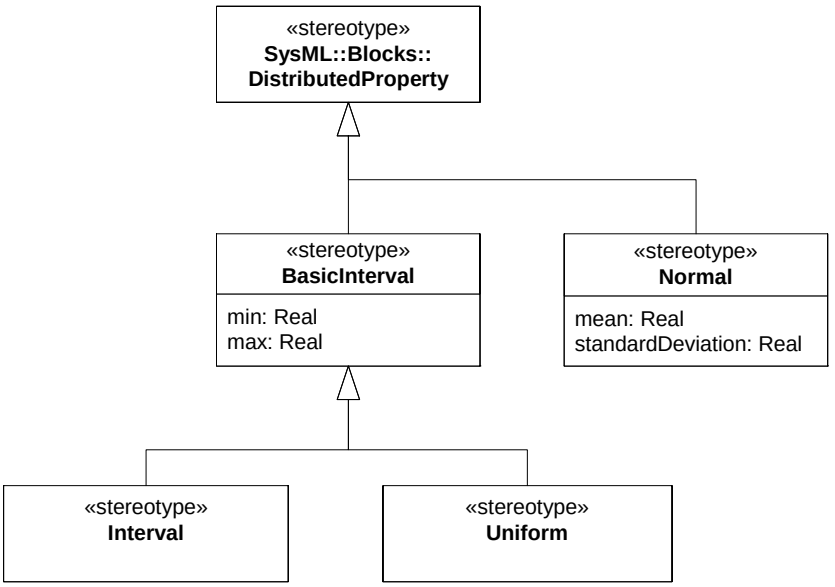


Figure D.14 - Basic distribution stereotypes

Table D.6 - Distribution Stereotypes

Stereotype	Base Class	Properties	Constraints	Description
«BasicInterval»	«DistributedProperty»	min:Real max:Real	N/A	Basic Interval distribution - value between min and max inclusive
«Interval»	«BasicInterval»	N/A	N/A	Interval distribution - unknown probability between min and max
«Uniform»	«BasicInterval»	N/A	N/A	Uniform distribution - constant probability between min and max
«Normal»	«DistributedProperty»	mean:Real standardDeviation:Real	N/A	Normal distribution - constant probability between min and max

D.6.3 Usage Example

Figure D.15 shows a simple example of using distributions; the force of the Cannon is specified using a Normal distribution with parameters mean and standard Deviation. Whereas the use of a Normal distribution can be inferred from the names of its parameters, an Interval distribution shares parameters with a Uniform distribution, hence the stereotype keyword «interval» is used to distinguish it.

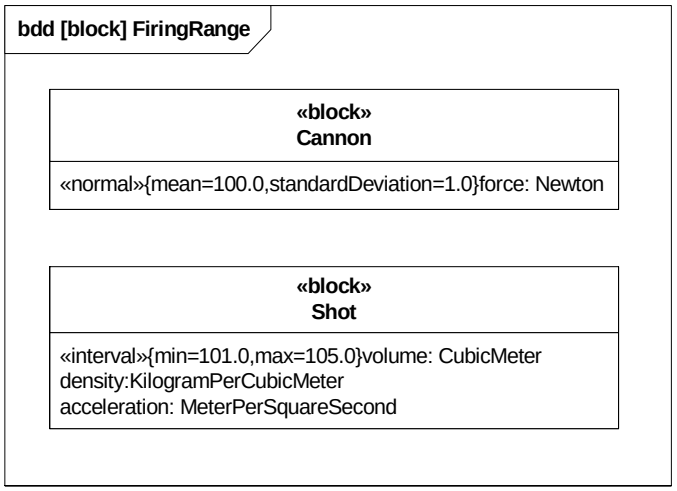


Figure D.15 - Distribution Example

Annex E: Model Interchange

(informative)

E.1 Overview

This annex describes two methods for exchanging SysML models between tools. The first method discussed is XML Metadata Interchange (XMI), which is the preferred method for exchanging models between UML-based tools. The second approach describes the use of ISO 10303-233 Application Protocol: Systems engineering (AP233), which is one of the series of STEP (Standard for the Exchange of Product Model Data) engineering data exchange standards. Other model interchange approaches are possible, but the ones described in this annex are expected to be the primary ones supported by SysML.

E.2 Context for Model Interchange

Developing today's complex systems typically requires engineering teams that are distributed in time and space and that are often composed of many companies, each with their own culture, methods, and tools. Effective collaboration requires agreement on, and a thorough understanding of, the various work assignments and resulting artifacts.

Many of these artifacts pertain to shared engineering data (e.g., requirements, system structural and behavioral models, verification & validation) that transcend the entire life cycle of the system of interest and are the basis for important systems engineering considerations and decisions. So it is critical that the system information contained in these artifacts and information models be accurately captured and readable by all appropriate team members in a timely manner.

