

Introduction to the SysML v2 Language

Textual Notation



This is a training presentation on the SysML v2 language.
It is updated as appropriate for each release of the
SysML v2 Pilot Implementation maintained by
the OMG® Systems Modeling Community (SMC)



Release: 2026-03

Copyright © 2019-2026 Model Driven Solutions, Inc.

Licensed under the Creative Commons Attribution 4.0 International License.

*To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/> or
send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.*

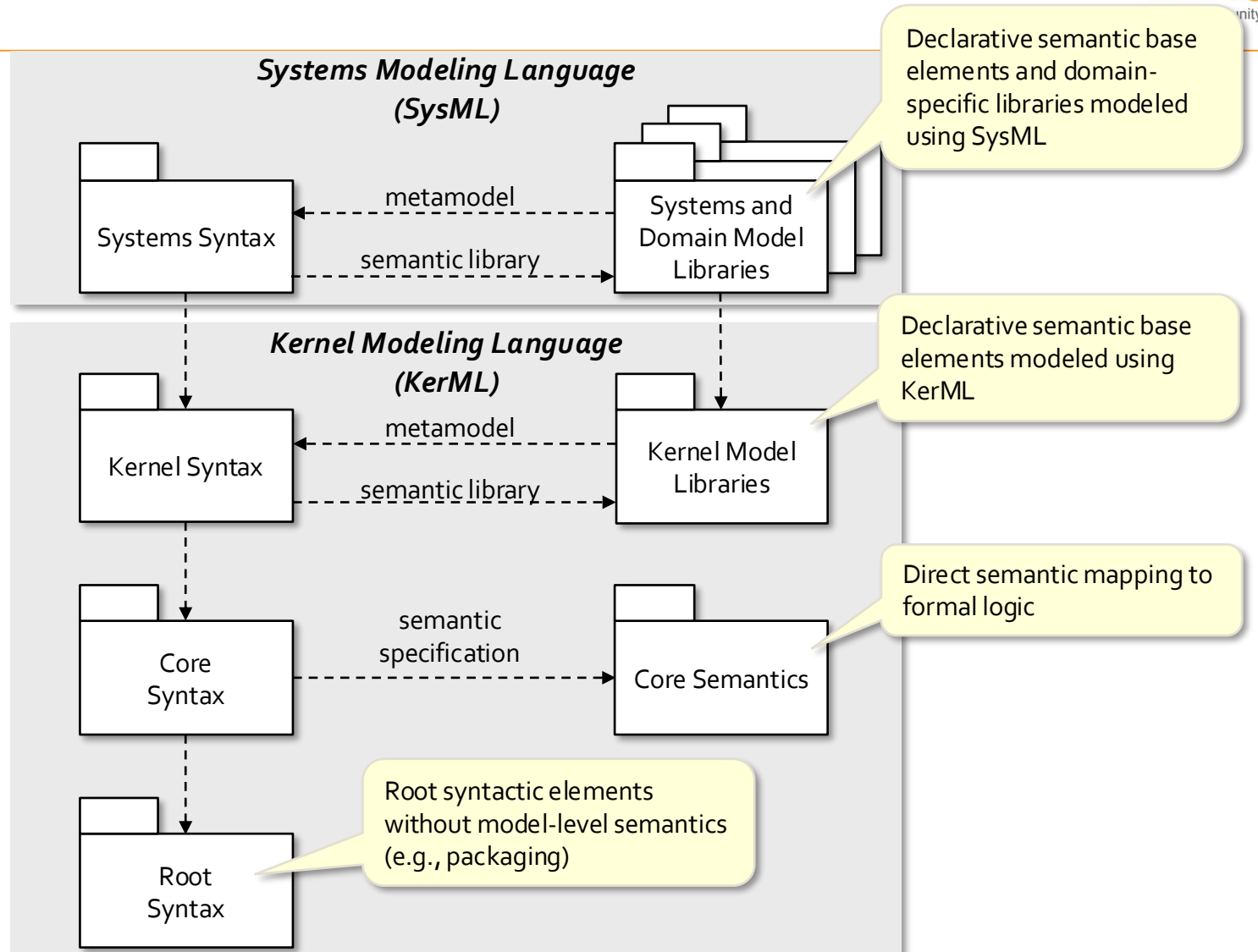


Changes in this Release

- In State Definition and subsequent slides on states, changed “**entry; then ... ;**” to “**first start then ... ;**”.
- Revised use of feature references for Element Import Filtering.

*To find slides that have changed recently, search for
Last changed: 2026-03
(and similarly for earlier releases).*

SysML v2 Language Architecture





Four-Layer Language Architecture



- Root – Root syntactic elements
 - Element, Comment, Documentation, Textual Representation, Namespace
 - Relationship, Annotation, Membership, Import, Dependency
- Core – Fundamental semantic concepts – Formal declarative semantics
 - Type, Classifier, Feature, Multiplicity
 - Feature Membership, Specialization, Conjugation, Type Featuring, Feature Chaining, Feature Inverting, Unioning, Intersecting, Differencing
- Kernel – Foundation for building modeling languages – Kernel semantic library
 - Class, Data Type, Behavior, Function, Metaclass, Step, Expression, Metadata Feature, Package
 - Association, Interaction, Connector, Binding Connector, Succession, Item Flow, Succession Item Flow
- Systems – Modeling language for systems engineering – Domain libraries
 - Definition and Usage of:
 - Attributes, Enumerations, Occurrences, Items, Parts, Ports, Actions, States, Constraints, Requirements, Concerns, Calculations, Cases, Analysis Cases, Verification Cases, Use Cases, Views, Viewpoints, Renderings, Metadata
 - Connections, Interfaces, Allocations, Flows, Succession Flows

Packages – Members

A *package* acts as a *namespace* for its members and a *container* for its owned members.

① A name with spaces or other special characters is surrounded in single quotes.

An *import* adds either a single *imported member* or all the members of an *imported package* to the *importing package*.

The *owned members* of a package are elements directly contained in the package.

```
package 'Package Example' {
  public import ISQ::TorqueValue;
  private import ScalarValues::*;

  part def Automobile;

  alias Car for Automobile;
  alias Torque for ISQ::TorqueValue;
}
```

A package can introduce *aliases* for its owned members or for individual members of other packages.

① A qualified name is a package name (which may itself be qualified) followed by the name of one of its members, separated by `::`.

Packages – Visibility

All members from a public import are visible (*re-exported*) from the importing package. Members from a private import are not.

⚠ Visibility *must* be shown *explicitly* on import declarations. Use private visibility unless re-export is required.

A *private* member is not visible outside the package (but it is visible to subpackages).

Members are *public* by default but can also be marked public explicitly.

```
package 'Package Example' {
  public import ISQ::TorqueValue;
  private import ScalarValues::*;

  private part def Automobile;

  public alias Car for Automobile;
  alias Torque for ISQ::TorqueValue;
}
```

Comments

A *comment* begins with `/*` and ends with `*/`.

A comment can optionally be named.

Import and alias declarations can also be commented.

A *note* begins with `//` and extends to the end of the line. (A multiline note begins with `//*` and ends with `*/`.)

```
package 'Comment Example' {
  /* This is comment, part of the model,
    * annotating (by default) it's owning package. */

  comment Comment1 /* This is a named comment. */

  comment about Automobile
  /* This is an unnamed comment, annotating an
    * explicitly specified element.
    */

  part def Automobile;

  alias Car for Automobile {
    /* This is a comment annotating its owning
      * element.
      */
  }

  // This is a note. It is in the text, but not part
  // of the model.
  alias Torque for ISQ::TorqueValue;
}
```

What the comment annotates can be explicitly specified (it is the owning namespace by default).

Documentation is a special kind of comment that is directly owned by the element it documents.

A documentation comment can be optionally named like a regular comment.

```
package 'Documentation Example' {
  doc /* This is documentation of the owning
      * package.
      */

  part def Automobile {
    doc Document1
      /* This is documentation of Automobile. */
  }

  alias Car for Automobile {
    doc /* This is documentation of the alias. */
  }
  alias Torque for ISQ::TorqueValue;
}
```

Part and Attribute Definitions

A *part definition* is a definition of a class of systems or parts of systems, which are mutable and exist in space and time.

An *attribute definition* is a definition of attributive data that can be used to describe systems or parts.

① Definitions and usages are also namespaces that allow **import**.

```
part def Vehicle {
  attribute mass : ScalarValues::Real;

  part eng : Engine;
  ref part driver : Person;
}

attribute def VehicleStatus {
  private import ScalarValues;

  attribute gearSetting : Integer;
  attribute acceleratorPosition : Real;
}

part def Engine;
part def Person;
```

An *attribute usage* of an *attribute definition*, used here as a feature of the part definition.

A *part usage* is a *composite* feature that is the usage of a part definition.

A *reference part usage* is a *referential* feature that is the usage of a part definition.

An attribute definition may not have composite features. Attribute usages are always referential.

Generalization/Specialization

An *abstract* definition is one whose instances must be members of some specialization.

```
abstract part def Vehicle;
```

```
part def HumanDrivenVehicle specializes Vehicle {  
    ref part driver : Person;  
}
```

```
part def PoweredVehicle :> Vehicle {  
    part eng : Engine;  
}
```

```
part def HumanDrivenPoweredVehicle :>  
    HumanDrivenVehicle, PoweredVehicle;
```

```
part def Engine;  
part def Person;
```

A *specialized* definition defines a subset of the classification of its generalization.

The `:>` symbol is equivalent to the `specializes` keyword.

A specialization can define additional features.

A definition can have multiple generalizations, *inheriting* the features of all general definitions.

```

part def Vehicle {
    part parts : VehiclePart[*];

    part eng : Engine subsets parts;
    part trans : Transmission subsets parts;
    part wheels : Wheel[4] :> parts;
}

abstract part def VehiclePart;
part def Engine :> VehiclePart;
part def Transmission :> VehiclePart;
part def Wheel :> VehiclePart;
    
```

Subsetting asserts that, in any common context, the values of one feature are a subset of the values of another feature.

Subsetting is a kind of generalization between features.

Redefinition

```

part def Vehicle {
    part eng : Engine;
}
part def SmallVehicle :> Vehicle {
    part smallEng : SmallEngine redefines eng;
}
part def BigVehicle :> Vehicle {
    part bigEng : BigEngine :>> eng;
}

part def Engine {
    part cyl : Cylinder[4..6];
}
part def SmallEngine :> Engine {
    part redefines cyl[4];
}
part def BigEngine :> Engine {
    part redefines cyl[6];
}

part def Cylinder;
    
```

A *specialized* definition can *redefine* a feature that would otherwise be inherited, to change its name and/or specialize its type.

The `:>>` symbol is equivalent to the **redefines** keyword.

A feature can also specify *multiplicity*.

① The default multiplicity for parts is **1..1**.

Redefinition can be used to constrain the multiplicity of a feature.

There is shorthand notation for redefining a feature with the same name.

Enumeration Definitions (1)

An *enumeration definition* is a kind of attribute definition that defines a set of *enumerated values*.

```
enum def TrafficLightColor {
    enum green;
    enum yellow;
    enum red;
}

part def TrafficLight {
    attribute currentColor : TrafficLightColor;
}

part def TrafficLightGo specializes TrafficLight {
    attribute redefines currentColor = TrafficLightColor::green;
}
```

The values of an attribute usage defined by an enumeration definition are limited to the defined set of enumerated values.

This shows an attribute being *bound* to a specific value (more on binding later).

Enumeration Definitions (2)

```

attribute def ClassificationLevel {
    attribute code : String;
    attribute color : TrafficLightColor;
}

enum def ClassificationKind specializes ClassificationLevel {
    unclassified {
        :>> code = "uncl";
        :>> color = TrafficLightColor::green;
    }
    confidential {
        :>> code = "conf";
        :>> color = TrafficLightColor::yellow;
    }
    secret {
        :>> code = "secre";
        :>> color = TrafficLightColor::red;
    }
}

enum def GradePoints :> Real {
    A = 4.0;
    B = 3.0;
    C = 2.0;
    D = 1.0;
    F = 0.0;
}

```

The **enum** keyword is optional when declaring enumerated values.

An enumeration definition can contain nothing but declarations of its enumerated values. However, it can specialize a regular attribute definition and inherit nested features.

⚠ An enumeration definition cannot specialize another enumeration definition.

The nested features can then be bound to specific values in each of the enumerated values.

Enumerated values can also be bound directly to specific values of a specialized attribute definition or data type.

Parts (1)

Parts can be specified outside the context of a specific part definition.

```
// Definitions
part def Vehicle {
    part eng : Engine;
}
part def Engine {
    part cyl : Cylinder[4..6];
}
part def Cylinder;

// Usages
part smallVehicle : Vehicle {
    part redefines eng {
        part redefines cyl[4];
    }
}
part bigVehicle : Vehicle {
    part redefines eng {
        part redefines cyl[6];
    }
}
```

The **defined by** relationship is a kind of generalization.

Parts inherit properties from their definitions and can redefine them, to any level of nesting.

Parts (2)

```
// Definitions
part def Vehicle;
part def Engine;
part def Cylinder;

// Usages
part vehicle : Vehicle {
    part eng : Engine {
        part cyl : Cylinder[4..6];
    }
}
part smallVehicle :> vehicle {
    part redefines eng {
        part redefines cyl[4];
    }
}
part bigVehicle :> vehicle {
    part redefines eng {
        part redefines cyl[6];
    }
}
```

Composite structure can be specified entirely on parts.

A part can specialize another part.

An *item definition* defines a class of things that exist in space and time but are not necessarily considered "parts" of a system being modeled.

① All parts can be treated as items, but not all items are parts. The design of a system determines what should be modeled as its "parts".

```

item def Fuel;
item def Person;

part def Vehicle {
  attribute mass : Real;

  ref item driver : Person;

  part fuelTank {
    item fuel: Fuel;
  }
}

```

① The default multiplicity for attributes and items is also [1..1](#).

A system model may reference discrete items that interact with or pass through the system.

① An item is continuous if any portion of it in space is the same kind of thing. A portion of fuel is still fuel. A portion of a person is generally no longer a person.

Items may also model continuous materials that are stored in and/or flow between parts of a system.

Connections (1)

Cross multiplicities constrain the number of connections, rather than end values. E.g., for every **TireBead** there must be a **PressureSeat** connection to exactly one **TireMountingRim**.

A *connection* is a usage of a connection definition, which connects two other features.

If a connection definition is not specified, a generic connection definition (called **Connection**) is used.

```
connection def PressureSeat {
  end [1] part bead : TireBead;
  end [1] part mountingRim: TireMountingRim;
}

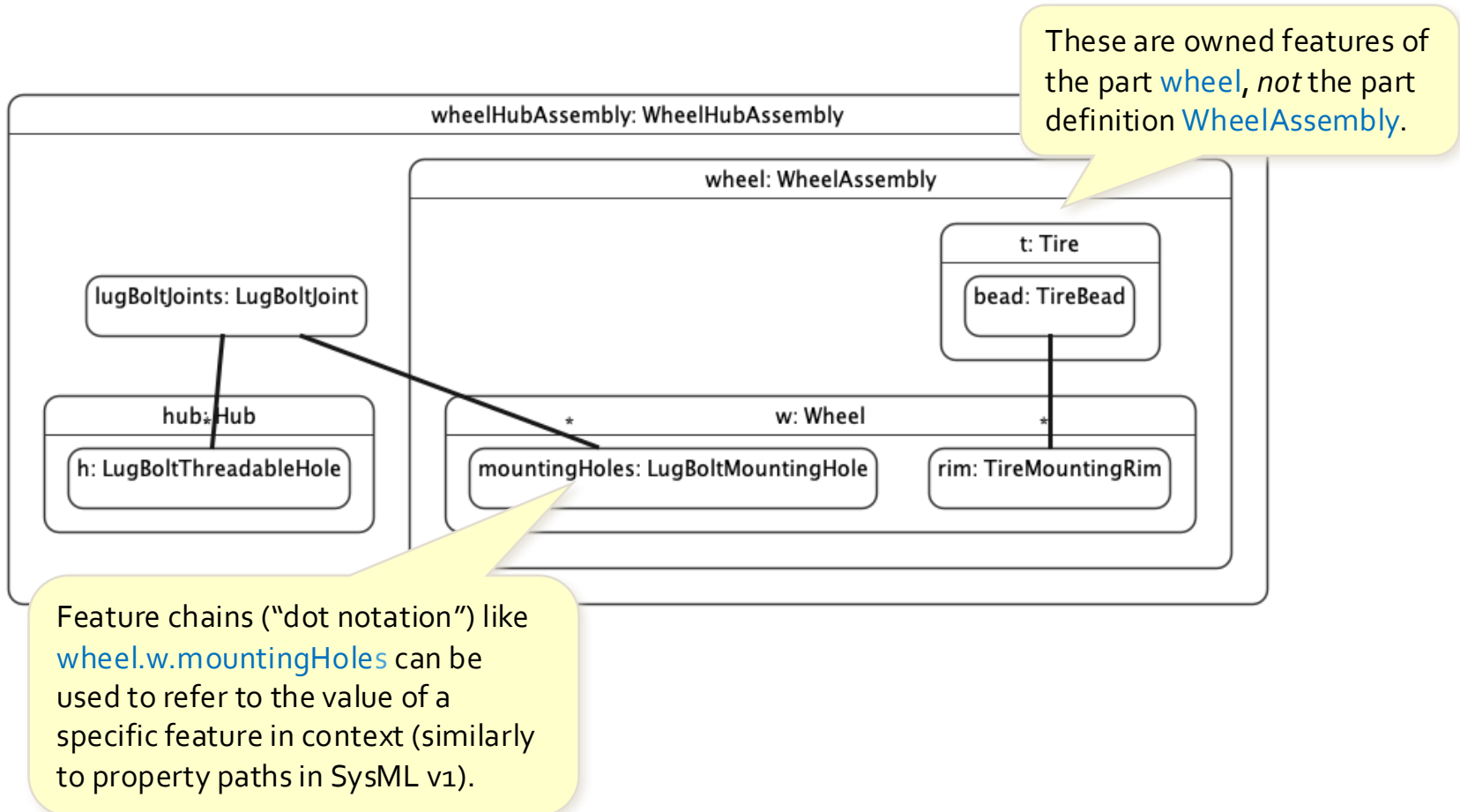
part wheelHubAssembly : WheelHubAssembly {
  part wheel : WheelAssembly[1] {
    part t : Tire[1] {
      part bead : TireBead[2];
    }
    part w: Wheel[1] {
      part rim : TireMountingRim[2];
      part mountingHoles : LugBoltMountingHole[5];
    }
    connection : PressureSeat
      connect bead references t.bead
      to mountingRim references w.rim;
  }

  part lugBoltJoints : LugBoltJoint[0..5];
  part hub : Hub[1] {
    part h : LugBoltThreadableHole[5];
  }
  connect [0..1] lugBoltJoints to [1] wheel.w.m mountingHoles;
  connect [0..1] lugBoltJoints to [1] hub.h;
}
```

A *connection definition* is a part definition whose usages are *connections* between its ends.

Reference subsetting (**references** or **::>**) relates feature to a connector ends. "Dot" notation specifies paths to nested features.

Connections (2)



Ports

A *port definition* defines features that can be made available via ports. (Replaces interface blocks in SysML v1).

① Directed features are always referential, so it is not necessary to explicitly use the **ref** keyword.

A *port* is a connection point through which a part definition makes some of its features available in a limited way. (Like a proxy port in SysML v1.)

```
port def FuelOutPort {
    attribute temperature : Temp;
    out item fuelSupply : Fuel;
    in item fuelReturn : Fuel;
}

port def FuelInPort {
    attribute temperature : Temp;
    in item fuelSupply : Fuel;
    out item fuelReturn : Fuel;
}

part def FuelTankAssembly {
    port fuelTankPort : FuelOutPort;
}

part def Engine {
    port engineFuelPort : FuelInPort;
}
```

Ports may have attribute and reference features. A feature with a direction (**in**, **out** or **inout**) is a *directed feature*.

Two ports are *compatible* for connection if they have directed features that match with inverse directions.

Port Conjugation

Every port definition has an implicit *conjugate* port definition that reverses input and output features. It has the name of the original definition with ~ prepended (e.g., '~FuelPort').

```
port def FuelPort {
  attribute temperature : Temp;
  out item fuelSupply : Fuel;
  in item fuelReturn : Fuel;
}

part def FuelTankAssembly {
  port fuelTankPort : FuelPort;
}

part def Engine {
  port engineFuelPort : ~FuelPort;
}
```

Using a ~ symbol on the port type is a shorthand for using the conjugate port definition (e.g., `FuelPort::~~FuelPort`).

An *interface definition* is a connection definition whose ends are ports.

① The default cross multiplicity is `0..*`.

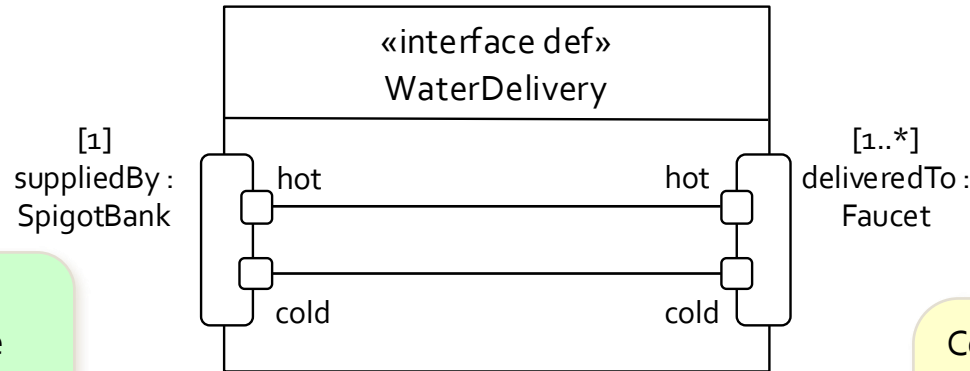
An *interface usage* is a connection usage defined by an interface definition, connecting two compatible ports.

```
interface def FuelInterface {
    end port supplierPort : FuelOutPort;
    end port consumerPort : FuelInPort;
}

part vehicle : Vehicle {
    part tankAssy : FuelTankAssembly;
    part eng : Engine;

    interface : FuelInterface connect
        supplierPort ::> tankAssy.fuelTankPort to
        consumerPort ::> eng.engineFuelPort;
}
```

Interface Decomposition



① Every **suppliedBy SpigotBank** must have one or more connections to **deliveredTo Faucets**.

Cross multiplicities correspond to navigating *across* connections.

```
interface def WaterDelivery {
  end [1] port suppliedBy : SpigotBank {
    port hot : Spigot;
    port cold : Spigot;
  }
  end [1..*] port deliveredTo : Faucet {
    port hot : FaucetInlet;
    port cold : FaucetInlet;
  }

  connect suppliedBy.hot to deliveredTo.hot;
  connect suppliedBy.cold to deliveredTo.cold;
}
```

Connection ends specify the *participants* in the connection. They always have multiplicity 1..1 relative to the connection definition.

Connection uses interconnect specific participants.

Binding Connections

```

part tank : FuelTankAssembly {
  port redefines fuelTankPort {
    out item redefines fuelSupply;
    in item redefines fuelReturn;
  }

  bind fuelTankPort.fuelSupply = pump.pumpOut;
  bind fuelTankPort.fuelReturn = tank.fuelIn;

  part pump : FuelPump {
    out item pumpOut : Fuel;
    in item pumpIn : Fuel;
  }

  part tank : FuelTank {
    out item fuelOut : Fuel;
    in item fuelIn : Fuel;
  }
}

```

A *binding connection* is a connection that asserts the *equivalence* of the connected features (i.e., they have equal values in the same context).

Usages on parts can also have direction (and are automatically referential).

```

part tank : FuelTankAssembly {
  port redefines fuelTankPort {
    out item redefines fuelSupply = pump.pumpOut;
    in item redefines fuelReturn;
  }

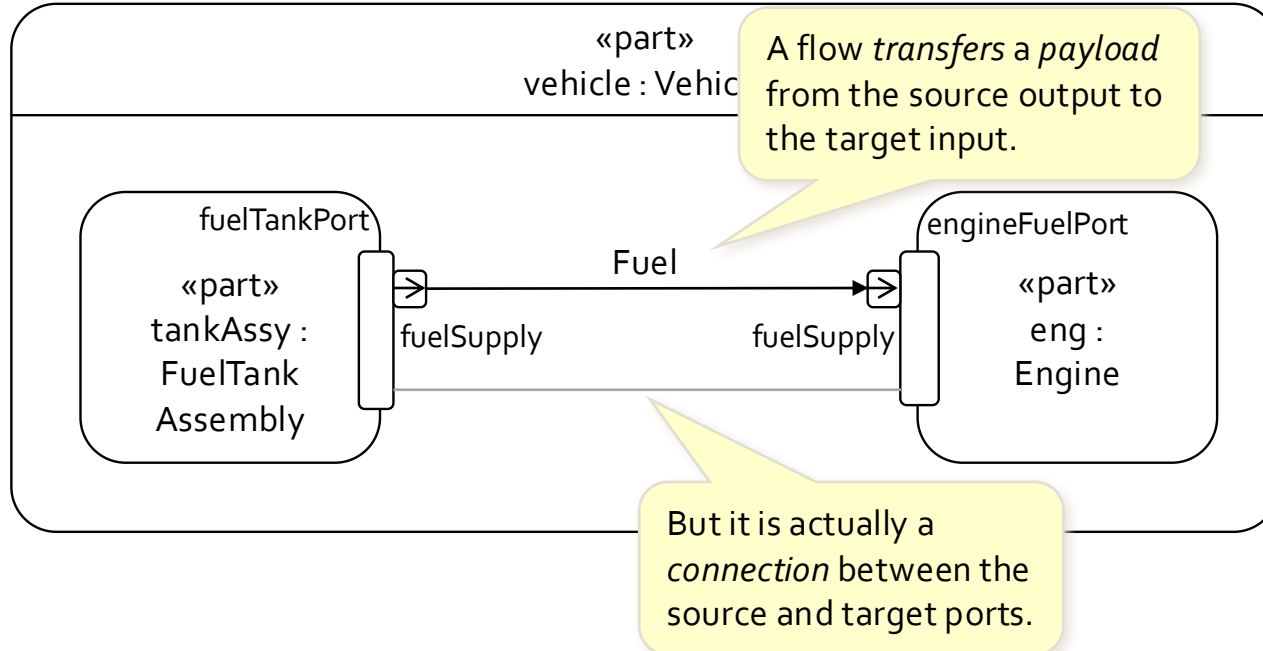
  part pump : FuelPump {
    out item pumpOut : Fuel;
    in item pumpIn : Fuel;
  }

  part tank : FuelTank {
    out item fuelOut : Fuel;
    in item fuelIn : Fuel = fuelTankPort.fuelReturn;
  }
}

```

The *feature value* notation combines the definition of a feature with a binding connection.

⚠ This is a binding, not an initial or default value.



- ① A flow is *streaming* if the transfer is ongoing between the source and target, as opposed to happening once after the source generates its output and before the target consumes its input.

A *flow usage* is a transfer of some *payload* from an output of a source port to an input of a target port.

❶ Specifying the payload type (e.g., “*of Fuel*”) is optional.

```
part vehicle : Vehicle {
  part tankAssy : FuelTankAssembly;
  part eng : Engine;

  flow of Fuel
    from tankAssy.fuelTankPort.fuelSupply
    to eng.engineFuelPort.fuelSupply;

  flow of Fuel
    from eng.engineFuelPort.fuelReturn
    to tankAssy.fuelTankPort.fuelReturn;
}
```

Flows in Interfaces

```

interface def FuelInterface {
  end supplierPort : FuelOutPort;
  end consumerPort : FuelInPort;

  flow supplierPort.fuelSupply to consumerPort.fuelSupply;
  flow consumerPort.fuelReturn to supplierPort.fuelReturn;
}

part vehicle : Vehicle {
  part tankAssy : FuelTankAssembly;
  part eng : Engine;

  interface : FuelInterface connect
    supplierPort ::> tankAssy.fuelTankPort to
    consumerPort ::> eng.engineFuelPort;
}

```

Flows can be defined within an interface definition.

The flows are established when the interface is used.

A *flow definition* is used to assert that a flow must exist between the instances of part definitions.

The definition can (optionally) constrain the payload transferred by usages.

```
flow def FuelFlow {
  ref :>> payload : Fuel;
  end port fuelOut : FuelOutPort;
  end port fuelIn : FuelInPort;
}
```

Note that the default end multiplicities on this connection are 1 to 1.

This usage realizes the required flow between the `tankAssy.fuelTankPort` and the `eng.engineFuelPort`.

```
part vehicle : Vehicle {
  part tankAssy : FuelTankAssembly;
  part eng : Engine;
```

The payload should be consistent with what is declared in the flow definition (if any).

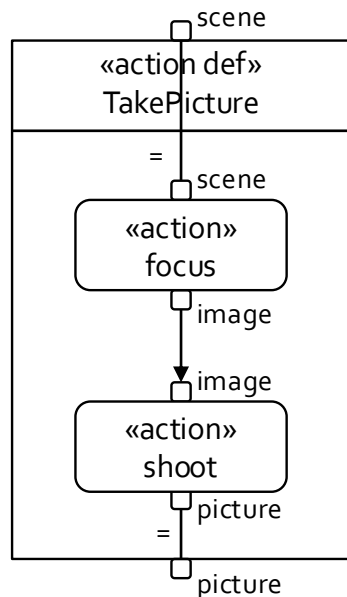
```
  flow : FuelFlow of Fuel
    from tankAssy.fuelTankPort.fuelSupply
    to eng.engineFuelPort.fuelSupply;
}
```

The connection is defined to be between the ports, but the transfer is still from an output to an input.

Action Definition

An *action definition* is a definition of some action to be performed.

Directed features of an action definition are considered to be action *parameters*.



```

action def Focus{ in scene : Scene; out image : Image; }
action def Shoot{ in image : Image; out picture : Picture; }

action def TakePicture {
  in scene : Scene;
  out picture : Picture;

  bind focus.scene = scene;

  action focus : Focus { in scene; out image;

  flow focus.image to shoot.image;

  action shoot : Shoot { in image; out picture; }

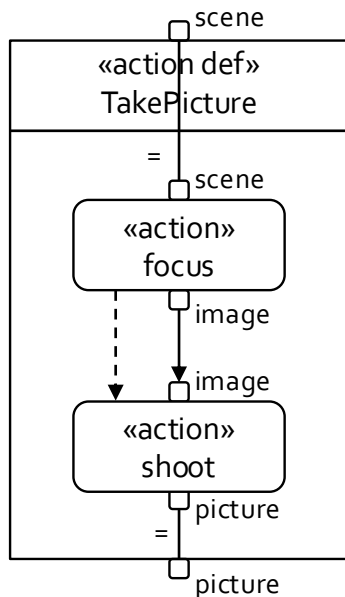
  bind shoot.picture = picture;
}
    
```

An *action* is a usage of an action definition performed in a specific context.

A flow can be used to transfer items between actions.

An action has parameters corresponding to its action definition.

Action Succession (1)



```

action def Focus{ in scene : Scene; out image : Image; }
action def Shoot{ in image : Image; out picture : Picture; }

action def TakePicture {
    in scene : Scene;
    out picture : Picture;

    bind focus.scene = scene;

    action focus : Focus { in scene; out image; }

    flow focus.image to shoot.image;

    first focus then shoot;

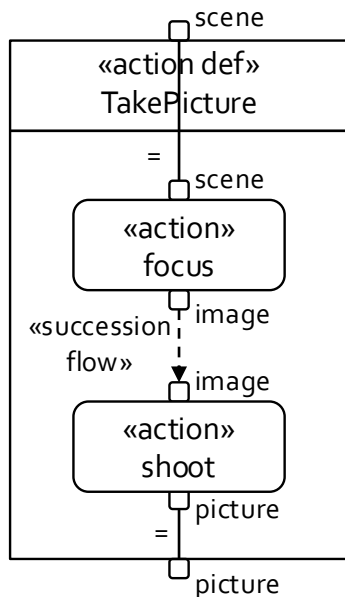
    action shoot : Shoot { in image; out picture; }

    bind shoot.picture = picture;
}

```

A *succession* asserts that the first action must complete before the second can begin.

Action Succession (2)



```

action def Focus{ in scene : Scene; out image : Image; }
action def Shoot{ in image : Image; out picture : Picture; }

action def TakePicture {
    in scene : Scene;
    out picture : Picture;

    bind focus.scene = scene;

    action focus : Focus { in scene; out image; }

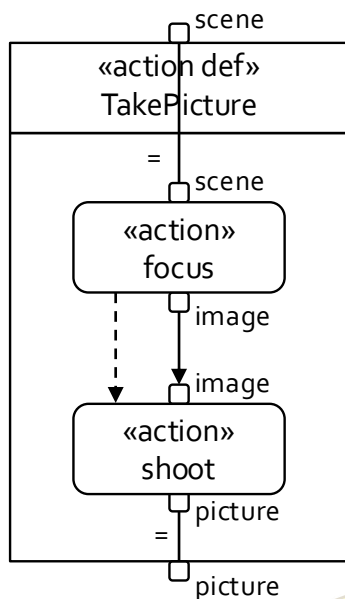
    succession flow focus.image to shoot.image;

    action shoot : Shoot { in image; out picture; }

    bind shoot.picture = picture;
}
    
```

A *succession flow* requires the first action to finish producing its output before the second can begin consuming it.

Action Notation Shorthands



This is a shorthand for a succession between the lexically previous action (**focus**) and this action (**shoot**).

```

action def Focus{ in scene : Scene; out image : Image; }
action def Shoot{ in image : Image; out picture : Picture; }

```

```

action def TakePicture {
    in scene : Scene;

    action focus : Focus {
        in scene = TakePicture::scene;
        out image;
    }

```

This qualified name refers to the **scene** parameter of **TakePicture**, rather than the parameter of **focus** with the same name.

This is the same shorthand for binding used previously.

```

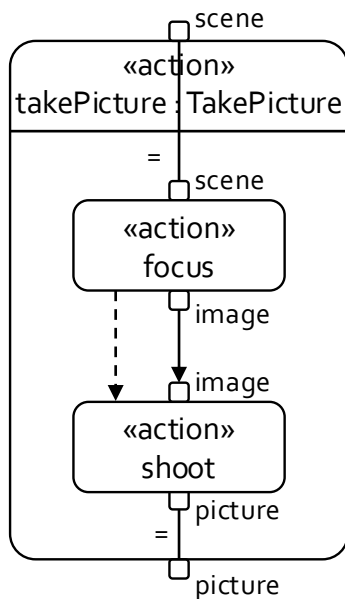
    flow focus.image to shoot.image;

    then action shoot : Shoot {
        in image;
        out picture;
    }

    out picture : Picture = shoot.picture;
}

```

Action Decomposition



```

action def Focus{ in scene : Scene; out image : Image; }
action def Shoot{ in image : Image; out picture : Picture; }
action def TakePicture{in scene : Scene; out picture : Picture;;}

```

```

action takePicture : TakePicture
    in scene;

```

An action usage can also be directly decomposed into other actions.

```

    action focus : Focus {
        in scene = takePicture::scene;
        out image;
    }

```

```

    flow focus.image to shoot.image;

```

```

    then action shoot : Shoot {
        in image;
        out picture;
    }

```

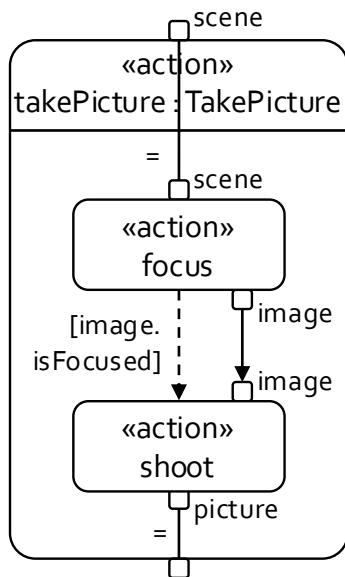
```

    out picture = shoot.picture;
}

```

`takePicture` is a usage, so dot notation could be used here, but it is unnecessary because `scene` is in an outer scope, not nested.

Conditional Succession (1)



```

action takePicture : TakePicture {
    in scene;

    action focus : Focus {
        in scene = takePicture::scene;
        out image;
    }

    first focus
        if focus.image.isFocused then shoot;

    flow focus.image to shoot.image;

    action shoot : Shoot {
        in image;
        out picture;
    }

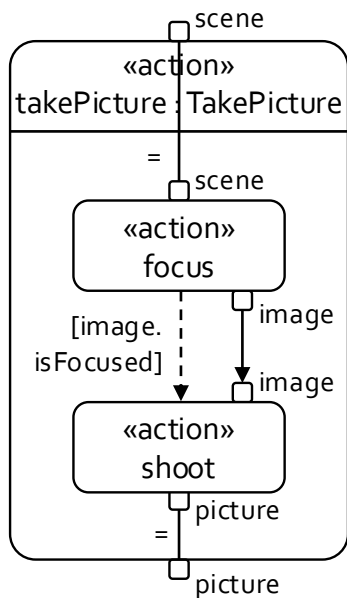
    out picture = shoot.picture;
}

```

A *conditional succession* asserts that the second action must follow the first only if a *guard* condition is true. If the guard is false, succession is not possible.

⚠ Note that, currently, the target action must be explicitly named (no shorthand).

Conditional Succession (2)



```

action takePicture : TakePicture {
  in scene;
  out picture;

```

```

  action focus : Focus {
    in scene = takePicture::scene;
    out image
  }

```

This is a shorthand for a conditional succession following the lexically previous action.

```

  if focus.image.isFocused then shoot;

```

```

  flow focus.image to shoot.image;

```

```

  action shoot : Shoot {
    in image;
    out picture;
  }

```

```

  out picture = shoot.picture;

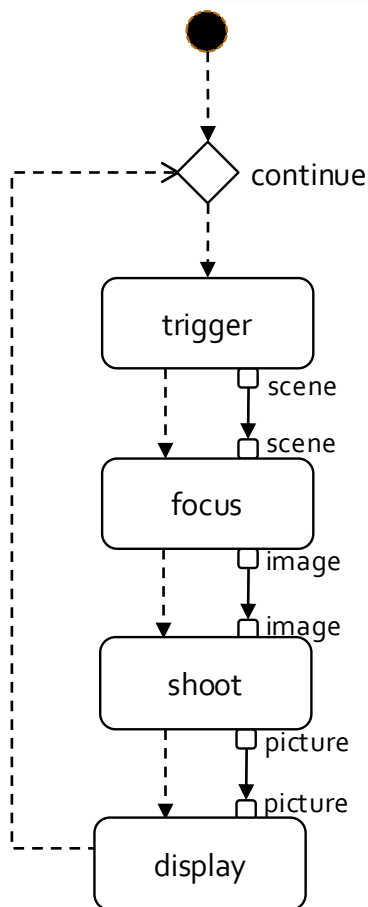
```

```

}
```


Merge Nodes

References the start of the action as the source of the initial succession.



```

action takePicture : TakePicture {
  first start;
  then merge continue;

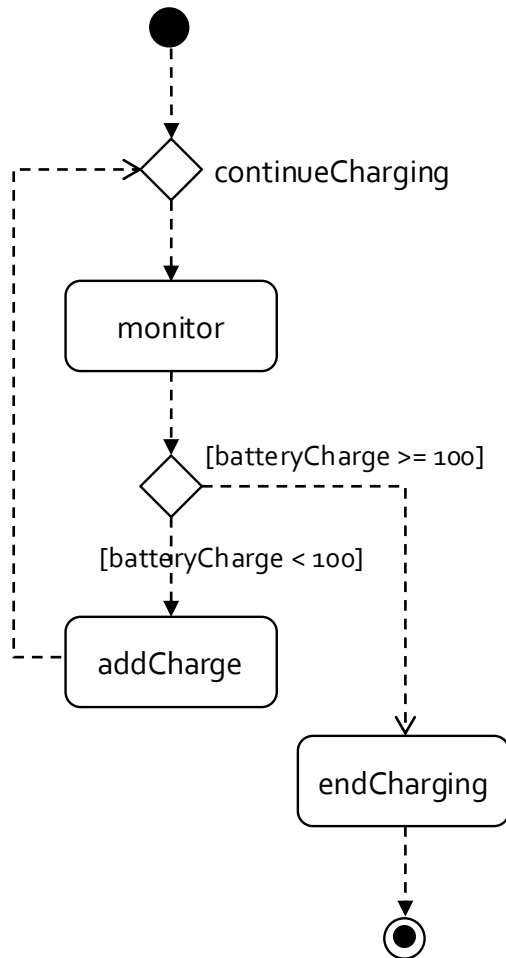
  then action trigger { out item;
  flow trigger.scene to focus.scene;
  then action focus : Focus {
    in scene;
    out image;
  }
  flow focus.image to shoot.image;
  then action shoot : Shoot {
    in image;
    out picture;
  }
  flow shoot.picture to display.picture;
  then action display {
    in picture;
  }

  then continue;
}
  
```

A merge node waits for exactly one predecessor to happen before continuing.

References the merge node named "continue" as the target of the succession.

Decision Nodes



```

action def ChargeBattery {
    first start;

    then merge continueCharging;
    then action monitor : MonitorBattery {
        out batteryCharge : Real;
    }

    then decide;
        if monitor.batteryCharge < 100 then addCharge;
        if monitor.batteryCharge >= 100 then endCharging;

    action addCharge : AddCharge {
        in charge = monitor.batteryCharge;
    }
    then continueCharging;

    action endCharging : EndCharging;

    then done;
}
    
```

A decision node (**decide** keyword) chooses *exactly one successor* to happen after it.

A decision node is typically followed by one or more conditional successions (the last "if...then" can be replaced by "else").

References the end of the action as the target of a succession.

Control Structures

① Control-structure actions provide a structured alternative to control nodes for conditionals and looping.

A *loop action* repeatedly performs a *body action*.

An *if action* performs its body if a Boolean condition is true. (It can also have an *else* part, which is performed if the condition is false.)

until introduces a Boolean condition that terminates the loop when it is true.

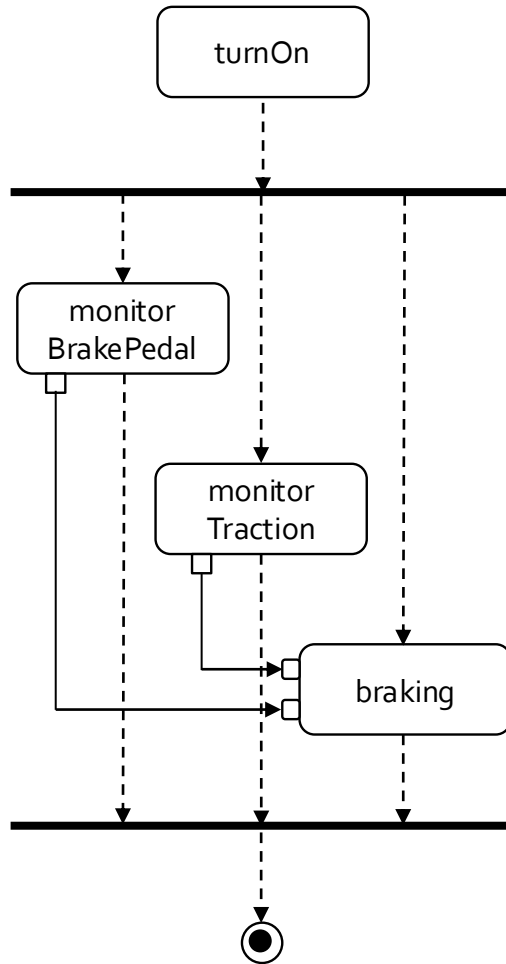
① A loop action can also begin with *while* (instead of *loop*), introducing a Boolean condition that terminates the loop when it is false.

```
action def ChargeBattery {
  loop action charging {
    action monitor : MonitorBattery {
      out charge;
    }
    then if monitor.charge < 100 {
      action addCharge : AddCharge {
        in charge = monitor.charge;
      }
    }
  } until charging.monitor.charge >= 100;
  then action endCharging : EndCharging;
  then done;
}
```

This is the *body action*, which is given the name *charging*. (The default name is *body*.)

⚠ *if* used in this way represents an action, not a succession.

Fork and Join Nodes



A **fork** node enables *all* its successors to happen after it.

The source for *all* these successions is the **fork** node.

```

action def Brake {
  action turnOn;
  then fork;
    then monitorBrakePedal;
    then monitorTraction;
  then braking;

  action monitorBrakePedal : MonitorBrakePedal {
    out brakePressure; }
  then joinNode;
  action monitorTraction : MonitorTraction {
    out modulationFrequency; }
  then joinNode;

  flow monitorBrakePedal.brakePressure to
    braking.breakPressure;
  flow monitorTraction.modulationFrequency to
    braking.modulationFrequency;
  action braking : Braking {
    in brakePressure;
    in modulationFrequency; }
  then joinNode;

  join joinNode;
  then done;
}

```

A **join** node waits for *all* its predecessors to happen before continuing.

Performed Actions

perform identifies the owner as the performer of an action.

This shorthand simply identifies the performed action owned elsewhere without renaming it locally.

```

part camera : Camera {
    perform action takePhoto[*] ordered
    references takePicture;

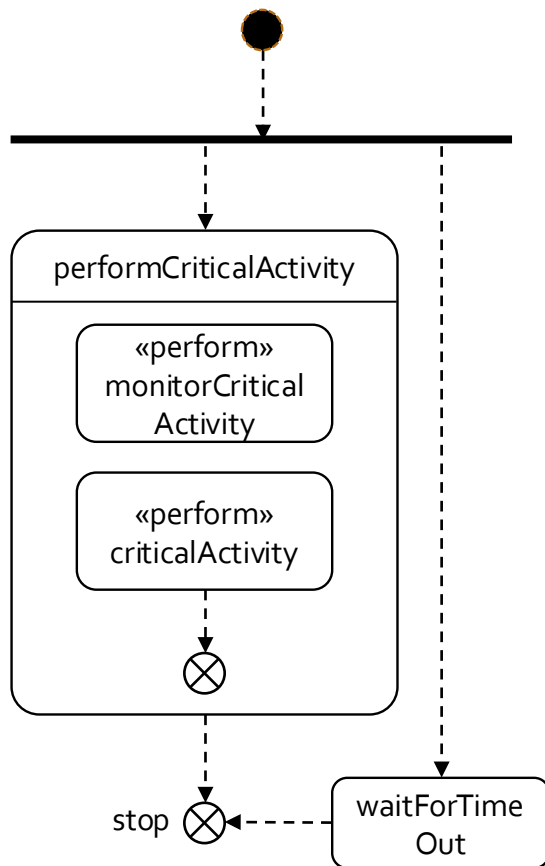
    part f : AutoFocus {
        perform takePhoto.focus;
    }

    part i : Imager {
        perform takePhoto.shoot;
    }
}
    
```

takePhoto is the local name of the *performing* action.

The performing action **references** the *performed* action **takePicture**.

Terminate Actions (1)



```

action def MonitoredActivity {
  first start;

  then fork;
    then performCriticalActivity;
    then waitForTimeOut;

  action performCriticalActivity {
    perform monitorCriticalActivity;

    perform criticalActivity;
    then terminate;
  }
  then stop;

  action waitForTimeOut;
  then stop;

  action stop terminate;
}
  
```

`monitorCriticalActivity` and `criticalActivity` are preformed concurrently.

A *terminate* action ends its immediately containing action. In this case, `performCriticalActivity` terminates even if `monitorCriticalActivity` is still ongoing.

This terminates `MonitoredActivity` when either `performCriticalActivity` or `waitForTimeOut` completes.

Terminate Actions (2)

```

part def Processor {
  ref action workflowProcess : WorkflowProcess;

  action internalProcess {
    // ...
  }
}

action terminateProcessing {
  in processor : Processor;

  terminate processor.workflowProcess;

  terminate processor;
}

```

A terminate action can have an argument that identifies the action to be terminated (instead of the default to the containing action).

A terminate action can also be used to terminate ("destroy") a part.

⚠ A composite action is automatically terminated when its containing part is terminated, but a referential action is not. That is why `workflowProcess` was terminated explicitly in this example, but not `internalProcess`.

Assignment Actions

A *for loop action* repeatedly performs its body, setting a *loop variable* to successive values from the result of a *sequence expression*.

An action may also be performed by another action (similar to a "call").

An *assignment action* sets the value of a feature to the result of an expression at the time of performance of the assignment.

```

action def ComputeMotion {
  in attribute powerProfile :> ISQ::power[*];
  in attribute vehicleMass :> ISQ::mass;
  in attribute initialPosition :> ISQ::length;
  in attribute initialSpeed :> ISQ::speed;
  in attribute deltaT :> ISQ::time;
  out attribute positions :> ISQ::length[*] := ( );

  private attribute position := initialPosition;
  private attribute speed := initialSpeed;

  for vehiclePower in powerProfile {
    perform action dynamics : StraightLineDynamics {
      in power = vehiclePower;
      in mass = vehicleMass;
      in delta_t = deltaT;
      in x_in = position; in v_in = speed;
      out x_out; out v_out;
    }
    then assign position := dynamics.x_out;
    then assign speed := dynamics.v_out;
    then assign positions :=
      positions->including(position);
  }
}

```

Using `:=` instead of `=` here indicates an *initial value*, instead of a binding.