Today, this information resides in an array of tools where each is only concerned with a portion of systems engineering data and can't share its data with other tools because they only understand their own native schema. To mitigate this situation, collaborating organizations are usually forced to either adopt a common set of tools or develop a unique, bidirectional interface between many of the tools that each organization uses. This can be an expensive and untimely approach to data exchange between team members. So there is a need to define standardized approaches for model interchange between the different data schemas in use.

E.3 XMI Serialization of SysML

UML 2.0 is formally defined using the OMG Meta Object Facility (MOF). MOF can be considered a language for specifying modeling languages. The OMG XML Metadata Interchange (XMI) 2.1 standard specifies an XML-based interchange format for any language modeled using MOF. This results in a standard, convenient format for serializing UML user models as XMI files for interchange between UML tools. The XMI specification also includes rules for generating an XML Schema that can be used for basic validation of the structure of those UML user model XMI files.

The UML language includes an extension mechanism called UML Profiles. UML Profiles are themselves defined as UML models (MOF is not used). However, their intent is to specify extensions to the UML language semantics in much the same way one could extend the UML language by adding to the MOF definition of UML. As UML Profiles are valid UML models, XMI does provide a mechanism for exchanging the UML Profiles between UML tools. However, as they are extensions to concepts defined in the UML language itself, the definition of a UML Profile refers to the UML language definitions. An XMI 2.1 representation of the SysML profile (i.e., the UML Profile for SysML), as well as XMI 2.1 representations of Model Libraries defined by SysML, are provided as support documents to this specification. As with UML, XMI provides a convenient serialized format for model interchange between SysML tools and basic validation of those files using an XML Schema as well.

The namespace for the standard profile is: http://schema.omg.org/spec/SysML/20090817/SysML-profile.xmi.

E.4 SysML Model Interchange Using AP233

AP233 is a data exchange standard designed to support the exchange of systems engineering data between the many and varied software tools that systems engineers use in the course of their work. Data from systems modeling tools is included in the scope of AP233, in fact, requirements for AP233 and SysML have been largely aligned by the OMG and the ISO teams working together and in close cooperation with the INCOSE Model Driven System Design working group.

E.4.1 Scope of AP233

AP233 is not a graphical modeling language, but specifies data exchange mechanisms to support the exchange of data between Engineering Tools that generate or consume systems engineering data. Figure E.1 illustrates the overlaps between the types of data that can be exchanged by a tool that supports the AP233 data exchange mechanisms, and the type of data that is generated or consumed by a SysML modeling tool. In general, there is considerable overlap indicating the potential support that AP233 can provide as a data exchange standard for SysML modeling tools.

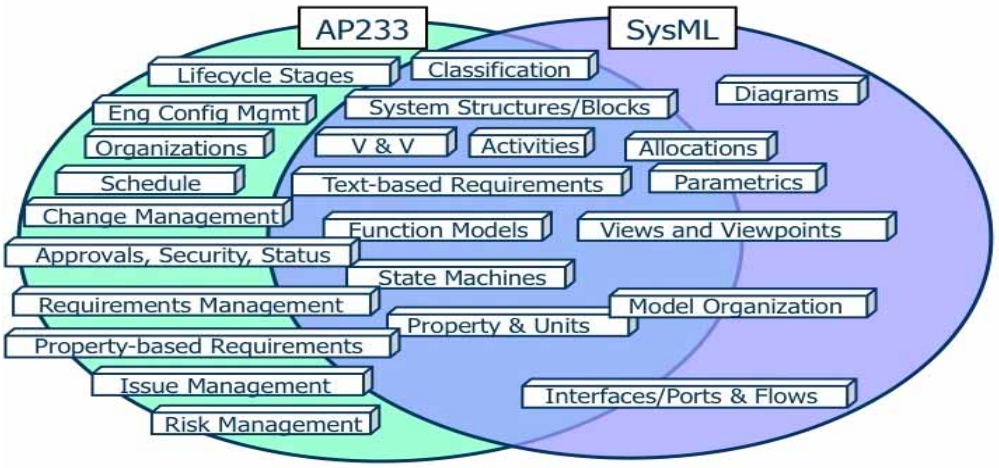


Figure E.1 - SysML/AP233 Data Overlaps

AP233 includes support for assigning program management information as well as system modeling information to systems engineering data.

Program management capabilities include issue management, risk management and aspects of project management such as project breakdown, project resource information, organization structure, schedule and work structure.

System modeling capabilities include requirements and requirements allocation, trade studies with measures of effectiveness, interface to analysis, function-based behavior, state-based behavior, system hierarchies for the design system, the realized system and all interfaces.