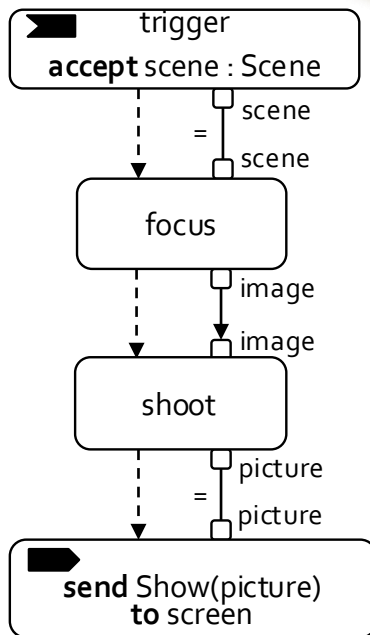
① The functions [including](#) comes from the [SequenceFunctions](#) library package.

Asynchronous Messaging

An *accept* action receives an incoming asynchronous transfer any kind of values.

This is the *name* of the action.

This is a declaration of what is being received, which can be anything.



```

action takePicture : TakePicture {
  action trigger accept scene : Scene;

  then action focus : Focus {
    in scene = trigger.scene;
    out image;
  }
  flow from focus.image to shoot.image;
  then action shoot : Shoot {
    in image;
    out picture;
  }

  then send new Show(shoot.picture) to screen;
}

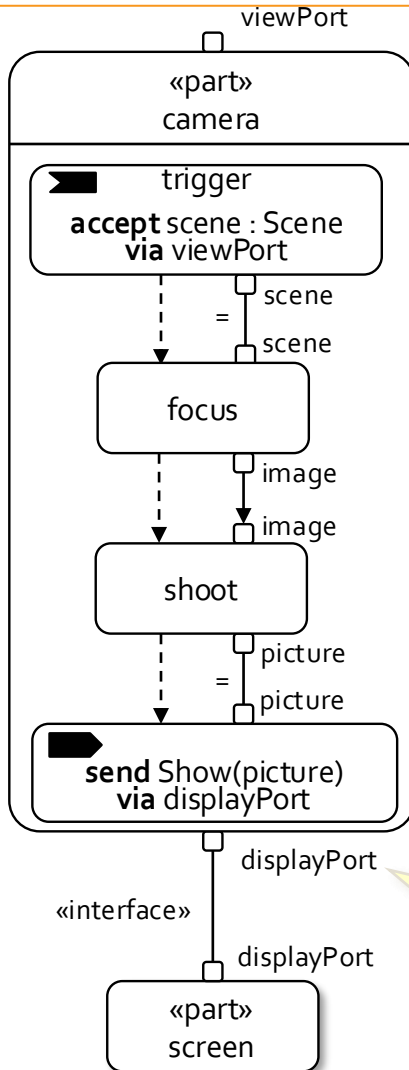
```

This is an *expression* that constructs the value to be sent.

A *send* action is an outgoing transfer of values to a specific target.

① The notation "**Show(...)**" means to create an instance of the definition **Show** with the given value for its nested attribute.

This is also an expression, evaluating to the target (in this case just a feature reference).



```

part camera : Camera {
  port viewport;
  port displayPort;

```

```

  action takePicture : TakePicture {
    action trigger accept scene : Scene via viewport;

```

```

    then action focus : Focus {
      in scene = trigger.scene;
      out image;
    }

```

```

    flow from focus.image to shoot.image;
    then action shoot : Shoot {
      in image;
      out picture;
    }

```

```

    then send new Show(shoot.picture) via displayPort;

```

An accept action can be *via* a specific port, meaning that the transfer is expected to be received *by* that port.

An asynchronous transfer sent via a port has that port as its *source*. The *target* of the transfer is determined by what the port is connected to.

A send action can also be sent *via* a port, meaning that the transfer is sent out *from* that port.

An "opaque" action definition or usage can be specified using a *textual representation* annotation in a language other than SysML.

```

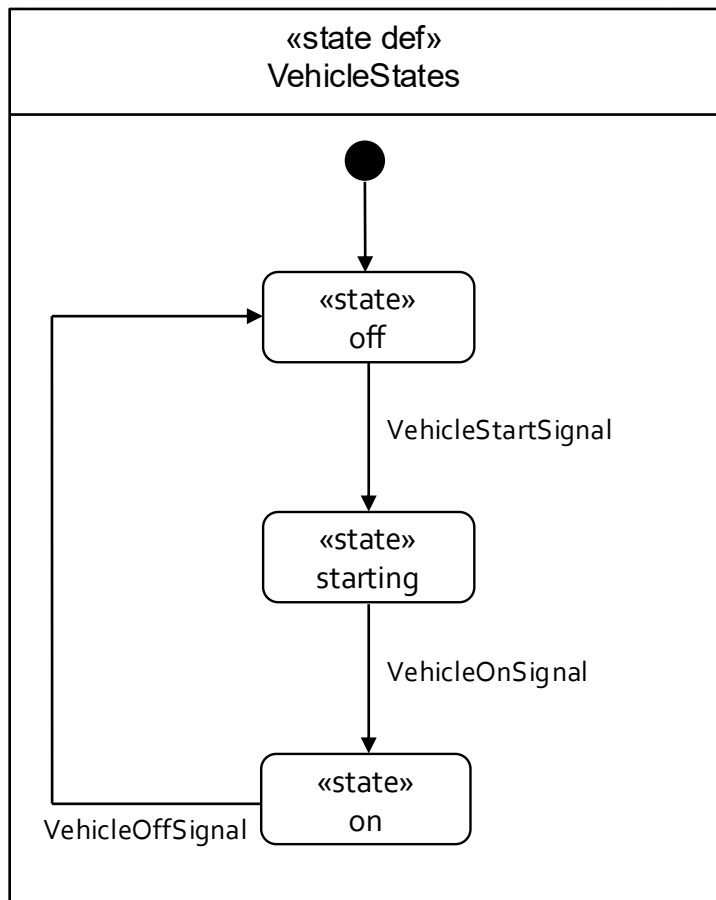
action def UpdateSensors {
  in sensors : Sensor[*];
  language "Alf"
  /*
   * for (sensor in sensors) {
   *   if (sensor.ready) {
   *     Update(sensor);
   *   }
   * }
  */
}
  
```

The textual representation body is written using comment syntax. The */*, */* and leading *** symbols are *not* included in the body text. Note that support for referencing SysML elements from the body text is tool-specific.

- ① A textual representation annotation can actually be used with any kind of element, not just actions. OMG-standard languages a tool may support include "OCL" (Object Constraint Language) and "Alf" (Action Language for fUML). A tool can also provide support for other languages (e.g., "JavaScript" or "Modelica").

State Definitions (1)

A *state definition* is like a state machine in UML and SysML v1. It defines a behavioral state that can be exhibited by a system.



```

state def VehicleStates {
    first start then off;

    state off;

    transition off_to_starting
        first off
        accept VehicleStartSignal
        then starting;

    state starting;

    transition starting_to_on
        first starting
        accept VehicleOnSignal
        then on;

    state on;

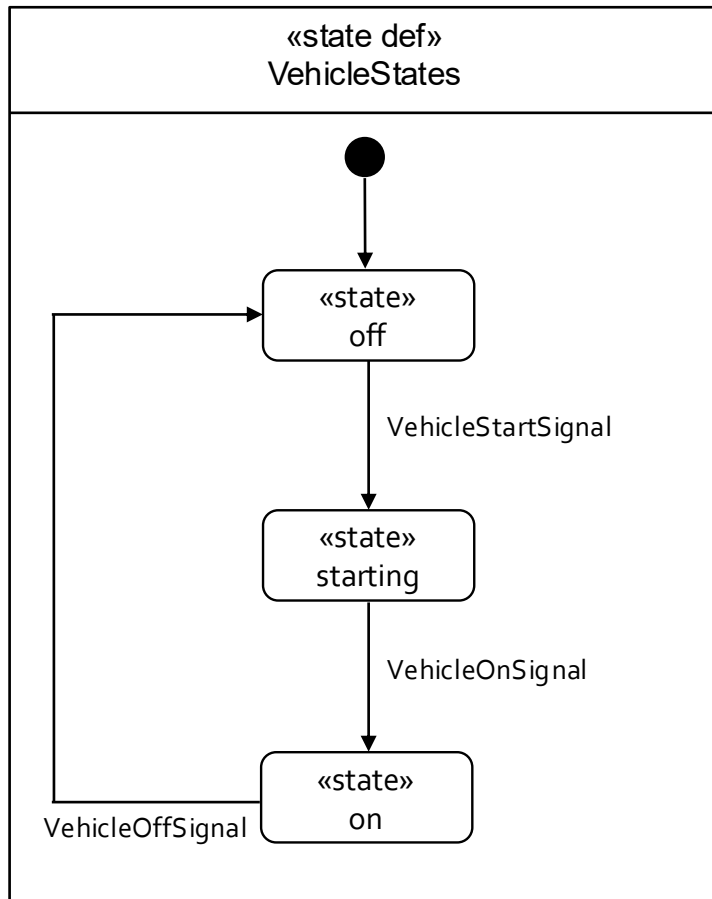
    transition on_to_off
        first on
        accept VehicleOffSignal
        then off;
}
    
```

This indicates the the initial state after entry is "off".

A state definition can specify a set of discrete nested *states*.

States are connected by *transitions* that fire on acceptance of item transfers (like accept actions).

State Definitions (2)



```

state def VehicleStates {
    first start then off;

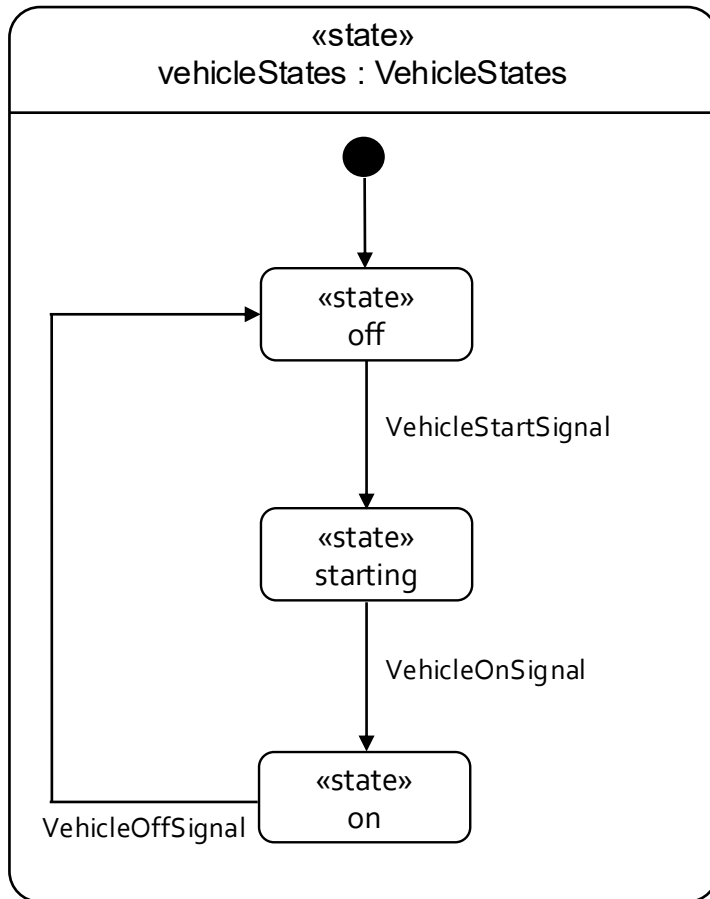
    state off;
    accept VehicleStartSignal
    then starting;

    state starting;
    accept VehicleOnSignal
    then on;

    state on;
    accept VehicleOffSignal
    then off;
}
    
```

This is a shorthand for a transition whose source is the lexically previous state.

State Decomposition



```

state def VehicleStates;

state vehicleStates : VehicleStates {
    first start then off;

    state off;
    accept VehicleStartSignal
    then starting;

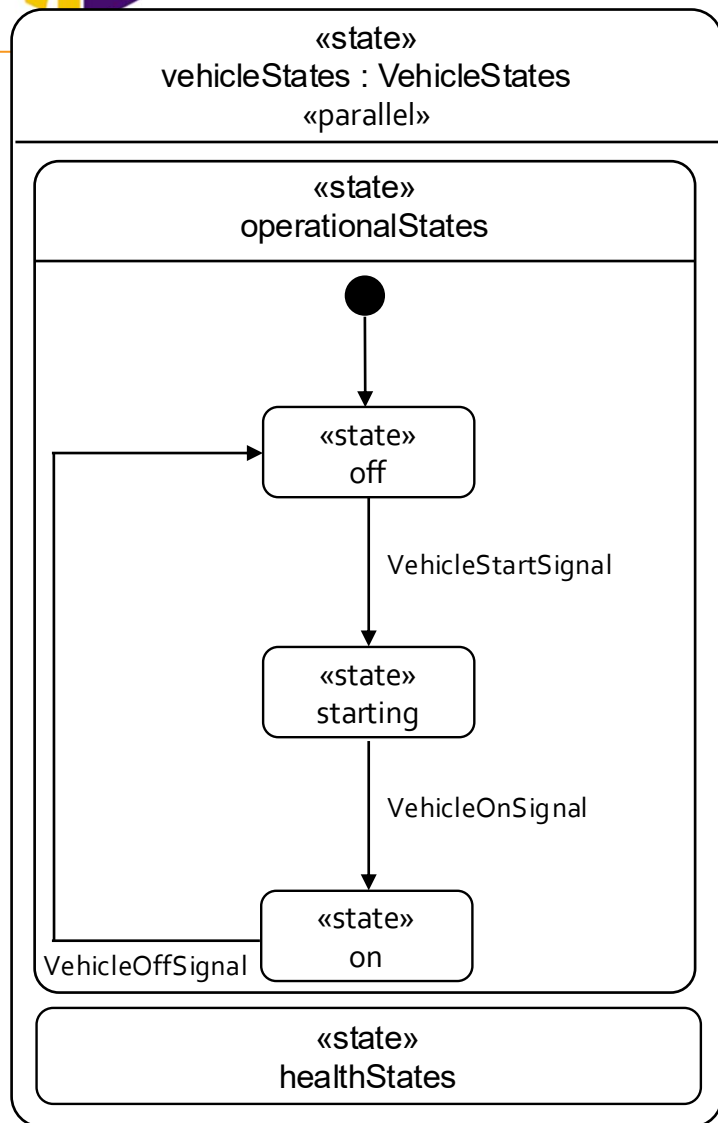
    state starting;
    accept VehicleOnSignal
    then on;

    state on;
    accept VehicleOffSignal
    then off;
}
    
```

A state can be explicitly declared to be a usage of a state definition.

A state can also be directly decomposed into other states.

Concurrent States



A *parallel state* is one whose nested states are concurrent.

```

state def VehicleStates;

state vehicleStates : VehicleStates parallel {

    state operationalStates {
        first start then off;

        state off;
        accept VehicleStartSignal
        then starting;

        state starting;
        accept VehicleOnSignal
        then on;

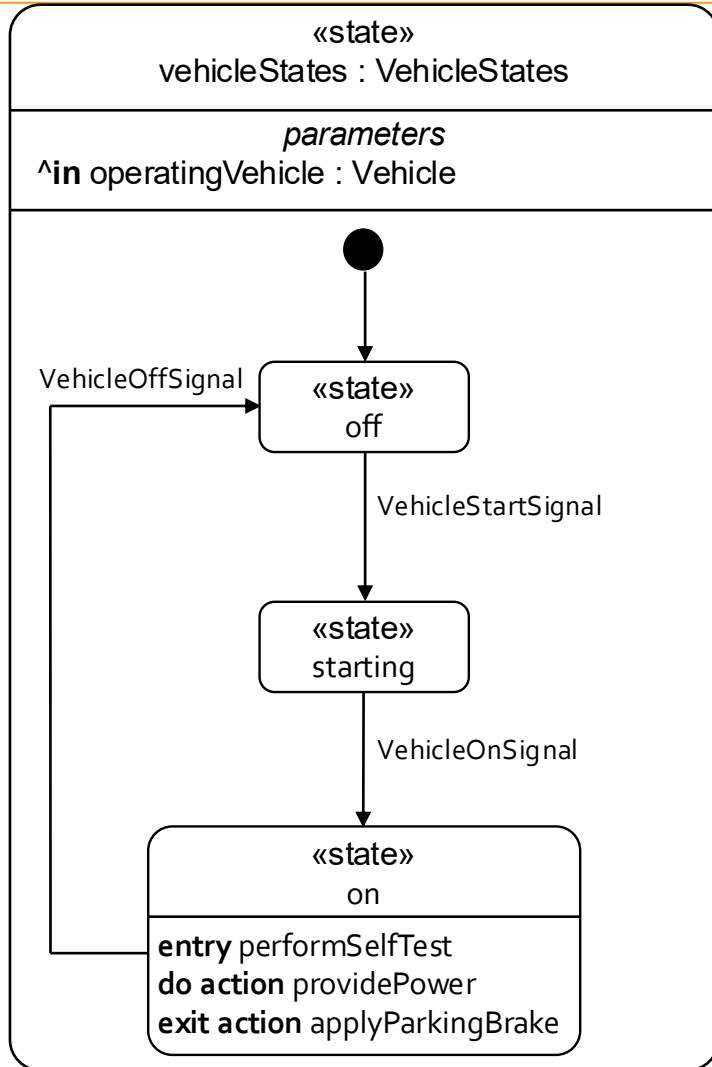
        state on;
        accept VehicleOffSignal
        then off;
    }

    state healthStates {
        ...
    }
}
  
```

⚠ Transitions are not allowed between concurrent states.

State Entry, Do and Exit Actions

States, like actions, can have parameters.



```

action performSelfTest{in vehicle : Vehicle;}

state def VehicleStates{in operatingVehicle : Vehicle;}

state vehicleStates : VehicleStates {
    in operatingVehicle : Vehicle;

    first start then off;

    state off;
    accept VehicleStartSignal
        then starting;

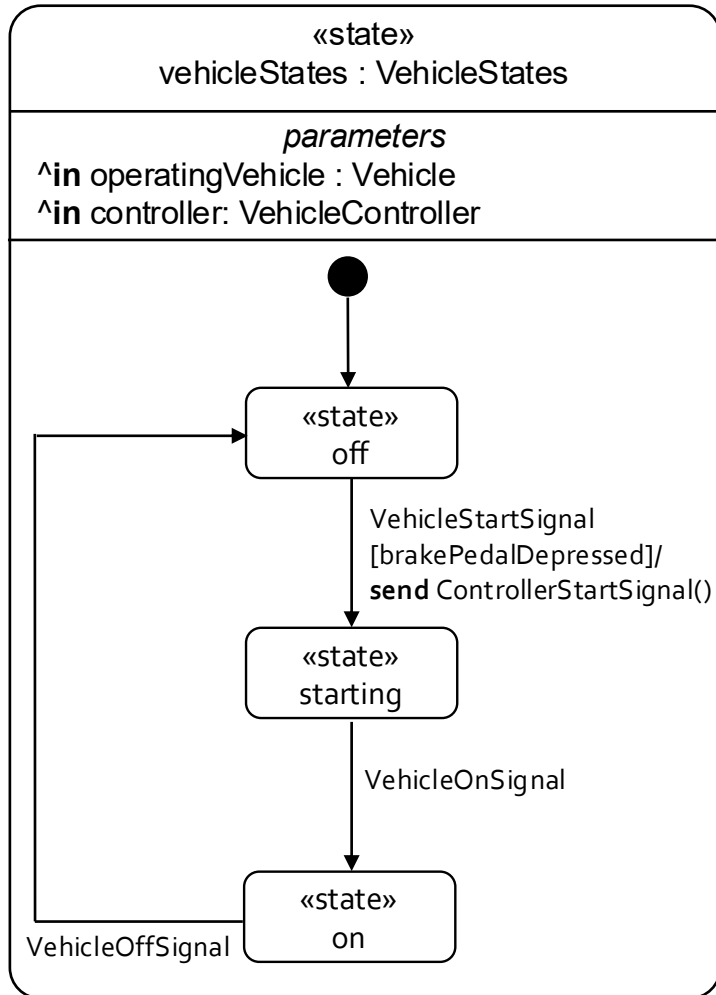
    state starting;
    accept VehicleOnSignal
        then on;

    state on {
        entry performSelfTest
            { in vehicle = operatingVehicle; }
        do action providePower { ... }
        exit action applyParkingBrake { ... }
    }
    accept VehicleOffSignal
        then off;
}
    
```

① An entry action is performed on entry to a state, a do action while in it, and an exit action on exit from it.

A state entry, do or exit can reference an action defined elsewhere...

... or the action can be defined within the state.



```
action performSelfTest{in vehicle : Vehicle;}
```

```
state def VehicleStates {
  in operatingVehicle : Vehicle;
  in controller : VehicleController; }
```

```
state vehicleStates : VehicleStates {
  in operatingVehicle : Vehicle;
  in controller : VehicleController;
```

```
first start then off;
```

```
state off;
accept VehicleStartSignal
  if operatingVehicle.brakePedalDepressed
  do send new ControllerStartSignal() to controller
  then starting;
```

```
state starting;
accept VehicleOnSignal
  then on;
```

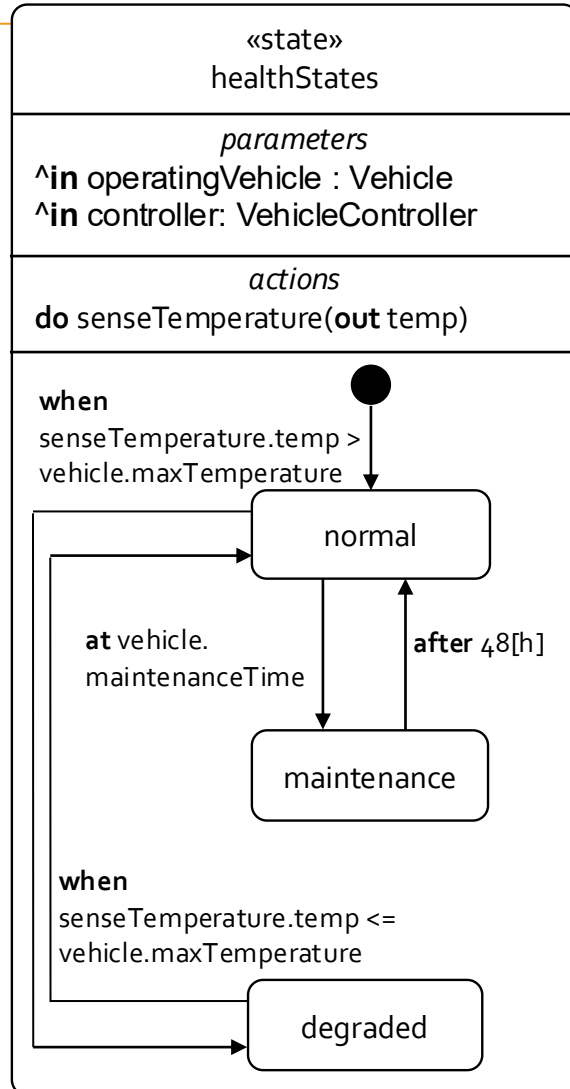
```
state on { ... }
accept VehicleOffSignal
  then off;
```

```
}
```

A *guard* is a condition that must be true for a transition to fire.

An *effect action* is performed when a transition fires, before entry to the target state.

Change and Time Triggers



```

action senseTemperature { out temp : TemperatureValue; }

state healthStates {
  in vehicle : Vehicle;
  in controller : VehicleController;
  do senseTemperature;

  first start then normal;

  state normal;
  accept at vehicle.maintenanceTime
    then maintenance;
  accept when senseTemperature.temp > vehicle.maxTemperature
    do send new OverTemp() to controller
    then degraded;

  state maintenance {
    entry assign vehicle.maintenanceTime :=
      vehicle.maintenanceTime + vehicle.maintenanceInterval;
  }
  accept after 48 [h]
    then normal;

  state degraded;
  accept when senseTemperature.temp <= vehicle.maxTemperature
    then normal;
}
  
```

An absolute time trigger fires at a given instant of time.

A change trigger fires when a given Boolean expression becomes true.

⚠ Time instant quantities are defined in the **Time** package, *not* the **ISO** package.

A relative time trigger fires after a given time duration.

① Change and time triggers can also be used in accept actions.

Redefining the **localClock** feature of a part provides a local time reference for composite items, parts and actions nested within it.

① A **localClock** can also be declared for an item or an action (or any kind of *occurrence* that happens over time).

All absolute and relative time references are relative to the **localClock**.

```

part def Server {
  part :>> localClock = new Time::Clock();

  attribute today : String;

  port requestPort;

  state ServerBehavior {
    first start then off;

    state off;
    accept Start via requestPort
      then waiting;

    state waiting;
    accept request : Request via requestPort
      then responding;
    accept at new Time::Iso8601DateTime(today + "11:59:00")
      then off;

    state responding;
    accept after 5 [SI::min]
      then waiting;
  }
}

```

Each **Server** instance will have its own independent **Clock**.

① If no **localClock** is declared, the default is a **universalClock** that provides a common time universal reference.

exhibit identifies the part that is exhibiting states that are defined elsewhere.

```

part vehicle : Vehicle {

    part vehicleController : VehicleController;

    exhibit vehicleStates {
        in operatingVehicle = vehicle;
        in controller = vehicleController;
    }

}
    
```

Parameters for a state usage can be bound in the same way as parameters of an action usage.

⚠ Like performed actions, exhibited states are *not* composite features, therefore, a **localClock** on a part will *not* apply to performed actions or exhibited states of the part.

① An *occurrence* is something that happens over time. Items and parts are structural occurrences. Actions are behavioral occurrences. Attributes are *not* occurrences.

A *time slice* represents a portion of the *lifetime* of an occurrence over some period of time.

A *snapshot* represents an occurrence at a specific point in time (a time slice with zero duration).

```

attribute def Date;

item def Person;

part def Vehicle {
    timeslice assembly;

    first assembly then delivery;

    snapshot delivery;

    first delivery then ownership;

    timeslice ownership[0..*] ordered;

    snapshot junked = done;
}
    
```

A *succession* asserts that the first occurrence ends before the second can begin.

done is the name of the *end-of-life snapshot* of any occurrence.

① Timeslices and snapshots are *suboccurrences*.

This is a shorthand for a succession after the lexically previous occurrence.

Time slices and snapshots can have nested time slices, snapshots and other features applicable during their occurrence.

```

part def Vehicle {
  ...

  snapshot delivery {
    attribute deliveryDate : Date;
  }

  then timeslice ownership[0..*] ordered {
    snapshot sale = start;

    ref item owner : Person[1];

    timeslice driven[0..*] {
      ref item driver : Person[1];
    }

    snapshot junked = done;
  }
}
    
```

`start` is the name of the *start-of-life snapshot* of any occurrence.

One occurrence references another during the same time period in the other reference's lifetime.

Event Occurrences

An *event occurrence* is a reference to something that happens during the lifetime of another occurrence.

Event occurrences can be sequenced as usual using succession.

```

part driver : Driver {
    event occurrence setSpeedSent;
}

part vehicle : Vehicle {
    part cruiseController : CruiseController {
        event occurrence setSpeedReceived;
        then event occurrence sensedSpeedReceived;
        then event occurrence fuelCommandSent;
    }

    part speedometer : Speedometer {
        event occurrence sensedSpeedSent;
    }

    part engine : Engine {
        event occurrence fuelCommandReceived;
    }
}
    
```

Messages (1)

An *occurrence definition* defines a class of occurrences, without committing to whether they are structural or behavioral.

A *message* asserts that there is a *transfer* of some payload of a certain type from one occurrence to another (specifying the payload type is optional).

Messages can be explicitly ordered. Otherwise, they are only partially ordered by the time-ordering of their respective source and target events.

```
occurrence def CruiseControlInteraction {
  ref part :>> driver;
  ref part :>> vehicle;

  message setSpeedMessage of SetSpeed
    from driver.setSpeedSent
    to vehicle.cruiseController.setSpeedReceived;

  message sensedSpeedMessage of SensedSpeed
    from vehicle.speedometer.sensedSpeedSent
    to vehicle.cruiseController.sensedSpeedReceived;

  message fuelCommandMessage of FuelCommand
    from vehicle.cruiseController.fuelCommandSent
    to vehicle.engine.fuelCommandReceived;

  first setSpeedMessage then sensedSpeedMessage;
}
```


Messages (2)

An event can also be specified by reference to an identified occurrence (such as the source or target of a message).

Each message has a [sourceEvent](#) and [targetEvent](#), even if it is not explicitly identified in the message declaration.

```
occurrence def CruiseControlInteraction {
  ref part driver : Driver {
    event setSpeedMessage.sourceEvent;
  }

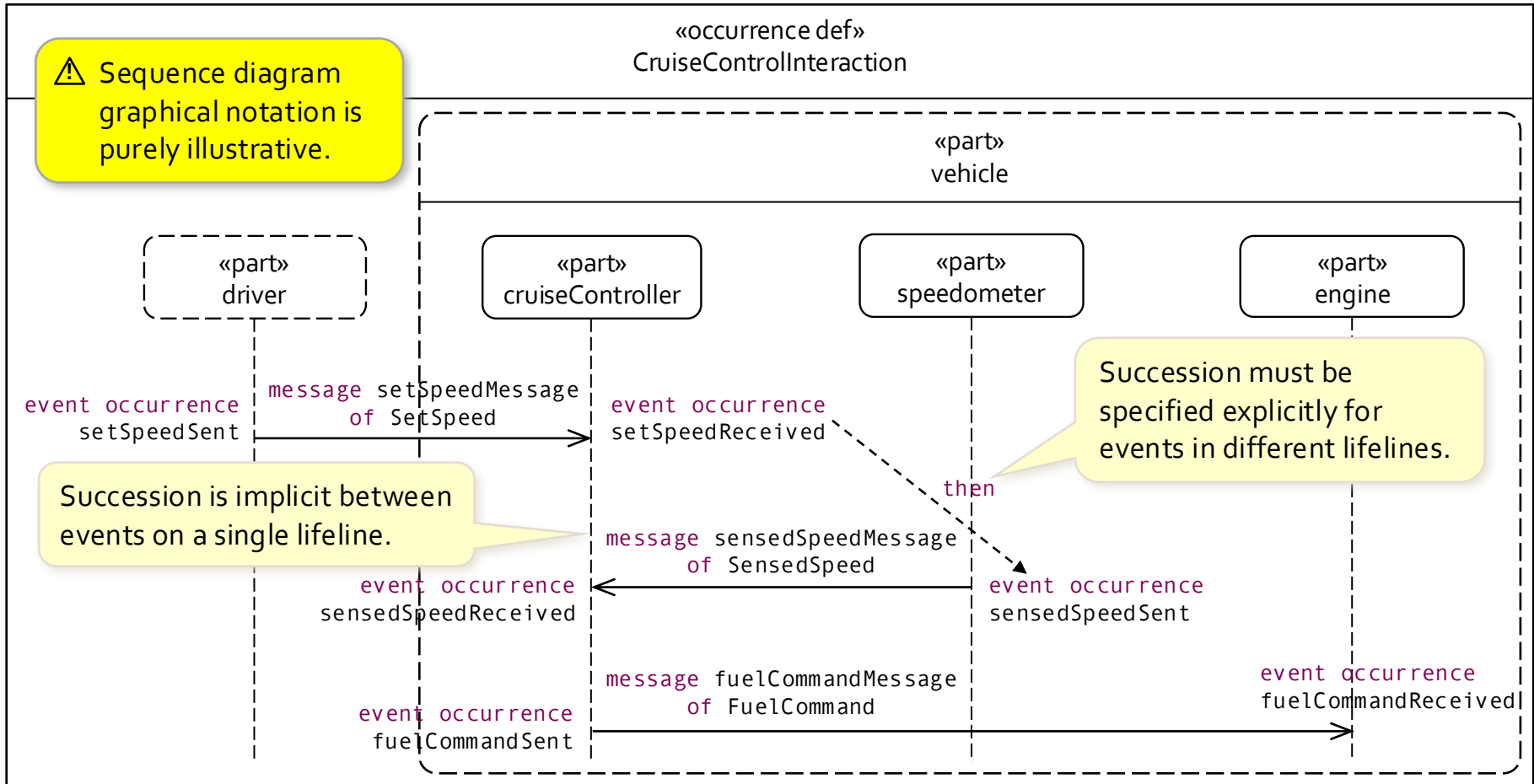
  ref part vehicle : Vehicle {
    part cruiseController : CruiseController {
      event setSpeedMessage.targetEvent;
      then event sensedSpeedMessage.targetEvent;
      then event fuelCommandMessage.sourceEvent;
    }
    part speedometer : Speedometer {
      event sensedSpeedMessage.sourceEvent;
    }
    part engine : Engine {
      event fuelCommandMessage.targetEvent;
    }
  }

  message setSpeedMessage of SetSpeed;
  then message sensedSpeedMessage of SensedSpeed;
  message fuelCommandMessage of FuelCommand;
}
```

Interactions

① Event occurrences and messages can be used together to specify an *interaction* between parts without committing to *how* the interaction happens.

⚠ Sequence diagram graphical notation is purely illustrative.



Interaction Realization by Actions (1)

```

part driver_a : Driver {
    perform action driverBehavior {
        send SetSpeed() to vehicle_a;
    }
}

part vehicle_a : Vehicle {
    part cruiseController_a : CruiseController {
        perform action controllerBehavior {
            accept SetSpeed via vehicle_a;
            then accept SensedReader via cruiseController_a;
            then send new FuelCommand() to engine_a;
        }
    }

    part speedometer_a : Speedometer {
        perform action speedometerBehavior {
            send new SensedReader() to cruiseController_a;
        }
    }

    part engine_a : Engine {
        perform action engineBehavior {
            accept FuelCommand via engine_a;
        }
    }
}
    
```

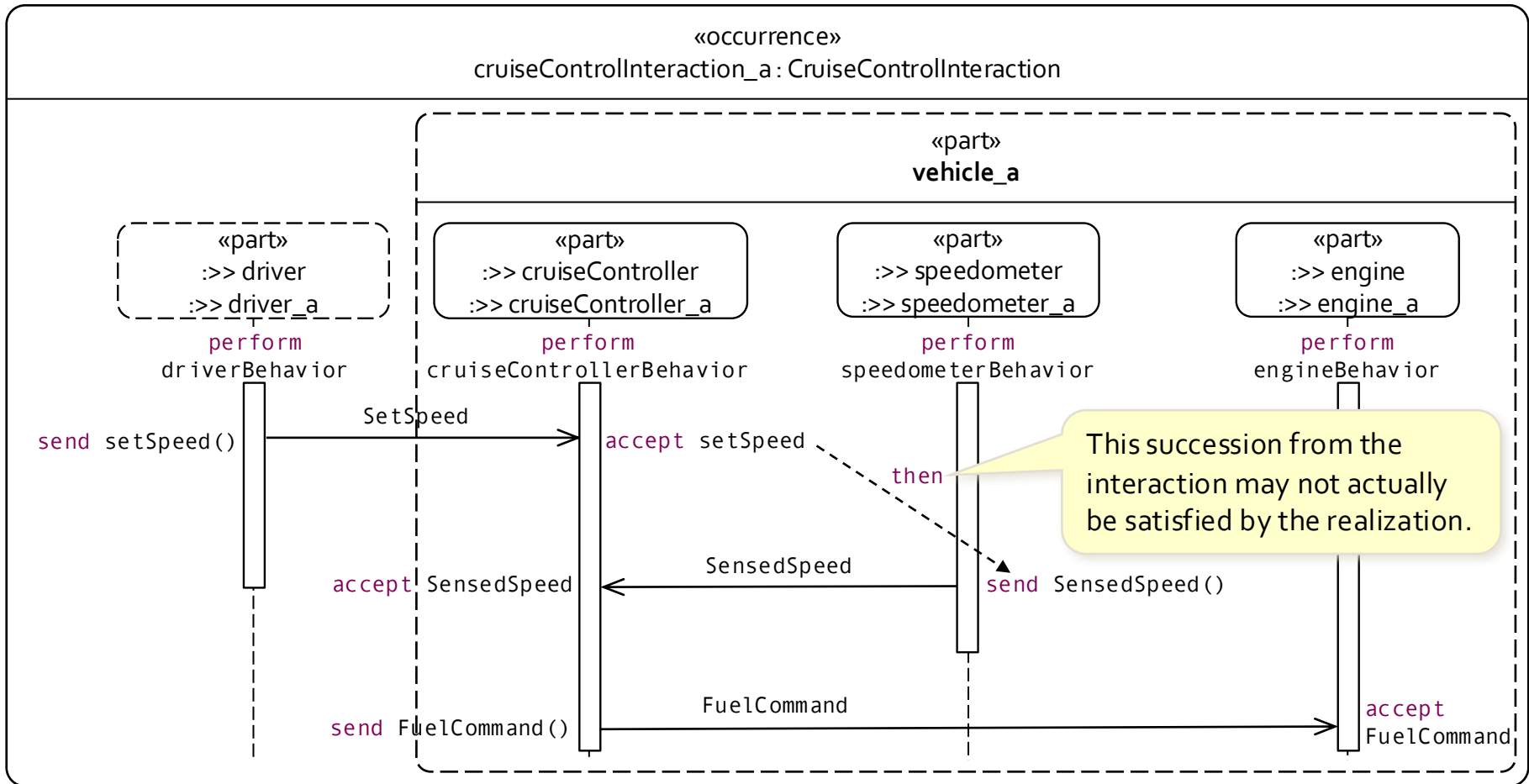
In this model, event occurrences are realized by nested send and accept actions.

Messages are realized by the implicit transfers between send actions and corresponding accept actions.

① An interaction can be *realized* by many different models. A valid realization must have suboccurrences that can be identified with all the events and messages in the interaction.

① In more realistic cases, a realizing model will have more extensive behavior, only a subset of which will conform to any one interaction model.

Interaction Realization by Actions (2)



Interaction Realization by Flows (1)

In this model, event occurrences are realized by values leaving from and arriving at ports.

Messages are realized by explicit flows between features of ports.

```

part driver_b : Driver {
    port setSpeedPort {
        out setSpeed : SetSpeed;
    }
}

interface driver_b.setSpeedPort to vehicle_b.setSpeedPort {
    flow of SetSpeed from driver_b.setSpeedPort.setSpeed
    to vehicle_b.setSpeedPort.setSpeed;
}

part vehicle_b : Vehicle {
    port setSpeedPort {
        in setSpeed : SetSpeed;
    }
}

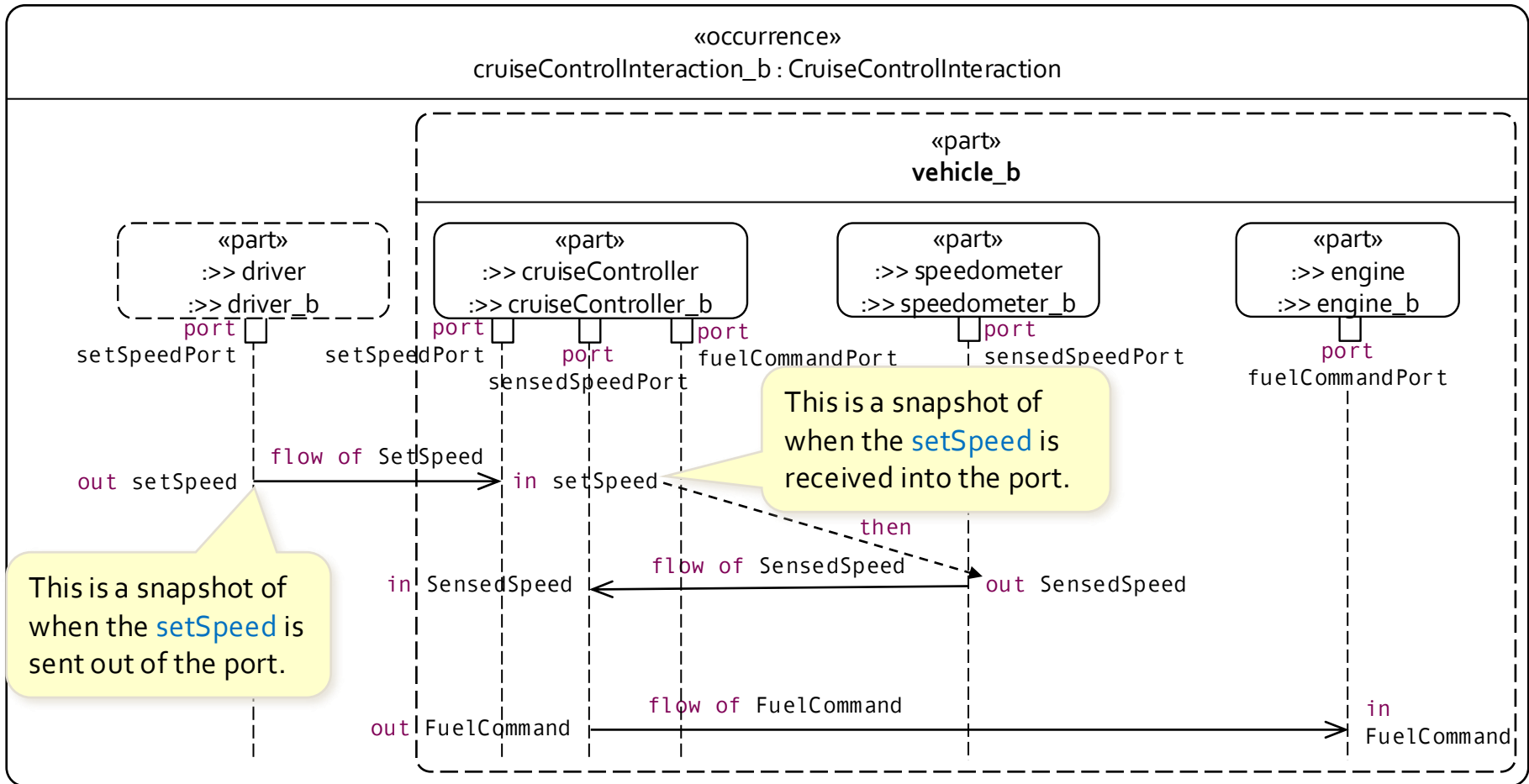
part cruiseController_b : CruiseController {
    port setSpeedPort {
        in setSpeed : SetSpeed = vehicle_b.setSpeedPort.setSpeed;
    }
    port sensedSpeedPort {
        in sensedSpeed : SetSpeed;
    }
    ...
}

flow of SetSpeed from speedometer_b.sensedSpeedPort.sensedSpeed
to cruiseController_b.sensedSpeedPort.sensedSpeed;

part speedometer_b : Speedometer {
    port sensedSpeedPort {
        out sensedSpeed : SetSpeed;
    }
}
    
```

① Ports are also suboccurrences of the parts that contain them.

Interaction Realization by Flows (2)



```

message setSpeedMessage of SetSpeed
  from driver.setSpeedSent
  to vehicle.cruiseController.setSpeedReceived;

then message sensedSpeedMessage of SensedSpeed
  from vehicle.speedometer.sensedSpeedSent
  to vehicle.cruiseController.sensedSpeedReceived;

then message fuelCommandMessage of fuelCommand : FuelCommand
  from vehicle.cruiseController.fuelCommandSent
  to vehicle.engineController.fuelCommandReceived;

then message fuelCommandForwardingMessage
  of fuelCommand : FuelCommand = fuelCommandMessage.fuelCommand
  from vehicle.engineController.fuelCommandForwarded
  to vehicle.engine.fuelCommandReceived;
  
```

① The default name for a message payload is **payload**.

The payload of a message can be declared with a name for reference.

⚠ Note that a payload declaration without a colon names the *type* of the payload, not the payload itself.

The payload of a message can also be bound to a specific value. In this case, the **fuelCommandMessage** payload is forwarded by the **engineController** to the **engine**.

Individuals and Snapshots (1)

As occurrences, individuals can have snapshots that describe them at a single instant of time.

An *individual definition* is an occurrence definition (of any kind) restricted to model a single individual and how it changes over its lifetime.

An individual definition will often specialize a definition of the general class of things the individual is one of.

```
individual part def Vehicle_1 :> Vehicle {
    snapshot part vehicle_1_t0 : Vehicle_1 {
        :>> mass = 2000.0;
        :>> status {
            :>> gearSetting = 0;
            :>> acceleratorPosition = 0.0;
        }
    }
}
```

This is a compact notation for showing redefinition of an attribute usage.

An attribute does not have snapshots, but it can be asserted to have a specific value in a certain snapshot.

Individuals and Snapshots (2)

```

individual part def Vehicle_1 :> Vehicle {

    snapshot part vehicle_1_t0 : Vehicle_1 {
        :>> mass = 2000.0;
        :>> status {
            :>> gearSetting = 0;
            :>> acceleratorPosition = 0.0;
        }
    }

    snapshot part vehicle_1_t1 : Vehicle_1 {
        :>> mass = 1500.0;
        :>> status {
            :>> gearSetting = 2;
            :>> acceleratorPosition = 0.5;
        }
    }

    first vehicle_1_t0 then vehicle_1_t1;
}
    
```

As usual, succession asserts that the first snapshot occurs before the second in time, in some context.

The values of the attributes of an individual can change over time.

Individuals and Roles

```

individual part def Vehicle_1 :> Vehicle {
    part leftFrontWheel : Wheel;
    part rightFrontWheel : Wheel;
}

individual part def Wheel_1 :> Wheel;

individual part vehicle_1 : Vehicle_1 {

    snapshot part vehicle_1_t0 {
        snapshot leftFrontWheel_t0 : Wheel_1 :>> leftFrontWheel;
    }

    then snapshot part vehicle_1_t1 {
        snapshot rightFrontWheel_t1 : Wheel_1 :>> rightFrontWheel;
    }
}
    
```

An *individual usage* represents an individual during some portion of its lifetime.

During the first snapshot of `vehicle_1`, `Wheel_1` has the role of the `leftFrontWheel`.

During a later snapshot, the *same* `Wheel_1` has the role of the `rightFrontWheel`.

Individuals and Time Slices

A time slice represents an individual over some period of time (that this is a time slice of a part is implicit).

start and **done** are snapshots at the beginning and end of the time slice.

Succession asserts that the first time slice must complete before the second can begin.

```
individual item def Alice :> Person;
individual item def Bob :> Person;

individual part : Vehicle_1 {

    timeslice aliceDriving {
        ref individual item :>> driver : Alice;

        snapshot :>> start {
            :>> mass = 2000.0;
        }

        snapshot :>> done {
            :>> mass = 1500.0;
        }
    }

    then timeslice bobDriving {
        ref individual item :>> driver : Bob;
    }
}
```

During this time slice of **Vehicle_1**, the **Alice** has the role of the **driver**.

During a later time slice of **Vehicle_1**, **Bob** has the role of the **driver**.

```
package MassRollup1 {
  private import ScalarFunctions::*;
  part def MassedThing {
    attribute simpleMass :> ISQ::mass;
    attribute totalMass :> ISQ::mass;
  }

  part simpleThing : MassedThing {
    attribute :>> totalMass = simpleMass;
  }

  part compositeThing : MassedThing {
    part subcomponents: MassedThing[*];
    attribute :>> totalMass =
      simpleMass + sum(subcomponents.totalMass);
  }
}
```

`ISQ::mass` is a standard quantity kind from the *International System of Quantities* library model.

A *feature value* is a shorthand for binding a feature to the result of an expression (here simply the value of another feature).

An *expression* in general is a computation expressed using a typical mathematical operator notation.

The dot notation can be used as an expression to read the values of a feature. Here, the `totalMass` is collected for each of the subcomponents.

Car Mass Rollup Example (1)

```
private import ScalarValues::*;
private import MassRollup1::*;

part def CarPart :> MassedThing {
  attribute serialNumber : String;
}

part car: CarPart :> compositeThing {
  attribute vin :>> serialNumber;
  part carParts : CarPart[*] :>> subcomponents;
  part engine :> simpleThing, carParts { ... }
  part transmission :> simpleThing, carParts { ... }
}

// Example usage
private import SI::kg;
part c :> car {
  attribute :>> simpleMass = 1000[kg];
  part :>> engine {
    attribute :>> simpleMass = 100[kg];
  }
  part attribute transmission {
    attribute :>> simpleMass = 50[kg];
  }
}

// c.totalMass --> 1150.0[kg]
```

This is an expression for a quantity with a specific numeric value and unit. Note that units are identified on the quantity *value*, *not* the type.