Additional information about AP233 can be found at http://www.ap233.org/.

E.4.2 STEP Architecture

AP233 is standardized under ISO Technical Committee 184 (Industrial Automation Systems and Integration), Subcommittee 4 (Industrial Data). AP233 is part of the family of ISO 10303 standards, referred to as STEP, that include standardized models and infrastructure for the exchange of product model data.

The STEP architecture is modular. This enables the component information models to be reused across disciplines and life-cycle stages in different application protocols, which are the models used for implementation. STEP models are written using the ISO 10303-11 EXPRESS language.

STEP also standardizes a series of implementation methods: a text file structure (ISO 10303-21), a data access interface (ISO 10303-22) and an XML file format (ISO 10303-28). The data access interface has bindings that provide standardized APIs for accessing EXPRESS-based data in various programming languages. A conforming STEP implementation is the combination of a STEP application protocol and one or more of the implementation methods.

The scope of STEP is very large and a number of data exchange standards (e.g., geometry, product life-cycle support, structural, electrical, and engineering analysis) have been in wide use in industry for more than 15 years. Support for several systems engineering viewpoints within the scope of AP233 are shared with other application protocols. Since AP233 is part of STEP, it is easy to relate systems engineering data to that of other engineering disciplines over the life cycle of a system and to related product models.

For more information on the STEP architecture see the ISO TC184/SC4 Industrial Data subcommittee web page at http://www.tc184-sc4.org.

E.4.3 EXPRESS

AP233, like all STEP application protocols, is defined using the EXPRESS modeling language. EXPRESS is a precise text-based information modeling language with a related graphical representation called EXPRESS-G.

An example of the text-based format follows:

```
SCHEMA Ap233_systems_engineering_arm_excerpt;
ENTITY Product;
  id : STRING;
  name : STRING;
  description : OPTIONAL STRING;
END_ENTITY;

ENTITY Product_version;
  id : STRING;
  description : OPTIONAL STRING;
  of_product : Product;
END_ENTITY;

ENTITY Product_view_definition;
  id                : OPTIONAL STRING;
  name              : OPTIONAL STRING;
  additional_characterization      : OPTIONAL STRING;
  initial_context   : View_definition_context;
  additional_contexts   : SET [0:?] OF View_definition_context;
  defined_version   : Product_version;
WHERE
  WR1: NOT (initial_context IN additional_contexts);
  WR2: EXISTS(id) OR (TYPEOF(SELF\Product_view_definition) <> TYPEOF(SELF));
END_ENTITY;

ENTITY View_definition_context;
  application_domain     : STRING;
  life_cycle_stage  : STRING;
  description       : OPTIONAL STRING;
WHERE
```

```
  WR1: (SIZEOF (USEDIN(SELF, 'AP233_SYSTEMS_ENGINEERING_ARM_EXCERPT.' +
    'PRODUCT_VIEW_DEFINITION.INITIAL_CONTEXT')) > 0) OR
    (SIZEOF (USEDIN(SELF, 'AP233_SYSTEMS_ENGINEERING_ARM_EXCERPT.' +
    'PRODUCT_VIEW_DEFINITION.ADDITIONAL_CONTEXTS')) > 0);
END_ENTITY;

ENTITY System
  SUBTYPE OF (Product);
END_ENTITY;

ENTITY System_version
  SUBTYPE OF (Product_version);
  SELF\Product_version.of_product : System;
END_ENTITY;

ENTITY System_view_definition
SUBTYPE OF (Product_view_definition);
  SELF\Product_view_definition.defined_version : System_version;
END_ENTITY;

END_SCHEMA;
```

EXPRESS expressions are similar in nature to OCL expressions and the two languages have similar expressiveness. EXPRESS has also been approved as an OMG standard with a September 2009 publication of Version 1.0 of the Reference Metamodel for the EXPRESS Information Modeling Language Specification. This will enable the use of OMG Model Driven Architecture technologies against AP233 and other STEP models written in EXPRESS.

E.4.4 SysML-AP233 Mapping

A formal and standardized mapping between SysML and AP233 is being developed within the OMG. The mapping is a specification for SysML and other tool vendors to implement so that their tools can import from and export to AP233 data exchange files. AP233 usage is aimed primarily at scenarios where SysML data is fed to downstream applications such as those used in manufacturing, life cycle management or systems maintenance. Additional information can be found at the OMG SysML Portal at http://www.omgwiki.org/OMGSysML/.

Annex F: Requirements Traceability

(informative)

The OMG SysML requirements traceability matrix traces this specification to the original source requirements in the UML for Systems Engineering RFP (ad/2003-03-41). The traceability matrix is included by reference in a separate document (ptc/2007-03-09).