The `totalMass` of `c` can be computed using the feature value expressions from `simpleThing` and `compositeThing`.

```
package MassRollup2 {
  private import ScalarFunctions::*;
  part def MassedThing {
    attribute simpleMass :> ISQ::mass;
    attribute totalMass :> ISQ::mass default simpleMass;
  }

  part compositeThing : MassedThing {
    part subcomponents: MassedThing[*];
    attribute :>> totalMass default mass + sum(subcomponents.totalMass);
  }

  part filteredMassThing :> composi
    attribute minMass : ISQ::mass
    attribute :>> totalMass =
      sum(subcomponents.totalMass.{in p:>ISQ::mass; p >= minMass}));
  }
}
```

A *default value* is feature value that applies *unless* explicitly overridden.

A default value can be overridden when the feature is redefined, with a default or bound value.

Once bound, the value of the feature is *fixed* as the result of the value expression.

A dot can be followed by a *select* expression in order to filter a collection of values.

Car Mass Rollup Example (2)

```
private alarValues::*;
private import MassRollup2::*;

part def CarPart :> MassedThing {
  attribute serialNumber : String;
}

part car: CarPart :> compositeThing {
  attribute vin :>> serialNumber;
  part carParts : CarPart[*] :>> subcomponents;
  part engine :> carParts { ... }
  part transmission :> carParts { ... }
}

// Example usage
private import SI::kg;
part c :> car {
  attribute :>> simpleMass = 1000[kg];
  part :>> engine {
    attribute :>> simpleMass = 100[kg];
  }
  part attribute transmission {
    attribute :>> simpleMass = 50[kg];
  }
}

// c.totalMass --> 1150.0[kg]
```

The default of **totalMass** to **simpleMass** is inherited.

The default for **totalMass** is overridden as specified in **compositeThing**.

The **totalMass** default is *not* overridden for the nested **carParts**.

When computing the **totalMass** of **c**, **c.engine** and **c.transmission**, the relevant defaults are used.

Calculation Definitions

A *calculation definition* is a reusable, parameterized expression.

```
calc def Power {
  in whlpwr : PowerValue;
  in Cd : Real;
  in Cf : Real;
  in tm : MassValue;
  in v : SpeedValue;
  return : PowerValue;
```

Similarly to actions, the directed features of a calculation are its parameters.

A calculation has a single *return result*.

```
  attribute drag = Cd * v;
  attribute friction = Cf * tm *

```

The calculation can include the computation of intermediate values.

```
  whlpwr - drag - friction

```

The calculation *result expression* must conform to the return type.

⚠ There is no semicolon at the end of a result expression.

```
}

calc def Acceleration { in tp: PowerValue; in tm : MassValue; in v: SpeedValue:
  return : AccelerationValue = tp / (tm * v);
}
```

The result expression can also be bound directly to the return parameter.

```
calc def Velocity { in dt : TimeValue; in v0 : SpeedValue; in a : AccelValue;
  return : SpeedValue = v0 + a * dt;
}
```

```
calc def Position (dt : TimeValue; x0 : LengthValue; v : SpeedValue;
  return : LengthValue = x0 + v * dt;
}
```


Calculation Usages (1)

Values are bound to the parameters of *calculation usages* (similar to action parameters).

```

part def VehicleDynamics {
  attribute C_d : Real;
  attribute C_f : Real;
  attribute wheelPower : PowerValue;
  attribute mass : MassValue;

  action straightLineDynamics { in delta_t : TimeValue;
    in v_in : SpeedValue; in x_in : LengthValue;

    calc acc : Acceleration {
      in tp = Power(wheelPower, C_d, C_f, mass, v_in);
      in tm = mass;
      in v = v_in;
      return a;
    }
    calc vel : Velocity {
      in dt = delta_t;
      in v0 = v_in;
      in a = acc.a;
      return v;
    }
    calc pos : Position {
      in dt = delta_t;
      in x0 = x_in;
      in v = vel.v;
      return x;
    }

    out v_out : SpeedValue = vel.v;
    out x_out : LengthValue = pos.x;
  }
}

```

A calculation definition can also be *invoked* as an expression, with input values given as arguments, evaluating to the result of the calculation.

Calculation results can be referenced by name (they are output parameters).

Calculation Usages (2)

```
attribute def DynamicState {
  attribute v: SpeedValue;
  attribute x: LengthValue;
}
```

```
part def VehicleDynamics {
  attribute C_d : Real;
  attribute C_f : Real;
  attribute wheelPower : PowerValue;
  attribute mass : MassValue;
```

A calculation can be specified without an explicit calculation definition.

Calculations can also handle structured values.

```
calc updateState {
  in delta_t : TimeValue; in currState : DynamicState;
```

```
  attribute totalPower : PowerValue =
    Power(wheelPower, C_d, C_f, mass, currState.v);
```

This is a declaration of the result *parameter* of the calculation, with bound subattributes.

```
  return attribute newState : DynamicState {
    :>> v = Velocity(delta_t, currState.v, Acceleration(totalPower, mass, currState.v));
    :>> x = Position(delta_t, currState.x, currState.v);
  }
}
```

Constraint Definitions (1)

```
private import ISQ::*;
private import SI::*;
private import ScalarFunctions::*

constraint def MassConstraint {
  in partMasses : MassValue[0..*];
  in massLimit : MassValue;

  sum(partMasses) <= massLimit
}

part def Vehicle {
  constraint massConstraint : MassConstraint {
    in partMasses = (chassisMass, engine.mass, transmission.mass);
    in massLimit = 2500[kg];
  }

  attribute chassisMass : MassValue;

  part engine : Engine {
    attribute mass : MassValue;
  }

  part transmission : Transmission {
    attribute mass : MassValue;
  }
}
```

A *constraint definition* is a reusable, parameterized Boolean expression.

Constraint parameters are always **in** parameters.

The *constraint expression* can be any Boolean expression using the constraint parameters.

Values are bound to constraint parameters (similarly to actions).

⚠ There is no semicolon at the end of a constraint expression.

A *constraint* is the usage of a constraint definition, which may be true or false in a given context.

① A constraint may be *violated* (false) without making the model inconsistent.

Constraint Definitions (2)

```
private import ISQ::*;
private import SI::*;
private import ScalarFunctions::*;

constraint def MassConstraint {
  attribute partMasses : MassValue[0..*];
  attribute massLimit : MassValue;

  sum(partMasses) <= massLimit
}

part def Vehicle {
  constraint massConstraint : MassConstraint {
    redefines partMasses = (chassisMass, engine.mass, transmission.mass);
    redefines massLimit = 2500[kg];
  }

  attribute chassisMass : MassValue;

  part engine : Engine {
    attribute mass : MassValue;
  }

  part transmission : Engine {
    attribute mass : MassValue;
  }
}
```

Alternatively, constraint parameters may be modeled as attribute or reference features.

The constraint parameter properties are then redefined in order to be bound.

Constraint Assertions (1)

```
private import ISQ::*;
private import SI::*;
private import ScalarFunctions::*;

constraint def MassConstraint {
  in partMasses : MassValue[0..*];
  in massLimit : MassValue;

  sum(partMasses) <= massLimit
}

part def Vehicle {
  assert constraint massConstraint : MassConstraint {
    in partMasses = (chassisMass, engine.mass, transmission.mass);
    in massLimit = 2500[kg];
  }

  attribute chassisMass : MassValue;

  part engine : Engine {
    attribute mass : MassValue;
  }

  part transmission : Engine {
    attribute mass : MassValue;
  }
}
```

A constraint assertion asserts that a constraint *must* be true.

❶ If an assertion is violated, then the model is *inconsistent*.

Constraint Assertions (2)

The constraint expression can also be defined on a *usage* of a constraint def.

```
constraint def MassConstraint {
  in partMasses : MassValue[0..*];
  in massLimit : MassValue;
}
constraint massConstraint : MassConstraint {
  sum(partMasses) <= massLimit
}

part def Vehicle {
  assert massConstraint {
    in partMasses = (chassisMass, engine.mass, transmission.mass);
    in massLimit = 2500[kg];
  }

  attribute chassisMass : MassValue;

  attribute engine : Engine {
    value mass : MassValue;
  }

  attribute transmission : Engine {
    value mass : MassValue;
  }
}
```

A named constraint can be asserted in multiple contexts.

Derivation Constraints

In UML and SysML v1, constraints are often used to define *derived values*.

```

part vehicle1 : Vehicle {
    attribute totalMass : MassValue;
    assert constraint {totalMass == chassisMass + engine.mass + transmission.mass}
}

part vehicle2 : Vehicle {
    attribute totalMass : MassValue = chassisMass + engine.mass + transmission.mass;
}

constraint def AveragedDynamics {
    in mass: MassValue;
    in initialSpeed : SpeedValue;
    in finalSpeed : SpeedValue;
    in deltaT : TimeValue;
    in force : ForceValue;

    force * deltaT == mass * (finalSpeed - initialSpeed) and
    mass > 0[kg]
}
    
```

In SysML v2 this can usually be done more directly using a binding.

⚠ Be careful about the difference between `==`, which is the Boolean-valued equality operator, and `=`, which denotes binding.

However, constraints allow for more general equalities and inequalities than direct derivation.

Analytical Constraints

```
constraint def StraightLineDynamicsEquations {
  in p : PowerValue, in m : MassValue, in dt : TimeValue;
  in x_i : LengthValue; in v_i : SpeedValue;
  in x_f : LengthValue; in v_f : SpeedValue,
  in a : AccelerationValue;
```

```
  attribute v_avg : SpeedValue = (v_i + v_f)/2;
```

```
  a == Acceleration(p, m, v_avg) and
  v_f == Velocity(dt, v_i, a) and
  x_f == Position(dt, x_i, v_avg)
```

```
}
```

```
action def StraightLineDynamics {
  in power : PowerValue; in mass : MassValue; in delta_t : TimeValue;
  in x_in : LengthValue; in v_in : SpeedValue;
  out x_out : LengthValue; out v_out : SpeedValue;
  out a_out : AccelerationValue;
```

```
  assert constraint dynamics : StraightLineDynamicsEquations {
    in p = power; in m = mass; in dt = delta_t;
    in x_i = x_in; in v_i = v_in;
    in x_f = x_out; in v_f = v_out;
    in a = a_out;
```

```
  }
```

```
}
```

This constraint definition provides a reusable specification of a system of (coupled) equations.

① Note the use of the calculation definitions defined earlier.

An action definition is inherently "causal" in the sense that outputs are determined in terms of inputs.

A constraint is inherently "acausal" – it is simply true or false for given values of its parameters.

① This specifies that the action outputs must be solved for analytically given the action inputs, consistent with the asserted constraint.


```

state healthStates {
  in vehicle : Vehicle;

  entry; then normal;

  state normal;
  accept at vehicle.maintenanceTime
    then maintenance;

  state maintenance {
    assert constraint { TimeOf(maintenance) > vehicle.maintenanceTime }
    assert constraint { TimeOf(maintenance) - TimeOf(normal.done) < 2 [s] }
  }

  entry assign vehicle.maintenanceTime :=
    vehicle.maintenanceTime + vehicle.maintenanceInterval;
}
accept MaintenanceDone
  then normal;

constraint { DurationOf(maintenance) <= 48 [h] }
}

```

① TimeOf and DurationOf are from the Time package in the Quantities and Units Domain Library.

TimeOf returns the time of the start of an occurrence (in this example, a state performance).

DurationOf returns the time duration of an occurrence (the different between its start and done times).

Requirement Definitions (1)

A *requirement definition* is a special kind of constraint definition.

A textual statement of the requirement can be given as a documentation comment in the requirement definition body.

```
requirement def MassLimitationRequirement {
  doc /* The actual mass shall be less than or equal
    * to the required mass. */

  attribute massActual : MassValue;
  attribute massReqd : MassValue;

  require constraint { massActual <= massReqd }
}
```

Like a constraint definition, a requirement definition can be parameterized using features.

The requirement can be formalized by giving one or more component *required constraints*.

Requirement Definitions (2)

```
part def Vehicle {
  attribute dryMass: MassValue;
  attribute fuelMass: MassValue;
  attribute fuelFullMass: MassValue;
  ...
}
```

A requirement definition may have a modeler specified *short name*, which is an alternate name for it.

① Actually, any identifiable element can have a short name, not just requirements.

```
requirement def <'1'> VehicleMassLimitationRequirement :> MassLimitationRequirement {
  doc /* The total mass of a vehicle shall be less than or equal to the required mass. */

  subject vehicle : Vehicle;

  attribute redefines massActual = vehicle.dryMass + vehicle.fuelMass;

  assume constraint { vehicle.fuelMass > 0[kg] }
}
```

A requirement definition is always about some *subject*, which may be implicit or specified explicitly.

A requirement definition may also specify one or more *assumptions*.

Features of the subject can be used in the requirement definition.

Requirement Definitions (3)

The subject of a requirement definition can have any kind of definition.

```
port def ClutchPort;
action def GenerateTorque;

requirement def <'2'> DrivePowerInterfaceRequirement {
  doc /* The engine shall transfer its generated torque to the transmission
    * via the clutch interface. */
  subject clutchPort: ClutchPort;
}

requirement def <'3'> TorqueGenerationRequirement {
  doc /* The engine shall generate torque as a function of RPM as shown in Table 1. */
  subject generateTorque: GenerateTorque;
}
```

A requirement may optionally have its own short name.

A *requirement* is the usage of a requirement definition.

```
requirement <'1.1'> fullVehicleMassLimit : VehicleMassLimitationRequirement {
  subject vehicle : Vehicle;
  attribute :>> massReqd = 2000[kg];

  assume constraint {
    doc /* Fuel tank is full. */
    vehicle.fuelMass == vehicle.fuelFullMass
  }
}

requirement <'1.2'> emptyVehicleMassLimit : VehicleMassLimitationRequirement {
  subject vehicle : Vehicle;
  attribute :>> massReqd = 1500[kg];

  assume constraint {
    doc /* Fuel tank is empty. */
    vehicle.fuelMass == 0[kg]
  }
}
```

A requirement will often bind requirement definition parameters to specific values.

Requirement Groups

A requirement may also be used to group other requirements.

① Grouped requirements are treated as required constraints of the group.

Requirements can be grouped by reference...

By default, the subject of grouped requirements is assumed to be the same as that of the group.

...or by composition.

The subject of a grouped requirement can also be bound explicitly.

```
requirement vehicleSpecification {
  doc /* Overall vehicle requirements group */
  subject vehicle : Vehicle;

  require fullVehicleMassLimit;
  require emptyVehicleMassLimit;
}

part def Engine {
  port clutchPort: ClutchPort;
  perform action generateTorque: GenerateTorque;
}

requirement engineSpecification {
  doc /* Engine power requirements group */
  subject engine : Engine;

  requirement drivePowerInterface : DrivePowerInterfaceRequirement {
    subject clutchPort = engine.clutchPort;
  }

  requirement torqueGeneration : TorqueGenerationRequirement {
    subject generateTorque = engine.generateTorque;
  }
}
```

```

part vehicle_c1 : Vehicle {
    part engine_v1: Engine { ... }
    ...
}

part 'Vehicle c1 Design Context' {
    ref vehicle_design :> vehicle_c1;

    satisfy vehicleSpecification by vehicle_design;
    satisfy engineSpecification by vehicle_design.engine_v1;
}
    
```

A *requirement satisfaction* asserts that a given requirement is satisfied when its subject parameter is bound to a specific thing.

① Formally, a requirement is *satisfied* for a subject if, when all its assumed constraints are true, then all its required constraints are true.

Analysis Case Definitions (1)

An *analysis case definition* defines the computation of the result of analyzing some *subject*, meeting an *objective*.

The subject may be specified similarly to the subject of a requirement definition.

The objective is a requirement on the result of the analysis case.

The analysis objective is specified as a requirement, allowing both assumed and required constraints.

The analysis result is declared as a return result (as for a calculation definition).

```
analysis def FuelEconomyAnalysis {
  subject vehicle : Vehicle;

  objective fuelEconomyAnalysisObjective {
    doc /*
     * The objective of this analysis is to determine whether the
     * subject vehicle can satisfy the fuel economy requirement.
     */

    assume constraint {
      vehicle.wheelDiameter == 33['in'] and
      vehicle.driveTrainEfficiency == 0.4
    }

    require constraint {
      fuelEconomyResult > 30[mi / gallon]
    }
  }
  ...

  return fuelEconomyResult : DistancePerVolumeValue = ...;
}
```


Analysis Case Definitions (2)

The *steps* of an analysis case are actions that, together, compute the analysis result.

The first step solves for the engine power needed for a given position/velocity scenario.

The second step computes the fuel economy result, given the power profile determined in the first step.

```

attribute def WayPoint {
    time : TimeValue;
    position : LengthValue;
    speed : SpeedValue;
}

analysis def FuelEconomyAnalysis {
    subject vehicle : Vehicle;
    objective fuelEconomyAnalysisObjective { ... }
    in attribute scenario : WayPoint[*];
    action solveForPower {
        out power: PowerValue[*];
        out acceleration: AccelerationValue[*];
        assert constraint {
            (1..size(scenario)-1)->forAll {in i : Positive;
                StraightLineDynamicsEquations (
                    power#(i), vehicle.mass,
                    scenario.time#(i+1) - scenario.time#(i),
                    scenario.position#(i), scenario.speed#(i),
                    scenario.position#(i+1), scenario.speed#(i+1),
                    acceleration#(i+1))
            }
        }
    }
    then action solveForFuelEconomy {
        in power : PowerValue[*] = solveForPower.power;
        out fuelEconomy : DistancePerVolumeValue;
        ...
    }
    return fuelEconomyResult : DistancePerVolumeValue;
        = solveForFuelEconomy.fuelEconomy;
    ... }

```

Additional parameters can be specified in the case body.

```

part vehicleFuelEconomyAnalysisContext {

    requirement vehicleFuelEconomyRequirements{subject vehicle : Vehicle; ... }

    attribute cityScenario : WayPoint[*] = { ... };
    attribute highwayScenario : WayPoint[*] = { ... };

    analysis cityAnalysis : FuelEconomyAnalysis {
        subject vehicle = vehicle_c1;
        in scenario = cityScenario;
    }

    analysis highwayAnalysis : FuelEconomyAnalysis {
        subject vehicle = vehicle_c1;
        in scenario = highwayScenario;
    }

    part vehicle_c1 : Vehicle {
        ...
        attribute :>> fuelEconomy_city = cityAnalysis.fuelEconomyResult;
        attribute :>> fuelEconomy_highway = highwayAnalysis.fuelEconomyResult;
    }

    satisfy vehicleFuelEconomyRequirements by vehicle_c1;
}

```

The previously defined analysis is carried out for a specific vehicle configuration for two different scenarios.

The subject and parameters are automatically redefined, so redefinition does not need to be specified explicitly.

If the vehicle fuel economy is set to the results of the analysis, then this configuration is asserted to satisfy the desired fuel economy requirements.

Trade-off Study Analysis

The **TradeStudy** analysis definition from the **TradeStudies** domain library model provides a general framework for defining basic trade study analyses.

① A trade-off study is an analysis with the objective of selecting the optimum alternative from a given set based on an evaluation of each alternative.

```
analysis engineTradeStudy : TradeStudies::TradeStudy {
  subject : Engine = (engine4cyl, engine6cyl);
  objective : MaximizeObjective;

  calc :>> evaluationFunction {
    in part anEngine :>> alternative : Engine;
    calc powerRollup: PowerRollup {in engine = anEngine,
      return power : ISQ::PowerValue; }
    calc massRollup: MassRollup {in engine = anEngine;
      return mass : ISQ::MassValue; }
    calc efficiencyRollup: EfficiencyRollup {in engine = anEngine;
      return efficiency : Real; }
    calc costRollup: CostRollup {in engine = anEngine;
      return cost : Real; }
    return :>> result : Real = EngineEvaluation(
      powerRollup.power, massRollup.mass,
      efficiencyRollup.efficiency, costRollup.cost
    );
  }

  return part :>> selectedAlternative : Engine;
}
```

Bind the analysis subject to the set of study alternatives. Select either **MaximizeObjective** or **MinimizeObjective**.

Redefine the **evaluationFunction** calculation to provide an evaluation of *each one* of the alternatives.

The result of the analysis will be the alternative with either the maximum or minimum evaluated value.

Verification Case Definitions (1)

```
requirement vehicleMassRequirement {
  subject vehicle : Vehicle;
  in massActual :> ISQ::mass;
  doc /* The vehicle mass shall be less
       * than or equal to 2500 kg. */
  require constraint {
    massActual == vehicle.mass and
    massActual <= 2500[SI::kg] }
}
verification def VehicleMassTest {
  private import VerificationCases::*;

  subject testVehicle : Vehicle;

  objective vehicleMassVerificationObjective {
    verify vehicleMassRequirement;
  }

  return verdict : VerdictKind;
}
```

Parameterizing the requirement allows it to be checked against a measured `massActual`, while asserting (below) that this must be equal to the `vehicle.mass`.

A *verification case definition* defines a process for verifying whether a *subject* satisfies one or more requirements.

The subject may be specified similarly to the subject of a requirement or analysis case.

The requirements to be verified are declared in the verification case objective. The subject of the verification case is automatically bound to the subject of the verified requirements.

A verification case always returns a verdict of type `VerdictKind`. (The default name is `verdict`, but a different name can be used if desired.)

❗ `VerdictKind` is an *enumeration* with allowed values `pass`, `fail`, `inconclusive` and `error`.

Verification Case Definitions (2)

The *steps* of a verification case are actions that, together, determine the verdict.

PassIf is a utility function that returns a **pass** or **fail** verdict depending on whether its argument is true or false.

❶ The use of named argument notation here is optional.

```
verification def VehicleMassTest {
  private import VerificationCases::*;

  subject testVehicle : Vehicle;
  objective vehicleMassVerificationObjective {
    verify vehicleMassRequirement;
  }

  action collectData {
    in part testVehicle : Vehicle = VehicleMassTest::testVehicle;
    out massMeasured :> ISQ::mass;
  }

  action processData {
    in massMeasured :> ISQ::mass = collectData.massMeasured;
    out massProcessed :> ISQ::mass;
  }

  action evaluateData {
    in massProcessed :> ISQ::mass = processData.massProcessed;
    out verdict : VerdictKind =
      PassIf(vehicleMassRequirement(
        vehicle = testVehicle,
        massActual = massProcessed));
  }

  return verdict : VerdictKind = evaluateData.verdict;
}
```

This is a *check* of whether the requirement is satisfied for the given parameter values.

Verification Case Usages (1)

This is a *verification case usage* in which the subject has been restricted to a specific test configuration.

A verification case can be performed as an action by a verification system.

Parts of the verification system can perform steps in the overall verification process.

```

part vehicleTestConfig : Vehicle { ... }

verification vehicleMassTest : VehicleMassTest {
  subject testVehicle :> vehicleTestConfig;
}

part massVerificationSystem : MassVerificationSystem {
  perform vehicleMassTest;

  part scale : Scale {
    perform vehicleMassTest.collectData {
      in part :>> testVehicle;

      bind measurement = testVehicle.mass;

      out :>> massMeasured = measurement;
    }
  }
}

```

In reality, this would be some more involved process to determine the measured mass.

Verification Case Usages (2)

Individuals can be used to model the carrying out of actual tests.

⚠ The keyword **action** is required here to permit the local redefinition.

```
individual def TestSystem :> MassVerificationSystem;
individual def TestVehicle1 :> Vehicle;
individual def TestVehicle2 :> Vehicle;

individual testSystem : TestSystem :> massVerificationSystem {

  timeslice test1 {
    perform action :>> vehicleMassTest {
      individual :>> testVehicle : TestVehicle1 {
        :>> mass = 2500[SI::kg];
      }
    }
  }

  then timeslice test2 {
    perform action :>> vehicleMassTest {
      individual :>> testVehicle : TestVehicle2 {
        :>> mass = 3000[SI::kg];
      }
    }
  }
}
```

The test on the individual **TestVehicle1** should pass.

The test on the individual **TestVehicle2** should fail.

Use Case Definition

An *actor* is a parameter of the use case representing a role played by entity external to the subject.

The *objective* of the use case is for the subject to provide a result of value to one or more of the actors.

① Actors may also be specified for other kinds of cases and for requirements.

A *use case definition* defines a required interaction between its *subject* and certain external *actors*.

```
use case def 'Provide Transportation' {
  subject vehicle : Vehicle;
  actor driver : Person;
  actor passengers : Person[0..4];
  actor environment : Environment;
  objective {
    doc /* Transport driver and passengers from
       * starting location to ending location.
       */
  }
}

use case def 'Enter Vehicle' {
  subject vehicle : Vehicle;
  actor driver : Person;
  actor passengers : Person[0..4];
}

use case def 'Exit Vehicle' {
  subject vehicle : Vehicle;
  actor driver : Person;
  actor passengers : Person[0..4];
}
```

⚠ Actors are (referential) part usages and so must have part definitions.

Use Case Usage

The required behavior of a use case can be specified as for an action.

A use case can define sub-use cases within its behavior.

A use case can *include* the performance of a use case defined elsewhere (similar to performing an action).

⚠ The traditional use case “extend” relationship is not supported yet.

A use case can also be specified without an explicit use case definition.

```
use case 'provide transportation' : 'Provide Transportation' {
  subject vehicle;
  first start;
  then include use case 'enter vehicle' : 'Enter Vehicle' {
    subject vehicle;
    actor :>> driver = 'provide transportation'::driver;
    actor :>> passengers = 'provide transportation'::passengers;
  }
  then use case 'drive vehicle' {
    subject vehicle;
    actor driver = 'provide transportation'::driver;
    actor environment = 'provide transportation'::environment;
    include 'add fuel'[0..*] {
      subject vehicle;
      actor :>> fueler = driver;
    }
  }
  then include use case 'exit vehicle' : 'Exit Vehicle' {
    subject vehicle;
    actor :>> driver = 'provide transportation'::driver;
    actor :>> passengers = 'provide transportation'::passengers;
  }
  then done;
}
use case 'add fuel' {
  subject vehicle : Vehicle,
  actor fueler : Person;
  actor 'fuel station' : 'Fuel Station';
}
```

⚠ A subject must be declared first, before any actors. It can only be omitted if there are no actors or other parameters.

The subject of a nested use case is implicitly bound to that of its containing use case, but actors need to be explicitly bound.

Variation Definitions

```

attribute def Diameter :> Real;

part def Cylinder {
    attribute diameter : Diameter[1];
}

part def Engine {
    part cylinder : Cylinder[2..*];
}
part '4cylEngine' : Engine {
    part redefines cylinder[4];
}
part '6cylEngine' : Engine {
    part redefines cylinder[6];
}

// Variability model

variation attribute def DiameterChoices :> Diameter {
    variant attribute diameterSmall = 70[mm];
    variant attribute diameterLarge = 100[mm];
}
variation part def EngineChoices :> Engine {
    variant '4cylEngine';
    variant '6cylEngine';
}
    
```

Any kind of definition can be marked as a *variation*, expressing variability within a product line model.

A variation definition will typically specialize a definition from a design model, representing the type of thing being varied. The variants must then be valid usages of this type.

A variation defines one or more *variant* usages, which represent the allowed choices for that variation.

Variants can also be declared by reference to usages defined elsewhere.

Variation Usages

Any kind of usage can also be a variation, defining allowable variants without a separate variation definition.

```
abstract part vehicleFamily : Vehicle {
    part engine : EngineChoices[1];

    variation part transmission : Transmission[1] {
        variant manualTransmission;
        variant automaticTransmission;
    }

    assert constraint {
        (engine == engine::'4cylEngine' and
         transmission == transmission::manualTransmission) xor
        (engine == engine::'6cylEngine' and
         transmission == transmission::automaticTransmission)
    }
}
```

A constraint can be used to model restrictions across the choices that can be made.

A variation definition can be used like any other definition, but valid values of the usage are restricted to the allowed variants of the variation.

① The operator `xor` means "exclusive or". So, this constraint means "choose either a `4cylEngine` and a `manualTransmission`, or a `6cylEngine` and an `automaticTransmission`".

Variation Configuration

An element from a variability model with variation usages can be *configured* by specializing it and making selections for each of the variations.

A selection is made for a variation by binding one of the allowed variants to the variation usage.

```
part vehicle4Cyl :> vehicleFamily {
    part redefines engine = engine::'4cylEngine';
    part redefines transmission = transmission::manualTransmission;
}

part vehicle6Cyl :> vehicleFamily {
    part redefines engine = engine::'6cylEngine';
    part redefines transmission = transmission::manualTransmission;
}
```

Choosing a [manualTransmission](#) with a [6cylEngine](#) is not allowed by the constraint asserted on [vehicleFamily](#), so this model of [vehicle6Cyl](#) is invalid.

```
package 'Dependency Example' {
  part 'System Assembly' {
    part 'Computer Subsystem' {
      ...
    }
    part 'Storage Subsystem' {
      ...
    }
  }
  package 'Software Design' {
    item def MessageSchema {
      ...
    }
    item def DataSchema {
      ...
    }
  }
}
```

```
dependency from 'System Assembly'::'Computer Subsystem' to 'Software Design';
```

```
dependency Schemata
  from 'System Assembly'::'Storage Subsystem'
  to 'Software Design'::MessageSchema, 'Software Design'::DataSchema;
}
```

① A dependency is a relationship that indicates that one or more client elements require one more supplier elements for their complete specification. A dependency is entirely a model-level relationship, without instance-level semantics.

A dependency can be between any kinds of elements, generally meaning that a change to a supplier may necessitate a change to the client element.

A dependency can have multiple clients and/or suppliers.

An *allocation* specifies that some or all of the responsibility for realizing the intent of the source is allocated to the target.

```
package LogicalModel {
    ...
    part torqueGenerator : TorqueGenerator {
        perform generateTorque;
    }
}

package PhysicalModel {
    private import LogicalModel::*;
    ...
    part powerTrain : PowerTrain {
        part engine : Engine {
            perform generateTorque;
        }
    }

    allocate torqueGenerator to powerTrain {
        allocate torqueGenerator.generateTorque
            to powerTrain.engine.generateTorque;
    }
}
```

① Allocations define traceable links across the various structures and hierarchies of a system model, perhaps as a precursor to more rigorous specifications and implementations.

An allocation can be refined using nested allocations that give a finer-grained decomposition of the containing allocation mapping.

⚠ Unlike SysML v1, an allocation in SysML v2 is *not* a dependency but, rather, an instantiable connection across model features.

An *allocation definition* defines a class of allocations between features of specific types.

An *allocation usage* must allocate features that conform to the types of the ends of its allocation definition.

```
package LogicalModel {
    ...
    part def LogicalElement;
    part def TorqueGenerator :> LogicalElement;

    part torqueGenerator : TorqueGenerator { ... }
}

package PhysicalModel {
    private import LogicalModel::*;
    ...
    part def PhysicalElement;
    part def PowerTrain :> PhysicalElement;

    part powerTrain : PowerTrain { ... }
}

allocation def LogicalToPhysical {
    end logical : LogicalElement;
    end physical : PhysicalElement;
}

allocation torqueGenAlloc : LogicalToPhysical
    allocate torqueGenerator to powerTrain { ... }
}
```

Metadata (1)

① Metadata is additional data that can be used to annotate the elements of a model.

Metadata is defined using a *metadata definition*.

A specific type of metadata can then be applied as an annotation to one or more model elements.

```
part vehicle {
  part interior {
    part alarm;
    part seatBelt[2];
    part frontSeat[2];
    part driverAirBag;
  }
  part bodyAssy {
    part body;
    part bumper;
    part keylessEntry;
  }
}
```

```
metadata def SafetyFeature;
metadata def SecurityFeature {
  :> annotatedElement : SysML::PartDefinition;
  :> annotatedElement : SysML::PartUsage;
}
metadata SafetyFeature about
  vehicle::interior::seatBelt,
  vehicle::interior::driverAirBag,
  vehicle::bodyAssy::bumper;

metadata SecurityFeature about
  vehicle::interior::alarm,
  vehicle::bodyAssy::keylessEntry;
```

① The [KerML](#) and [SysML](#) library models include all the metaclasses from the KerML and SysML abstract syntax.

To restrict what kind of element can be annotated, subset the [annotatedElement](#) feature with the desired element type(s).

① At its simplest, a metadata annotation can be used to simply "tag" certain elements, so that they can, e.g., be grouped by tooling.

Metadata (2)

```
standard library package AnalysisTooling {
  private import ScalarValues::*;
  metadata def ToolExecution {
    attribute toolName : String;
    attribute uri : String;
  }
  metadata def ToolVariable {
    attribute name : String;
  }
}
```

① The **AnalysisTooling** package is part of the **Analysis** domain library.

```
action computedDynamics {
  private import AnalysisTooling
```

If the metadata definition has nested features, these must be bound to values in the metadata annotation.

```
  metadata ToolExecution {
    toolName = "ModelCenter";
    uri = "aserv://localhost/Vehicle/Equation1";
  }
```

The **@** symbol is equivalent to the **metadata** keyword.

```
  in dt : ISQ::TimeValue  {@ToolVariable {name = "deltaT";}}
  in a : ISQ::AccelerationValue  {@ToolVariable {name = "mass";}}
  in v_in : ISQ::VelocityValue  {@ToolVariable {name = "v0";}}
  in x_in : ISQ::LengthValue  {@ToolVariable {name = "x0";}}

  out v_out : ISQ::VelocityValue  {@ToolVariable {name = "v";}}
  out x_out : ISQ::LengthValue  {@ToolVariable {name = "x";}}
}
```

A metadata annotation contained in the body of a namespace (package, definition or usage) is, by default, about that namespace.

Element Import Filtering (1)

```
metadata def Safety {
  attribute isMandatory : Boolean;
}
```

```
part vehicle {
  part interior {
    part alarm;
    part seatBelt[2] {
      @Safety{isMandatory = true;}}
    part frontSeat[2];
    part driverAirBag {
      @Safety{isMandatory = false;}}
  }
  part bodyAssy {
    part body;
    part bumper {
      @Safety{isMandatory = true;}}
    part keylessEntry;
  }
  part wheelAssy {
    part wheel[2];
    part antilockBrakes[2] {
      @Safety{isMandatory = false;}}
  }
}
```

⚠ Currently, a metadata annotation must be owned by the annotated element to be accessed in a filter condition.

```
package 'Safety Features' {
  public import vehicle::**;
  filter @Safety;
}
```

```
package 'Mandatory Safety Features' {
  public import vehicle::**;
  filter @Safety and (as Safety).isMandatory;
}
```

A recursive import (using ******) imports members of a namespace *and*, recursively, members of any nested namespaces.

A *filter* is a Boolean condition that must be true for an element to actually be imported into a package.

@ checks if an element has the given metadata and **as** "casts" to that metadata type.

Element Import Filtering (2)

```

metadata def Safety {
  attribute isMandatory : Boolean;
}

part vehicle {
  part interior {
    part alarm;
    part seatBelt[2] {
      @Safety{isMandatory = true;}}
    part frontSeat[2];
    part driverAirBag {
      @Safety{isMandatory = false;}}
  }
  part bodyAssy {
    part body;
    part bumper {
      @Safety{isMandatory = true;}}
    part keylessEntry;
  }
  part wheelAssy {
    part wheel[2];
    part antilockBrakes[2] {
      @Safety{isMandatory = false;}}
  }
}

```

① The **filter** keyword is only allowed in a package (or view), but a filtered import can be used anywhere an import is allowed.

Filter conditions can also be combined with the import itself.

```

package 'Safety Features' {
  public import vehicle::**[@Safety];
}

package 'Mandatory Safety Features' {
  public import vehicle::**
    [@Safety and (as Safety).isMandatory];
}

```

⚠ A filter condition expression must be model-level evaluable. It can reference only literals and metadata annotations and attributes, connected using basic arithmetic, comparison and Boolean operators.

Language Extension (1)

For a domain-specific extension to SysML, first create a library model of domain-specific concepts. This can be explicitly identified as a **library package**.

```
library package FailureModeling {
  abstract occurrence def Situation;
  abstract occurrence situations : Situation[*] nonunique;

  abstract occurrence def Cause {
    attribute probability : Real;
  }
  abstract occurrence causes : Cause[*] nonunique :> situations;

  abstract occurrence def Failure {
    attribute severity : Level;
  }
  abstract occurrence failures : Failure[*] nonunique :> situations;

  abstract connection def Causation :> Occurrences::HappensBefore {
    end cause : Situation[*];
    end effect : Situation[*];
  }
  abstract connection causations : Causation[*] nonunique;

  item def Scenario {
    occurrence :>> situations;
    occurrence :>> causes :> situations;
    occurrence :>> failures :> situations;
  }
  item scenarios : Scenario[*] nonunique;
}
```

For each concept, provide a base definition and a base usage modeling the concept semantics.

HappensBefore is the base type for successions, from the Kernel Library **Occurrences** package.

Language Extension (2)

```
library package FailureModelingMetadata {
  private import FailureModeling::*;
  private import Metaobjects::SemanticMetadata;

  metadata def situation :> SemanticMetadata {
    :>> baseType = situations meta SysML::Usage;
  }

  metadata def cause :> SemanticMetadata {
    :>> baseType = causes meta SysML::Usage;
  }

  metadata def failure :> SemanticMetadata {
    :>> baseType = failures meta SysML::Usage;
  }

  metadata def causation :> SemanticMetadata {
    :>> baseType = causations meta SysML::Usage;
  }

  metadata def scenario :> SemanticMetadata {
    :>> baseType = scenarios meta SysML::Usage;
  }
}
```

Declare *semantic metadata* for each concept to be included in the language extension.

Bind `baseType` to the base *usage* for the relevant concept.

The *meta-cast* "`meta SysML::Usage`" allows `failures` to be referenced as a usage, rather than evaluated as an expression.

Language Extension (3)

A user-defined keyword, starting with **#** and referencing a semantic metadata definition, can be used to declare a definition or usage.

Implicitly specializes the library usage **situations** (as given in the semantic metadata **situation**).

Inheritance and redefinition work as for explicit specialization.

```
private import FailureModelingMetadata::*;

#scenario def DeviceFailure {
  ref device : Device;
  attribute minPower : Real;

  #cause 'battery old' {
    :>> probability = 0.01;
  }

  #causation connect 'battery old' to 'power low';

  #situation 'power low' {
    constraint { device.battery.power < minPower }
  }

  #causation connect 'power low' to 'device shutoff';

  #failure 'device shutoff' {
    :>> severity = LevelEnum::high;
  }
}
```

Implicitly specializes the library definition **Scenario** (the definition of the **scenarios** base type).

Stakeholders and Concerns

```
part def 'Systems Engineer';
part def 'IV&V';
```

A *concern* is a specific topic that one or more stakeholders desire to be addressed.

```
concern 'system breakdown' {
  doc /*
    * To ensure that a system covers all its required capabilities,
    * it is necessary to understand how it is broken down into
    * subsystems and components that provide those capabilities.
    */
```

```
  subject system;
  stakeholder se: 'systems engineer';
  stakeholder ivv: 'IV&V';
}
```

⚠ A subject must be declared first, before any stakeholders. It can only be omitted if there are no stakeholders or other parameters.

```
concern modularity {
  doc /*
    * There should be well defined interfaces between the parts of
    * a system that allow each part to be understood individually,
    * as well as being part of the whole system.
    */
```

A *stakeholder* is a person, organization or other entity with concerns to be addressed.

```
  subject system;
  stakeholder 'systems engineer';
}
```

ℹ Stakeholders may be specified for any kind of requirement, not just concerns.

A *viewpoint* is a requirement to present information from a model in a *view* that addresses certain stakeholder concerns.

```
viewpoint 'system structure perspective' {
  frame 'system breakdown';
  frame modularity;

  require constraint {
    doc /*
      * A system structure view shall show the hierarchical
      * part decomposition of a system, starting with a
      * specified root part.
      */
  }
}
```

A viewpoint *frames* the concerns that will be addressed by a view that satisfies the viewpoint.

❶ Any requirement may be modeled as framing relevant stakeholder concerns, not just viewpoints.

Views (1)

A view definition can filter the elements to be included in the views it specifies.

```
view def 'Part Structure View' {
    satisfy 'system structure perspective';

    filter @SysML::PartUsage;
}

view 'vehicle structure view' :
    'Part Structure View' {

    expose vehicle::**;

    render asTreeDiagram;
}
```

A *view definition* specifies how information can be extracted from a model in order to satisfy one or more viewpoints.

The [SysML](#) library package models the SysML abstract syntax, which can be used to filter on specific kinds of model elements.

A *view usage* specifies a certain view conforming to a view definition.

A view usage *exposes* the model elements to be included in the view, which are filtered as specified in the view definition.

❶ The view rendering can also be given in the view definition, in which case it will be the same for all usages of that definition. But only view usages can expose elements to be viewed.

A view usage specifies how the view is to be *rendered* as a physical artifact.

Views (2)

Specialized kinds of renderings can be defined in a user model.

① The Views library package includes four basic kinds of rendering: `asTreeDiagram`, `asInterconnectionDiagram`, `asTextualNotation` and `asElementTable`.

```
rendering asTextualNotationTable :> asElementTable {
  view :>> columnView[1] {
    render asTextualNotation;
  }
}

view 'vehicle tabular views' {
  view 'safety features view' : 'Part Structure View' {
    expose vehicle::**[@Safety];
    render asElementTable;
  }

  view 'non-safety features view' : 'Part Structure View' {
    expose vehicle::**[not (@Safety)];
    render asElementTable;
  }
}
```

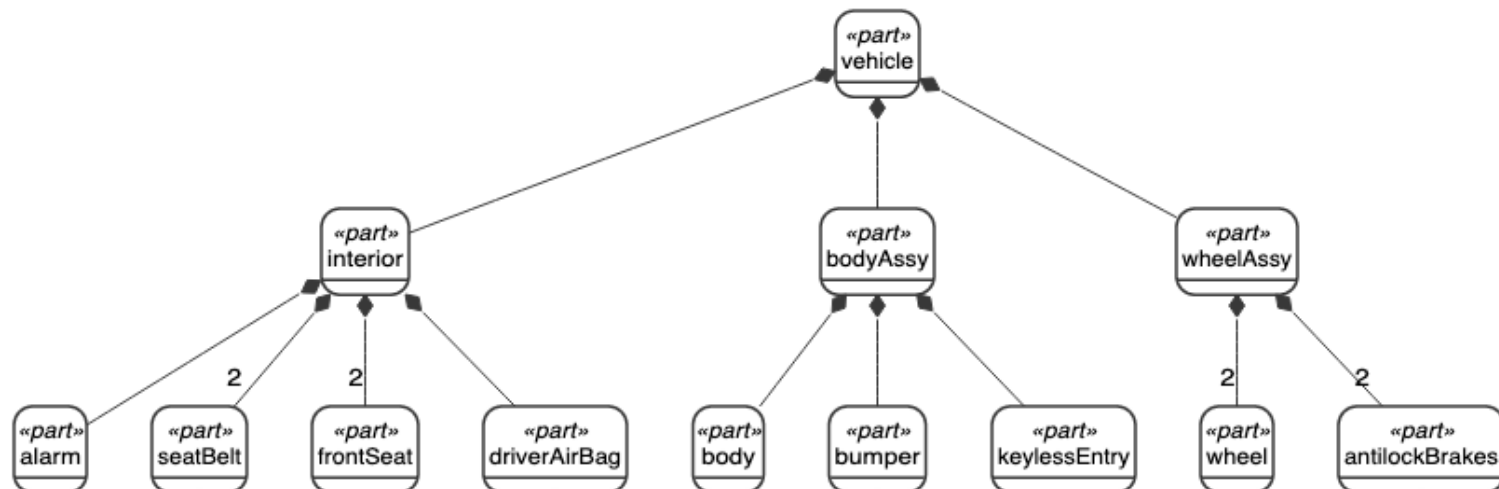
Views can have subviews.

⚠ A more comprehensive rendering library model is planned for future release.

Views can specify additional filtering conditions on an `expose`, using the same notation as for an `import`.

Example View Rendering

```
view 'vehicle structure view' {
  expose vehicle::**[@SysML::PartUsage];
  render asTreeDiagram;
}
```



Kernel Data Type Library

Scalar Values

```
standard library package ScalarValues {  
  private import Base::*;  
  
  abstract datatype ScalarValue specializes Value;  
  datatype Boolean specializes ScalarValue;  
  datatype String specializes ScalarValue;  
  abstract datatype NumericalValue specializes ScalarValue;  
  
  abstract datatype Number specializes NumericalValue;  
  datatype Complex specializes Number;  
  datatype Real specializes Complex;  
  datatype Rational specializes Real;  
  datatype Integer specializes Rational;  
  datatype Natural specializes Integer;  
  datatype Positive specializes Natural;  
}
```

Collections (1)

```

standard library package Collections {
  ...
  abstract datatype Collection {
    feature elements[0..*] nonunique;
  }
  abstract datatype OrderedCollection :> Collection {
    feature :>> elements[0..*] ordered nonunique;
  }
  abstract datatype UniqueCollection :> Collection {
    feature :>> elements[0..*];
  }

  datatype Array :> OrderedCollection {
    feature dimensions: Positive[0..*] ordered nonunique;
    feature rank: Natural[1] = size(dimensions);
    feature flattenedSize: Positive[1] = dimensions->reduce '*' ?? 1;
    inv { flattenedSize == size(elements) }
  }

  datatype Bag :> Collection;
  datatype Set :> UniqueCollection;
  datatype OrderedSet :> OrderedCollection, UniqueCollection {
    feature :>> elements[0..*] ordered;
  }
  datatype List :> OrderedCollection;
  ...
}

```

Collections (2)

```
standard library package Collections {  
  ...  
  
  datatype KeyValuePair {  
    feature key: Anything[0..*] ordered nonunique;  
    feature val: Anything[0..*] ordered nonunique;  
  }  
  
  datatype Map :> Collection {  
    feature :>> elements: KeyValuePair[0..*];  
  }  
  
  datatype OrderedMap :> Map {  
    feature :>> elements: KeyValuePair[0..*] ordered;  
  }  
}
```



Vector Values

```
standard library package VectorValues {  
  private import ScalarValues::NumericalValue;  
  private import ScalarValues::Real;  
  private import Collections::Array;  
  
  abstract datatype VectorValue;  
  
  datatype NumericalVectorValue :> VectorValue, Array {  
    feature dimension[0..1] :>> dimensions;  
    feature :>> elements : NumericalValue;  
  }  
  
  datatype CartesianVectorValue :> NumericalVectorValue {  
    feature :>> elements : Real;  
  }  
  
  datatype ThreeVectorValue :> NumericalVectorValue {  
    feature :>> dimension = 3;  
  }  
  
  datatype CartesianThreeVectorValue :> CartesianVectorValue, ThreeVectorValue;  
}
```


Kernel Function Library (selected models)

```

standard library package BaseFunctions {
  private import Base::Anything;
  private import ScalarValues::*;

  abstract function '==' { in x: Anything[0..1]; in y: Anything[0..1];
    return : Boolean[1];
  }
  function '!=' { in x: Anything[0..1]; in y: Anything[0..1];
    return : Boolean[1] = not (x == y)
  }
  abstract function '===' { in x: Anything[0..1]; in y: Anything[0..1];
    return : Boolean[1];
  }
  function '!===' { in x: Anything[0..1]; in y: Anything[0..1];
    return : Boolean[1] = not (x === y)
  }

  function ToString { in Anything[0..1];
    return : String[1];
  }

  abstract function '[' { in x: Anything[0..*] nonunique; in y: Anything[0..*] nonunique;
    return : Anything[0..*] nonunique;
  }
  abstract function '#' { in seq: Anything[0..*] ordered nonunique; in index: Positive[1..*] ordered nonunique;
    return : Anything[0..1];
  }
  abstract function ', '{
    in seq1: Anything[0..*] ordered nonunique; seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
  }
  ...
}

```

standard library package DataFunctions {

...
 abstract function '==' specializes BaseFunctions::'=='
 {in x: DataValue[0..1]; in y: DataValue[0..1]; return : Boolean[1];}

function '===' specializes BaseFunctions::'==='
 {in x: DataValue[0..1]; in y: DataValue[0..1]; return : Boolean[1];}

abstract function '+' {in x: DataValue[1]; in y: DataValue[0..1]; return : DataValue[1];}

abstract function '-' {in x: DataValue[1]; in y: DataValue[0..1]; return : DataValue[1];}

abstract function '*' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '/' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '**' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '^' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '%' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function 'not' {in x: DataValue[1]; return : DataValue[1];}

abstract function 'xor' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '~' {in x: DataValue[1]; return : DataValue[1];}

abstract function '|' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '&' {in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1];}

abstract function '<' {in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1];}

abstract function '>' {in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1];}

abstract function '<=' {in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1];}

abstract function '>=' {in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1];}

abstract function max {in x: DataValue[1]; return : DataValue[1];}

abstract function min {in x: DataValue[1]; return : DataValue[1];}

abstract function '..'
 {in lower:DataValue[1]; in upper:DataValue[1]; return : DataValue[0..*] ordered;

}

}

}

① **ScalarFunctions** package is similar but specialized for **ScalarValue**.

Boolean Functions

```
standard library package BooleanFunctions {
  public import ScalarValues::*;

  function 'not' specializes ScalarFunctions::'not'
    {in x: Boolean; return : Boolean;}
  function 'xor' specializes ScalarFunctions::'xor'
    {in x: Boolean; in y: Boolean; return : Boolean;}

  function '|' specializes ScalarFunctions::'|'
    {in x: Boolean; in y: Boolean; return : Boolean;}
  function '&' specializes ScalarFunctions::'&'
    {in x: Boolean; in y: Boolean; return : Boolean;}

  function '==' specializes BaseFunctions::'=='
    {in x: Boolean; in y: Boolean; return : Boolean;}

  function ToString specializes BaseFunctions::ToString
    {in x: Boolean; return : String;}
  function ToBoolean(x: String): Boolean;
    {in x: String; return : Boolean;}
}
```

String Functions

```
standard library package StringFunctions {
  public import ScalarValues::*;

  function '+' specializes ScalarFunctions::'+'
    {in x: String; in y: String; return : String;}

  function Size {in x: String; return : Natural;}
  function Substring
    {in x: String; in lower: Integer; in upper: Integer; return : String;}

  function '<' specializes ScalarFunctions::'<'
    {in x: String; in y: String; return : Boolean;}
  function '>' specializes ScalarFunctions::'>'
    {in x: String; in y: String; return : Boolean;}
  function '<=' specializes ScalarFunctions::'<='
    {in x: String; in y: String; return : Boolean;}
  function '>=' specializes ScalarFunctions::'>='
    {in x: String; in y: String; return : Boolean;}
  function '=' specializes BaseFunctions::'=='
    {in x: String; in y: String; return : Boolean;}

  function ToString specializes BaseFunctions::ToString
    {in x: String; return : String;
  }
```

```

standard library package NumericalFunctions {
  public import ScalarValues::*;
  function abs {in x: NumericalValue[1]; return : NumericalValue[1];}
  abstract function '+' specializes ScalarFunctions:: '+'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function '-' specializes ScalarFunctions:: '-'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function '*' specializes ScalarFunctions:: '*'
    {in x: NumericalValue[1]; in y: NumericalValue[1]; return : NumericalValue[1];}
  abstract function '/' specializes ScalarFunctions:: '/'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function '**' specializes ScalarFunctions:: '**'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function '^' specializes ScalarFunctions:: '^'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function '%' specializes ScalarFunctions:: '%'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function '<' specializes ScalarFunctions:: '<'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : Boolean[1];}
  abstract function '>' specializes ScalarFunctions:: '>'
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : Boolean[1];}
  abstract function '<=' specializes ScalarFunctions:: '<='
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : Boolean[1];}
  abstract function '>=' specializes ScalarFunctions:: '>='
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : Boolean[1];}
  abstract function max specializes ScalarFunctions:: max
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}
  abstract function min specializes ScalarFunctions:: min
    {in x: NumericalValue[1]; in y: NumericalValue[0..1]; return : NumericalValue[1];}

  abstract function sum {in collection: NumericalValue[0..*] ordered; return : NumericalValue[1];}
  abstract function product {in collection: NumericalValue[0..*] ordered; return : NumericalValue[1];}
}

```

Complex Functions

```

standard library package ComplexFunctions {
  public import ScalarValues::*;

  feature i: Complex[1] = rect(0.0, 1.0);

  function rect{in re: Real[1]; in im: Real[1]; return : Complex[1];}
  function polar{in abs: Real[1]; in arg: Real[1]; return : Complex[1];}
  function re{in x: Complex[1]; return : Real[1];}
  function im {in x: Complex[1]; return : Real[1];}

  function abs specializes NumericalFunctions::abs {in x: Complex[1]; return : Real[1];}
  function {in x: Complex[1]; return : Real[1];}
  function '+' specializes NumericalFunctions::'+'
    {in x: Complex[1]; in y: Complex[0..1]; return : Complex[1];}
  function '-' specializes NumericalFunctions::'-'
    {in x: Complex[1]; in y: Complex[0..1]; return : Complex[1];}
  function '*' specializes NumericalFunctions::'*'
    {in x: Complex[1]; in y: Complex[1]; return : Complex[1];}
  function '/' specializes NumericalFunctions::'/'
    {in x: Complex[1]; in y: Complex[1]; return : Complex[1];}
  function '**' specializes NumericalFunctions::'**'
    {in x: Complex[1]; in y: Complex[1]; return : Complex[1];}
  function '^' specializes NumericalFunctions::'^'
    {in x: Complex[1]; in y: Complex[1]; return : Complex[1];}
  function '==' specializes DataFunctions::'=='
    {in x: Complex[1]; in y: Complex[1]; return : Complex[1];}

  function ToString specializes BaseFunctions::ToString {in x: Complex[1]; return : String[1];}
  function ToComplex {in x: String[1]; return : Complex[1];}

  function sum specializes NumericalFunctions::sum {in collection: Complex[0..*]; return : Complex[1];}
  function product specializes NumericalFunctions::product {in collection: Complex[0..*]; return : Complex[1];}
}

```

```

standard library package RealFunctions {
  public import ScalarValues::*;
  function abs specializes ComplexFunctions::abs {in x: Real[1]; return : Real[1];}
  function '+' specializes ComplexFunctions::'+' {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function '-' specializes ComplexFunctions::'-' {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function '*' specializes ComplexFunctions::'*' {in x: Real[1]; in y: Real[1]; return: Real[1];}
  function '/' specializes ComplexFunctions::'/' {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function '**' specializes ComplexFunctions::'**' {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function '^' specializes ComplexFunctions::'^' {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function '<' specializes ComplexFunctions::'<'
    {in x: Real[1]; in y: Real[0..1]; return: Boolean[1];}
  function '>' specializes ComplexFunctions::'>'
    {in x: Real[1]; in y: Real[0..1]; return: Boolean[1];}
  function '<=' specializes ComplexFunctions::'<='
    {in x: Real[1]; in y: Real[0..1]; return: Boolean[1];}
  function '>=' specializes ComplexFunctions::'>='
    {in x: Real[1]; in y: Real[0..1]; return: Boolean[1];}
  function max specializes ComplexFunctions::max {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function min specializes ComplexFunctions::min {in x: Real[1]; in y: Real[0..1]; return: Real[1];}
  function '==' specializes ComplexFunctions::'=='
    {in x: Real[1]; in y: Real[0..1]; return: Boolean[1];}
  function sqrt {in x: Real[1]; return : Real[1];}
  function floor {in x: Real[1]; return : Integer[1];}
  function round {in x: Real[1]; return : Integer[1];}

  function ToString specializes ComplexFunctions::ToString {in x: Real[1]; return : String[1];}
  function ToInteger {in x: Real[1]; return : Integer[1];}
  function ToRational {in x: Real[1]; return : Rational[1];}
  function ToReal {in x: String[1]; return : Real[1];}
  function ToComplex(x: Real[1]): Complex;
  function sum specializes ComplexFunctions::sum {in collection: Real[0..*]; return : Real[1];}
  function product specializes ComplexFunctions::product {in collection: Real[0..*]; return : Real[1];}
}

```


Rational Functions

```

standard library package RationalFunctions {
  public import ScalarValues::*;

  function rat {in number: Integer[1]; in denum: Integer[1]; return : Rational[1];}
  function number {in rat: Rational[1]; return : Integer[1];}
  function denom {in rat: Rational[1]; return : Integer[1];}

  function abs specializes RealFunctions::abs {in x: Rational[1]; return : Rational[1];}
  function '+' specializes RealFunctions::'+' {in x: Rational[1]; in y: Rational[0..1]; return : Rational[1];}
  function '-' specializes RealFunctions::'-' {in x: Rational[1]; in y: Rational[0..1]; return : Rational[1];}
  function '*' specializes RealFunctions::'*' {in x: Rational[1]; in y: Rational[1]; return : Rational[1];}
  function '/' specializes RealFunctions::'/' {in x: Rational[1]; in y: Rational[1]; return : Rational[1];}
  function '**' specializes RealFunctions::'**' {in x: Rational[1]; in y: Rational[1]; return : Rational[1];}
  function '^' specializes RealFunctions::'^' {in x: Rational[1]; in y: Rational[1]; return : Rational[1];}
  function '<' specializes RealFunctions::'<' {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}
  function '>' specializes RealFunctions::'>' {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}
  function '<=' specializes RealFunctions::'<=' {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}
  function '>=' specializes RealFunctions::'>=' {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}
  function max specializes RealFunctions::max {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}
  function min specializes RealFunctions::min {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}
  function '==' specializes RealFunctions::'==' {in x: Rational[1]; in y: Rational[1]; return : Boolean[1];}

  function gcd {in x: Rational[1]; in y: Rational[0..1]; return : Integer[1];}
  function floor {in x: Rational[1]; return : Integer[1];}
  function round {in x: Rational[1]; return : Integer[1];}

  function ToString specializes RealFunctions::ToString {in x: Rational[1]; return : String[1];}
  function ToInteger {in x: Rational[1]; return : Integer[1];}
  function ToRational {in x: String[1]; return : Rational[1];}

  function sum specializes RealFunctions::sum {in collection: Rational[0..*]; return : Rational[1];}
  function product specializes RealFunctions::product {in collection: Rational[0..*]; return : Rational[1];}
}

```

Integer Functions

```

standard library package IntegerFunctions {
  public import ScalarValues::*;

  function abs specializes RationalFunctions::abs {in x: Integer[1]; return : Natural[1];}
  function '+' specializes RationalFunctions::'+' {in x: Integer[1]; in y: Integer[0..1]; return : Integer[1];}
  function '-' specializes RationalFunctions::'-' {in x: Integer[1]; in y: Integer[0..1]; return : Integer[1];}
  function '*' specializes RationalFunctions::'*' {in x: Integer[1]; in y: Integer[1]; return : Integer[1];}
  function '/' specializes RationalFunctions::'/' {in x: Integer[1]; in y: Integer[1]; return : Rational[1];}
  function '**' specializes RationalFunctions::'**' {in x: Integer[1]; in y: Natural[1]; return : Integer[1];}
  function '^' specializes RationalFunctions::'^' {in x: Integer[1]; in y: Natural[1]; return : Integer[1];}
  function '%' specializes NumericalFunctions::'%' {in x: Integer[1]; in y: Integer[1]; return : Integer[1];}
  function '<' specializes RationalFunctions::'<' {in x: Integer[1]; in y: Integer[1]; return : Boolean[1];}
  function '>' specializes RationalFunctions::'>' {in x: Integer[1]; in y: Integer[1]; return : Boolean[1];}
  function '<=' specializes RationalFunctions::'<=' {in x: Integer[1]; in y: Integer[1]; return : Boolean[1];}
  function '>=' specializes RationalFunctions::'>=' {in x: Integer[1]; in y: Integer[1]; return : Boolean[1];}

  function max specializes RationalFunctions::max {in x: Integer[1]; in y: Integer[1]; return : Integer[1];}
  function min specializes RationalFunctions::min {in x: Integer[1]; in y: Integer[1]; return : Integer[1];}

  function '==' specializes DataFunctions::'==' {in x: Integer[0..1]; in y: Integer[0..1]; return : Boolean[1];}
  function '..' specializes ScalarFunctions::'..'
    {in lower: Integer[1]; in upper: Integer[1]; return : Integer[0..*];}
  function ToString specializes RationalFunctions::ToString {in x: Integer[1]; return : String[1];}
  function ToNatural {in x: Integer[1]; return : Natural[1];}
  function ToInteger {in x: String[1]; return : Integer[1];}
  function sum specializes RationalFunctions::sum {in collection: Integer[0..*]; return : Integer[1];}
  function product specializes RationalFunctions::product {in collection: Integer[0..*]; return : Integer[1];}
}

```

```
standard library package NaturalFunctions {
  public import ScalarValues::*;

  function '+' specializes IntegerFunctions::'+' {in x: Natural[1]; in y: Natural[0..1]; return : Natural[1];}
  function '*' specializes IntegerFunctions::'*' {in x: Natural[1]; in y: Natural[1]; return : Natural[1];}
  function '/' specializes IntegerFunctions::'/' {in x: Natural[1]; in y: Natural[1]; return : Natural[1];}
  function '%' specializes IntegerFunctions::'%' {in x: Natural[1]; in y: Natural[1]; return : Natural[1];}
  function '<' specializes IntegerFunctions::'<' {in x: Natural[1]; in y: Natural[1]; return : Boolean[1];}
  function '>' specializes IntegerFunctions::'>' {in x: Natural[1]; in y: Natural[1]; return : Boolean[1];}
  function '<=' specializes IntegerFunctions::'<=' {in x: Natural[1]; in y: Natural[1]; return : Boolean[1];}
  function '>=' specializes IntegerFunctions::'>=' {in x: Natural[1]; in y: Natural[1]; return : Boolean[1];}

  function max specializes IntegerFunctions::max {in x: Natural[1]; in y: Natural[1]; return : Natural[1];}
  function min specializes IntegerFunctions::min {in x: Natural[1]; in y: Natural[1]; return : Natural[1];}

  function '==' specializes IntegerFunctions::'=='
    {in x: Natural[0..1]; in y: Natural[0..1]; return : Boolean[1];}
  function ToString specializes IntegerFunctions::ToString {in x: Natural[1]; return : String[1];}
  function ToNatural {in x: String[1]; return : Natural[1];}
}
```

Trigonometric Functions

```

standard library package TrigFunctions {
  public import ScalarValues::Real;

  feature pi : Real;
  inv piPrecision {
    RealFunctions::round(pi * 1E20) == 314159265358979323846.0
  }

  function deg {in theta_rad : Real[1]; return : Real[1];}
  function rad {in theta_deg : Real; return : Real[1];}

  datatype UnitBoundedReal :> Real {
    inv unitBound { -1.0 <= that & that <= 1.0 }
  }
  function sin {in theta : Real[1]; return : UnitBoundedReal[1];}
  function cos {in theta : Real[1]; return : UnitBoundedReal[1];}
  function tan {in theta : Real[1]; return : Real;}
  function cot {in theta : Real; return : Real;}
  function arcsin {in x : UnitBoundedReal[1]; return : Real[1];}
  function arccos {in x : UnitBoundedReal[1]; return : Real[1];}
  function arctan {in x : Real[1]; return : Real[1];}
}

```

Sequence Functions (1)

```

standard library package SequenceFunctions {
...
  function equals {in x: Anything[0..*] ordered nonunique; in y: Anything[0..*] ordered nonunique;
    return : Boolean[1];
  }
  function same {in x: Anything[0..*] ordered nonunique; in y: Anything[0..*] ordered nonunique;
    return : Boolean[1];
  }

  function size {in seq: Anything[0..*] ordered nonunique; return : Natural[1];}
  function isEmpty {in seq: Anything[0..*] ordered nonunique; return : Boolean[1];}
  function notEmpty {in seq: Anything[0..*] ordered nonunique; return : Boolean[1];}

  function includes {in seq1: Anything[0..*] ordered nonunique; in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
  }
  function excludes {in seq1: Anything[0..*] ordered nonunique; in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
  }
  function including {in seq: Anything[0..*] ordered nonunique; in values: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
  }
  function includingAt{in seq: Anything[0..*] ordered nonunique; in values: Anything[0..*] ordered nonunique;
    in index: Positive[1]; return : Anything[0..*] ordered nonunique;
  }
  function excluding {in seq: Anything[0..*] ordered nonunique; in values: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
  }
  function excludingAt{in seq: Anything[0..*] ordered nonunique; in values: Anything[0..*] ordered nonunique;
    in index: Positive[1]; return : Anything[0..*] ordered nonunique;
  }
...

```

Sequence Functions (2)

```

standard library package SequenceFunctions {
  ...

  function head {in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..1];
  }
  function tail {in seq: Anything[0..*] ordered nonunique ;
    return : Anything[0..*] ordered nonunique;
  }
  function last {in seq: Anything[0..*] ;
    return : Anything[0..1];
  }

  function '#' specializes BaseFunctions::'#' {
    in seq: Anything[0..*] ordered nonunique; in index: Natural[1]; return : Anything[0..1];
  }

  behavior add {
    inout seq: Anything[0..*] ordered nonunique; in values: Anything[0..*] ordered nonunique;
  }
  behavior addAt {
    inout seq: Anything[0..*] ordered nonunique; in values: Anything[0..*] ordered nonunique;
    in index: Positive[1];
  }
  behavior remove{
    inout seq: Anything[0..*] ordered nonunique; in values: Anything[0..*];
  }
  behavior removeAt{
    inout seq: Anything[0..*] ordered nonunique;
    in startIndex: Positive[1]; in endIndex: Positive[1] default startIndex;
  }
}

```

```

standard library package CollectionFunctions {
  ...
  public import Collections::*;

  function '==' specializes BaseFunctions::'=='
    {in col1: Collection[0..1]; in col2: Collection[0..1]; return : Boolean[1];}

  function size {in col: Collection[1]; return : Natural[1];}
  function isEmpty {in col: Collection[1]; return : Boolean[1];}
  function notEmpty {in col: Collection[1]; return : Boolean[1];}
  function contains {in col: Collection[1]; in values: Anything[*]; return : Boolean[1];}
  function containsAll {in col1: Collection[1]; in col2: Collection[2]; return : Boolean[1];}
  function head {in col: OrderedCollection[1]; return : Anything[0..1];}
  function tail {in col: OrderedCollection[1]; return : Anything[0..*] ordered nonunique;}
  function last {in col: OrderedCollection[1]; return : Anything[0..1];}

  function '#' specializes BaseFunctions::'#'
    {in col: OrderedCollection[1]; in index: Positive[1]; return : Anything[0..1];}
  function 'array#' specializes BaseFunctions::'#'
    {in arr: Array[1]; in indexes: Positive[n] ordered nonunique; return : Anything[0..1];}
}

```

Vector Functions (1)

```

standard library package VectorFunctions {
  ...

  public import VectorValues::*;

  abstract function isZeroVector {in v: VectorValue[1]; return : Boolean[1];}
  abstract function '+' specializes DataFunctions::'+'
    {in v: VectorValue[1]; in w: VectorValue[0..1]; return u: VectorValue[1];}
  abstract function '-' specializes DataFunctions::-'-'
    {in v: VectorValue[1]; in w: VectorValue[0..1]; return u: VectorValue[1];}

  function VectorOf {in components: NumericalValue[1..*] ordered nonunique;
    return : NumericalVectorValue[1];}

  abstract function scalarVectorMult specializes DataFunctions::'*'
    {in x: NumericalValue[1]; in v: NumericalVectorValue[1]; return w: NumericalVectorValue[1];}
  alias '*' for scalarVectorMult;
  abstract function vectorScalarMult specializes DataFunctions::'*'
    {in v: NumericalVectorValue[1]; in x: NumericalValue[1]; return : NumericalVectorValue[1];}
  abstract function vectorScalarDiv specializes DataFunctions::'/'
    {in v: NumericalVectorValue[1]; in x: NumericalValue[1]; return : NumericalVectorValue[1];}
  abstract function inner specializes DataFunctions::'*'
    {in v: NumericalVectorValue[1]; in w: NumericalVectorValue[1]; return x: NumericalValue[1];}
  abstract function norm {in v: NumericalVectorValue[1]; return l : NumericalValue[1];}
  abstract function angle {in v: NumericalVectorValue[1]; in w: NumericalVectorValue[1];
    return theta: NumericalValue[1];}

  ...
}

```


Vector Functions (2)

```

standard library package VectorFunctions {
  ...

  function CartesianVectorOf
    {in components: NumericalValue[1..*] ordered nonunique; return : CartesianVectorValue[1];}
  function CartesianThreeVectorOf specializes CartesianVectorOf
    {in components: Real[3] ordered nonunique; return : CartesianVectorValue[1];}

  function isCartesianZeroVector specializes isZeroVector
    {in v: CartesianVectorValue[1]; return : Boolean[1];}

  function 'cartesian+' specializes '+'
    {in v: CartesianVectorValue[1]; in w: CartesianVectorValue[0..1]; return: CartesianVectorValue[1];}
  function 'cartesian-' specializes '-'
    {in v: CartesianVectorValue[1]; in w: CartesianVectorValue[0..1]; return: CartesianVectorValue[1];}
  function cartesianScalarVectorMult specializes scalarVectorMult
    {in x: Real[1]; in v: CartesianVectorValue[1]; return : CartesianVectorValue[1];}
  function cartesianVectorScalarMult specializes vectorScalarMult
    {in v: CartesianVectorValue[1]; in x: Real[1]; return : CartesianVectorValue[1];}
  function cartesianInner specializes inner
    {in v: CartesianVectorValue[1]; in w: CartesianVectorValue[0..1]; return : Real[1];}
  function cartesianNorm specializes norm
    {in v: CartesianVectorValue[1]; return : NumericalValue[1];}
  function cartesianAngle specializes angle
    {in v: CartesianVectorValue[1]; in w: CartesianVectorValue[0..1]; return : Real[1];}

  function sum {in coll: CartesianThreeVectorValue[*]; return : CartesianThreeVectorValue[0..1];}
}

```

```

standard library package ControlFunctions {
  private import Base::Anything;
  private import ScalarValues::ScalarValue;

  abstract function 'if' {in test: Boolean[1];
    in expr thenValue[0..1] {return : Anything[0..*] ordered nonunique;}
    in expr elseValue[0..1] {return : Anything[0..*] ordered nonunique;}
    return : Anything[0..*] ordered nonunique;}
  abstract function '??' {in firstValue: Anything[0..*];
    in expr secondValue[0..1] {return : Anything[0..*] ordered nonunique;}
    return : Anything[0..*] ordered nonunique;}

  function 'and'
    {in firstValue: Anything[0..*]; in expr secondValue[0..1] {return : Boolean[1];} return : Boolean[1];}
  function 'or'
    {in firstValue: Anything[0..*]; in expr secondValue[0..1] {return : Boolean[1];} return : Boolean[1];}
  function 'implies'
    {in firstValue: Anything[0..*]; in expr secondValue[0..1] {return : Boolean[1];} return : Boolean[1];}

  abstract function collect {in collection: Anything[0..*] ordered nonunique;
    in expr mapper[0..*] (argument: Anything[1]): Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique {
  }
  abstract function select { in collection: Anything[0..*] ordered nonunique;
    in expr selector[0..*] (argument: Anything[1]): Boolean[1];
    return : Anything[0..*] ordered nonunique {
  }
  abstract function reject { in collection: Anything[0..*] ordered nonunique;
    in expr rejector[0..*] (argument: Anything[1]): Boolean[1];
    return : Anything[0..*] ordered nonunique {
  }
  ...
}

```

Occurrence Functions

```
standard library package OccurrenceFunctions {
```

```
...
function '===' specializes BaseFunctions::'===' {
  in x: Occurrence[0..1]; in y: Occurrence[0..1];
  return : Boolean[1];
}
```

Test whether two occurrences represent different portions of the same entity (i.e., whether they have the same "identity"). Can be invoked using the `===` operator.

```
function isDuring {
  in occ: Occurrence[1];
  return : Boolean[1];
}
```

Test whether a performance of this function happens during the input occurrence.

```
function create {
  inout occ: Occurrence[1];
  return : Occurrence[1];
}
```

Create or destroy an occurrence.

```
function destroy {
  inout occ: Occurrence[0..1];
  return : Occurrence[0..1];
}
```

Add a newly created instance to a group of occurrences.

```
function addNew {
  inout group: Occurrence[0..*] nonunique; inout occ: Occurrence[1];
  return : Occurrence[1];
}
```

```
function addNewAt {
  inout group: Occurrence[0..*] ordered nonunique; inout occ: Occurrence[1]; in index: Positive[1];
  return : Occurrence[1];
}
```

Remove an instance from a group of occurrences and destroy it.

```
behavior removeOld {
  inout group: Occurrence[0..*] nonunique; inout occ: Occurrence[0..1];
}
```

```
behavior removeOldAt {
  inout group: Occurrence[0..*] ordered nonunique; in index: Positive[1];
}
```

```
}
```

Systems Library (selected models)

Standard View Definitions

```
standard library package StandardViewDefinitions {  
  
    view def <gv> GeneralView;  
    view def <iv> InterconnectionView;  
    view def <afv> ActionFlowView specializes InterconnectionView;  
    view def <stv> StateTransitionView specializes InterconnectionView;  
    view def <sv> SequenceView;  
    view def <gev> GeometryView;  
    view def <gv> GridView;  
    view def <bv> BrowserView;  
}
```

Metadata Domain Library

```
standard library package RiskMetadata {
  private import ScalarValues::Real;
  attribute def Level :> Real {
    assert constraint { that >= 0.0 and that <= 1.0 }
  }
  enum def LevelEnum :> Level {
    low = 0.25;
    medium = 0.50;
    high = 0.75;
  }
  attribute def RiskLevel {
    attribute probability : Level;
    attribute impact : Level [0..1];
  }
  enum def RiskLevelEnum :> RiskLevel {
    low = RiskLevel(probability = LevelEnum::L);
    medium = RiskLevel(probability = LevelEnum::M);
    high = RiskLevel(probability = LevelEnum::H);
  }
  metadata def Risk {
    attribute totalRisk : RiskLevel [0..1];
    attribute technicalRisk : RiskLevel [0..1];
    attribute scheduleRisk : RiskLevel [0..1];
    attribute costRisk : RiskLevel [0..1];
  }
}
```

General risk level in terms of probability and impact.

Standard low, medium and high risk levels.

To be used to annotate model elements as to their risk level in typical risk areas.

```

standard library package ModelingMetadata {
  ...
  enum def StatusKind {
    open;
    tbd; // To be determined
    tbr; // To be resolved
    tbc; // To be confirmed
    done;
    closed;
  }
  metadata def StatusInfo {
    attribute originator : String [0..1];
    attribute owner : String [0..1];
    attribute status : StatusKind;
    attribute risk : Risk [0..1];
  }
  metadata def Rationale {
    attribute text : String;
    ref explanation : Anything [0..1];
  }
  metadata def Issue {
    attribute text : String;
  }
  metadata def <refinement> Refinement {
    :>> annotatedElement : SysML::Dependency;
  }
}

```

To be used to annotate model elements with status information.

To be used to give a rationale for a model element.

Generic issue annotation.

To be used to model a dependency in which source elements refine target elements.

This is an optional reference to a feature that further explains this rationale (e.g., a trade study analysis).

Semantic metadata for identifying an attribute as a measure of effectiveness.

Semantic metadata for identifying an attribute as a measure of performance.

```
standard library package ParametersOfInterestMetadata {
  private import Metaobjects::SemanticMetadata;

  attribute measuresOfEffectiveness[*] nonunique;
  attribute measuresOfPerformance[*] nonunique;

  metadata def <moe> MeasureOfEffectiveness :> SemanticMetadata {
    :>> annotatedElement : SysML::Usage;
    :>> baseType = measuresOfEffectiveness meta SysML::Usage;
  }

  metadata def <mop> MeasureOfPerformance :> SemanticMetadata {
    :>> annotatedElement : SysML::Usage;
    :>> baseType = measuresOfPerformance meta SysML::Usage;
  }
}
```

Provides data necessary for the physical definition of a graphical image.

To be used to annotate a model element with an image.

Full-sized image for rendering the annotated element.

Smaller image for use as an adornment or marker.

```
standard library package ImageMetadata {
  private import ScalarValues::String;

  attribute def Image {
    attribute content : String[0..1];

    attribute encoding : String[0..1];

    attribute type : String[0..1];

    attribute location : String[0..1];
  }

  metadata def Icon {
    attribute fullImage : Image[0..1];

    attribute smallImage : Image[0..1];
  }
}
```

Binary data for the image.

Binary data character encoding (e.g., "base64").

MIME type of the content.

URI for image content, alternative to embedding it in the `content` attribute.

Analysis Domain Library

Trade Studies

```

standard library package TradeStudies {
  private import Base::Anything;
  private import ScalarValues::*;
  private import ScalarFunctions::*;
  abstract calc def EvaluationFunction(alternative) result : ScalarValue[1];
  abstract requirement def TradeStudyObjective {
    subject selectedAlternative : Anything;
    in ref alternatives : Anything[1..*];
    in calc eval : EvaluationFunction;
    out attribute best : ScalarValue;
    require constraint { eval(selectedAlternative) == best }
  }
  requirement def MinimizeObjective :> TradeStudyObjective {
    ...
    attribute :>> best = alternatives->minimize {in x; eval(x)};
  }
  requirement def MaximizeObjective :> TradeStudyObjective {
    ...
    attribute :>> best = alternatives->maximize {in x; eval(x)};
  }
  abstract analysis def TradeStudy {
    subject studyAlternatives : Anything[1..*];
    abstract calc evaluationFunction : EvaluationFunction;
    objective tradeStudyObjective : TradeStudyObjective {
      subject :>> selectedAlternative;
      in ref :>> alternatives = studyAlternatives;
      in calc :>> eval = evaluationFunction;
    }
    return selectedAlternative : Anything =
      studyAlternatives->selectOne {in ref a; tradeStudyObjective(selectedAlternative = a)};
  }
}

```

An **EvaluationFunction** is a calculation that evaluates a **TradeStudy** alternative.

A **TradeStudyObjective** is the base definition for the objective of a **TradeStudy**, requiring the **selectedAlternative** to have best evaluation according to a given **EvaluationFunction**.

A **TradeStudy** is an analysis case whose subject is a set of alternatives and whose result is a selection of one of those alternatives, based on a given **EvaluationFunction** such that it satisfies the objective of the **TradeStudy**.

```
standard library package AnalysisTooling {
  private import ScalarValues::*;

  metadata def ToolExecution {
    attribute toolName : String;
    attribute uri : String;
  }

  metadata def ToolVariable {
    attribute name : String;
  }
}
```

ToolExecution metadata identifies an external analysis tool to be used to implement the annotated action.

ToolVariable metadata is used in the context of an action that has been annotated with **ToolExecution** metadata. It is used to annotate a parameter or other feature of the action with the name of the variable in the tool that is to correspond to the annotated feature.

State Space Representation (1)

```

standard library package StateSpaceRepresentation {
  private import ISQ::DurationValue;
  private import Quantities::VectorQuantityValue;
  private import VectorCalculations::*;

  abstract attribute def StateSpace :> VectorQuantityValue;
  abstract attribute def Input :> VectorQuantityValue;
  abstract attribute def Output :> VectorQuantityValue;
  abstract calc def GetNextState
    {in input: Input; in stateSpace: StateSpace; in timeStep: DurationValue; return : StateSpace;}
  abstract calc def GetOutput
    {in input: Input; in stateSpace: StateSpace; return : Output;}
  abstract action def StateSpaceEventDef;
  action def ZeroCrossingEventDef :> StateSpaceEventDef;
  item def StateSpaceItem;

  abstract action def StateSpaceDynamics {
    in attribute input: Input;
    abstract calc getNextState: GetNextState;
    abstract calc getOutput: GetOutput;
    attribute stateSpace: StateSpace;
    out attribute output: Output = getOutput(input, stateSpace);
  }
  abstract attribute def StateDerivative :> VectorQuantityValue { ... }
  abstract calc def GetDerivative
    {in input: Input; in stateSpace: StateSpace; return : StateDerivative;}
  abstract calc def Integrate {in getDerivative: GetDerivative; in input: Input;
    in initialState: StateSpace; in timeInterval: DurationValue; return result: StateSpace;}
  ...

```

`StateSpaceDynamics` is the simplest form of state space representation, and `getNextState` directly computes the `stateSpace` of the next timestep.

State Space Representation (2)

ContinuousStateSpaceDynamics represents continuous behavior. The **derivative** needs to return a time derivative of **stateSpace**, i.e. dx/dt .

```
...
abstract action def ContinuousStateSpaceDynamics :> StateSpaceDynamics {
  abstract calc getDerivative: GetDerivative;
  calc :>> getNextState: GetNextState {
    calc integrate: Integrate {
      in derivative = ContinuousStateSpaceDynamics::getDerivative;
      in input = GetNextState::input;
      in initialState = GetNextState::stateSpace;
      in timeInterval = GetNextState::timeStep;
      return resultState = result;
    }
  }
  event occurrence zeroCrossingEvents[0..*] : ZeroCrossingEventDef;
}

abstract calc def GetDifference {
  in input: Input; in stateSpace: StateSpace;
  return : StateSpace;}

abstract action def DiscreteStateSpaceDynamics :> StateSpaceDynamics {
  abstract calc getDifference: GetDifference;
  calc :>> getNextState: GetNextState {
    attribute diff: StateSpace = getDifference(input, stateSpace);
    stateSpace + diff
  }
}
```

DiscreteStateSpaceDynamics represents discrete behavior. **getDifference** returns the difference of the **stateSpace** for each timestep.

Sampled Functions

SampleFunction is a variable-size, ordered collection of **SamplePair** elements representing a discretely sampled mathematical function.

Domain and **Range** return all the domain values and range values of a sampled function.

Sample a calculation on given domain values to create a **SampledFunction**.

Interpolate a **SampledFunction** to compute a result for a given domain value.

SamplePair is a key-value pair of a domain-value and a range-value, used as a sample element in **SampledFunction**.

The function is constrained to be strictly increasing or strictly decreasing.

```
standard library package SampledFunction
...
attribute def SamplePair :> KeyValuePair {
  attribute domainValue :>> key;
  attribute rangeValue :>> val;
}

attribute def SampledFunction :> OrderedMap {
  attribute samples: SamplePair[0..*] ordered :>> elements;
  assert constraint { ... }
}

calc def Domain{in fn : SampledFunction;
  return : Anything[0..*] = fn.samples.domainValue;}
calc def Range{in fn : SampledFunction;
  return : Anything[0..*] = fn.samples.rangeValue;}

calc def Sample {in calc calculation {in x;}
  in attribute domainValues [0..*];
  return sampling;}

calc def Interpolate {in attribute fn : SampledFunction;
  in attribute value;
  return attribute result;}
calc interpolateLinear : Interpolate { ... }
}
```


Cause and Effect Domain Library

A connection between one or more *cause occurrences* and one or more *effect occurrences*. Specializations add ends for specific causes and effects.

A binary connection between a single cause and a single effect. (But a single cause can separately have multiple effects, and a single effect can separately have multiple causes.)

```
standard library package CausationConnections {
  abstract occurrence causes[*];
  abstract occurrence effects[*];

  abstract connection def Multicausation {
    abstract constant ref occurrence causes[1..*]
      :>> causes :> participant;
    abstract constant ref occurrence effects[1..*]
      :>> effects :> participant;

    ...
  }

  abstract connection multicausations : Multicausation[*] {

    connection def Causation :> Multicausation {
      end theCauses[*] occurrence theCause
        :>> causes :>> source;
      end theEffects[*] occurrence theEffect
        :>> effects :>> target;
    }

    abstract connection causations : Causation[*]
      :> multicausations;
  }
}
```

Cause and Effect Metadata

Used to tag the “cause” ends of a multicausation.

Used to tag the “effect” ends of a multicausation.

Additional metadata about a causation connection.

Semantic metadata for multicausation.

Semantic metadata for binary causation.

```
standard library package CauseAndEffect {
  public import CausationConnections::*;
  private import ScalarValues::*;
  private import Metaobjects::SemanticMetadata;
  metadata def <cause> CauseMetadata :> SemanticMetadata {
    ref :>> annotatedElement : SysML::Usage;
    ref :>> baseType = causes meta SysML::Usage;
  }
  metadata def <effect> EffectMetadata :> SemanticMetadata {
    ref :>> annotatedElement : SysML::Usage;
    ref :>> baseType = effects meta SysML::Usage;
  }
  metadata def CausationMetadata {
    ref :> annotatedElement : SysML::ConnectionDefinition;
    ref :> annotatedElement : SysML::ConnectionUsage;
    attribute isNecessary : Boolean default false;
    attribute isSufficient : Boolean default false;
    attribute probability : Real[0..1];
  }
  metadata def <multicausation> MulticausationSemanticMetadata
    :> CausationMetadata, SemanticMetadata {
    ref :>> baseType = multicausations meta SysML::Usage;
  }
  metadata def <causation> CausationSemanticMetadata
    :> CausationMetadata, SemanticMetadata {
    ref :>> baseType = causations meta SysML::Usage;
  }
}
```

Requirements Derivation Domain Library

Derivation Connections

A connection between one *original requirement* and one or more *derived requirements*. Specializations add ends for specific original and derived requirements.

Original requirement must not be a derived requirement.

Whenever the original requirement is satisfied, all the derived requirements must also be satisfied.

```
standard library package DerivationConnections {
  requirement originalRequirements[*];
  requirement derivedRequirements[*];

  abstract connection def Derivation {
    ref requirement originalRequirement[1]
      :>> originalRequirements :> participant;
    ref requirement :>> derivedRequirements[1..*]
      :> participant;

    private assert constraint originalNotDerived { ... }
    private assert constraint originalImpliesDerived { ... }
  }

  abstract connection derivations : Derivation[*];
}
```

Used to tag the “original” ends of a derivation.

Used to tag the “derived” ends of a derivation.

Semantic metadata for requirement derivation.

```
standard library package RequirementDerivation {
  public import DerivationConnections::*;
  private import Metaobjects::SemanticMetadata;

  metadata def <original> OriginalRequirementMetadata
    :> SemanticMetadata {
    :> annotatedElement : SysML::Usage;
    :>> baseType = originalRequirements meta SysML::Usage;
  }

  metadata def <derive> DerivedRequirementMetadata
    :> SemanticMetadata {
    :> annotatedElement : SysML::Usage;
    :>> baseType = derivedRequirements meta SysML::Usage;
  }

  metadata def <derivation> DerivationMetadata
    :> SemanticMetadata {
    :> annotatedElement : SysML::ConnectionDefinition;
    :> annotatedElement : SysML::ConnectionUsage;
    :>> baseType = derivations meta SysML::Usage;
  }
}
```

Quantities and Units Domain Library (selected models)

Quantities

standard library package Quantities

A *tensor quantity value* represents the value of the most general kind of quantity, a *tensor*.

The numeric value of the tensor is given as an array of numbers.

```
abstract attribute def TensorQuantityValue :> Collections::Array {
  attribute num : ScalarValues::Number[1..*] ordered nonunique :>> elements;
  attribute mRef : UnitsAndScales::TensorMeasurementReference;
  attribute :>> dimensions = mRef::dimensions;
  ...
}
```

The value of a tensor quantity is relative to a multi-dimensional *tensor measurement reference*.

```
abstract attribute def VectorQuantityValue :> TensorQuantityValue {
  attribute :>> mRef : UnitsAndScales::VectorMeasurementReference;
}
```

A *vector* is a tensor with a single dimension.

```
abstract attribute def ScalarQuantityValue :> ScalarQuantityValue {
  attribute :>> mRef : UnitsAndScales::ScalarMeasurementReference;
}
```

A *scalar* is a vector with a single element. Its measurement reference may be a *measurement unit* if the scale is a *ratio scale*.

```
abstract attribute tensorQuantities: TensorQuantityValue[*]
abstract attribute vectorQuantities: VectorQuantityValue[*]
abstract attribute scalarQuantities: ScalarQuantityValue[*]
```

```
alias TensorQuantityValue as QuantityValue;
alias tensorQuantities as quantity;
```

A *quantity* is a usage of a quantity value to represent a feature of something.


```

standard library package MeasurementReferences {
  attribute def TensorMeasurementReference :> Array {
    attribute mRefs : ScalarMeasurementReference[1..*] :>> elements;
    ...
  }

  attribute def VectorMeasurementReference
    :> TensorMeasurementReference {
    attribute :>> dimensions: Positive[0..1];
    ...
  }

  attribute def ScalarMeasurementReference
    :> VectorMeasurementReference {
    attribute :>> dimensions = ();
    ...
  }

  attribute def CoordinateFrame :> VectorMeasurementReference {
    attribute transformation: CoordinateTransformation[0..1];
  }
  ...
}

```

A *tensor measurement reference* is defined as an array of *scalar measurement references*, one for each element of the tensor.

A *coordinate frame* is a kind of vector measurement reference that can be located and oriented relative to another frame using a *coordinate transformation*.

A *measurement unit* is a measurement scale defined as a sequence of unit power factors.

A *simple unit* is a measurement unit with no power factor dependencies on other units.

A *derived unit* is any unit that is not simple.

A *unit power factor* is a representation of a reference unit raised to an exponent.

```
standard library package Measurement {
  ...

  abstract attribute def MeasurementUnit
    :> ScalarMeasurementReference {
    attribute unitPowerFactor : UnitPowerFactor[1..*] ordered;
    attribute unitConversion : UnitConversion[0..1];
  }

  abstract attribute def SimpleUnit :> MeasurementUnit {
    attribute redefines unitPowerFactor[1] {
      attribute redefines unit = SimpleUnit::self;
      attribute redefines exponent = 1;
    }
  }

  abstract attribute def DerivedUnit :> MeasurementUnit;

  attribute def UnitPowerFactor {
    attribute unit : MeasurementUnit;
    attribute exponent : ScalarValues::Real;
  }
  ...
}
```

The *International System of Quantities* defines seven abstract units (length, mass, time, electric current, temperature, amount of substance, luminous intensity) and many other units derived from those.

```
standard library package ISQ
  import ISQBase::*;
  import ISQSpaceTime::*;
  ...
}
```

A *length unit* is a simple unit.

A *length value* is a quantity value with a real magnitude and a length-unit scale.

A *length* is a quantity with a length value.

```
standard library package ISQBase {

  attribute def LengthUnit :> SimpleUnit {...}

  attribute def LengthValue :> ScalarQuantityValue {
    attribute redefines num : ScalarValues::Real;
    attribute redefines mRef : LengthUnit;
  }

  attribute length: LengthValue :> quantity;
  ...
}
```

The ISQ standard (ISO 80000) is divided into several parts. For example, Part 3 defines units related to space and time.

```
package ISQSpaceTime {
  import ISQBase::*;
  ...
}
```

International System of Units / Système International (SI)

The *International System of Units* defines base units for the seven abstract ISQ unit types.

Each unit declaration includes the full unit name and its abbreviation as an identifier.

```
standard library package SI {
  public import ISQ::*;
  public import SIPrefixes::*;

  attribute <g> gram : MassUnit;

  attribute <m> metre : LengthUnit;
  attribute <kg> kilogram : MassUnit {
    attribute redefines unitConversion : ConversionByPrefix {
      attribute redefines prefix = kilo;
      attribute redefines referenceUnit = g
    }
  }

  attribute <s> second : TimeUnit;
  attribute <A> ampere : ElectricCurrentUnit;
  attribute <K> kelvin : ThermodynamicTemperatureUnit;
  attribute <mol> mole : AmountOfSubstanceUnit;
  attribute <cd> candela : LuminousIntensityUnit;
  ...

  attribute <N> newton : ForceUnit = kg * m / s ^ 2;
```

A unit can be defined using prefix-based conversion from a reference unit.

A derived unit can be defined from an arithmetic expression of other units.

US Customary Units

```

standard library package USCustomaryUnits {
  public import ISQ::*;
  private import SI::*;

  attribute <ft> foot : LengthUnit {
    attribute redefines unitConversion : ConversionByConvention {
      attribute redefines referenceUnit = m,
      attribute redefines conversionFactor = 3048/10000;
    }
  }
  ...
  attribute <mi> mile : LengthUnit {
    attribute redefines unitConversion : ConversionByConvention {
      attribute redefines referenceUnit = ft,
      attribute redefines conversionFactor = 5280;
    }
  }

  attribute <'mi/hr'> 'mile per hour' : SpeedUnit = mi / hr;
  alias mph for 'mi/hr';
  ...
}

```

US customary units are defined by conversion from SI units.

An alias for mile per hour.

Quantity Calculations (1)

```

standard library package QuantityCalculations {
  private import ScalarValues::*;
  private import Quantities::ScalarQuantityValue;
  private import MeasurementReferences::ScalarMeasurementReference;
  private import MeasurementReferences::DimensionOneValue;

  calc def '[' specializes BaseFunctions::'[' { in num: Number[1]; in mRef: ScalarMeasurementReference[1];
    return quantity : ScalarQuantityValue[1]; }

  ...
  calc def abs specializes NumericalFunctions::abs
    { in x: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }
  calc def '+' specializes NumericalFunctions::'+'
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[0..1]; return : ScalarQuantityValue; }
  calc def '-' specializes NumericalFunctions::'-'
    { in x: ScalarQuantityValue; in y: ScalarQuantityValue[0..1]; return : ScalarQuantityValue[1]; }
  calc def '*' specializes NumericalFunctions::'*'
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }
  calc def '/' specializes NumericalFunctions::'/'
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }
  calc def '**' specializes NumericalFunctions::'**'
    { in x: ScalarQuantityValue[1]; in y: Real[1]; return : ScalarQuantityValue[1]; }
  calc def '^' specializes NumericalFunctions::'^'
    { in x: ScalarQuantityValue[1]; in y: Real[1]; return : ScalarQuantityValue[1]; }
  calc def '<' specializes NumericalFunctions::'<'
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : Boolean[1]; }
  calc def '>' specializes NumericalFunctions::'>'
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : Boolean[1]; }
  calc def '<=' specializes NumericalFunctions::'<='
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : Boolean[1]; }
  calc def '>=' specializes NumericalFunctions::'>='
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : Boolean[1]; }

  ...
}

```

Quantity Calculations (2)

```

standard library package QuantityCalculations {
  ...
  calc def max specializes NumericalFunctions::max
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }
  calc def min specializes NumericalFunctions::min
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }

  calc def '==' specializes DataFunctions::'=='
    { in x: ScalarQuantityValue[1]; in y: ScalarQuantityValue[1]; return : Boolean[1]; }
  calc def sqrt { in x: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }

  calc def floor { in x: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }
  calc def round { in x: ScalarQuantityValue[1]; return : ScalarQuantityValue[1]; }
  calc def ToString specializes BaseFunctions::ToString { in x: ScalarQuantityValue[1]; return : String; }
  calc def ToInteger { in x: ScalarQuantityValue[1]; return : Integer[1]; }
  calc def ToRational { in x: ScalarQuantityValue[1]; return : Rational[1]; }
  calc def ToReal { in x: ScalarQuantityValue[1]; return : Real[1]; }
  calc def ToDimensionOneValue { in x: Real[1]; return : DimensionOneValue[1]; }
  calc def sum specializes NumericalFunctions::sum
    { in collection: ScalarQuantityValue[0..*]; return : ScalarQuantityValue; }
  calc def product specializes NumericalFunctions::product
    { in collection: ScalarQuantityValue[0..*]; return : ScalarQuantityValue; }

  calc def ConvertQuantity { in x: ScalarQuantityValue[1]; in targetMRef: ScalarMeasurementReference[1];
    return : ScalarQuantityValue[1]; }
}

```

```

standard library package MeasurementRefCalculations {
  private import ScalarValues::String;
  private import ScalarValues::Real;
  private import MeasurementReferences::MeasurementUnit;
  private import MeasurementReferences::ScalarMeasurementReference;
  private import MeasurementReferences::CoordinateFrame;

  calc def '*' specializes DataFunctions::'*'
    { in x: MeasurementUnit[1]; in y: MeasurementUnit[1]; return : MeasurementUnit[1]; }
  calc def '/' specializes DataFunctions::'/'
    { in x: MeasurementUnit[1]; in y: MeasurementUnit[1]; return : MeasurementUnit[1]; }
  calc def '**' specializes DataFunctions::'**'
    { in x: MeasurementUnit[1]; in y: Real[1]; return : MeasurementUnit[1]; }
  calc def '^' specializes DataFunctions::'^'
    { in x: MeasurementUnit[1]; in y: Real[1]; return : MeasurementUnit[1]; }

  calc def 'CoordinateFrame*' specializes DataFunctions::'*'
    { in x: CoordinateFrame[1]; in y: MeasurementUnit[1]; return : CoordinateFrame[1]; }
  calc def 'CoordinateFrame/' specializes DataFunctions::'/'
    { in x: CoordinateFrame[1]; in y: MeasurementUnit[1]; return : CoordinateFrame[1]; }

  calc def ToString specializes BaseFunctions::ToString
    { in x: ScalarMeasurementReference[1]; return : String[1]; }
}

```


Vector Calculations

```

standard library package VectorCalculations {
  private import ScalarValues::Boolean;
  private import ScalarValues::Number;
  private import Quantities::VectorQuantityValue;

  calc def '+' specializes VectorFunctions::'+'
    {in : VectorQuantityValue; in : VectorQuantityValue; return : VectorQuantityValue;}
  calc def '-' specializes VectorFunctions::'-'
    {in : VectorQuantityValue; in : VectorQuantityValue; return : VectorQuantityValue;}

  calc def scalarVectorMult specializes VectorFunctions:: scalarVectorMult
    {in : Number; in : VectorQuantityValue; return : VectorQuantityValue;}
  calc def vectorScalarMult specializes VectorFunctions:: vectorScalarMult
    {in : VectorQuantityValue; in : Number; return : VectorQuantityValue;}
  calc def vectorScalarDiv specializes RealFunctions:: '*'
    {in : VectorQuantityValue; in : Number; return : VectorQuantityValue;}
  calc def inner specializes VectorFunctions::inner
    {in : VectorQuantityValue; in : VectorQuantityValue; return : Number;}
  alias scalarVectorMult as '*';

  calc def norm specializes VectorFunctions::norm
    {in : VectorQuantityValue; return : Number;}
  calc def angle specializes VectorFunctions::angle
    {in : VectorQuantityValue; in : VectorQuantityValue; return : Number;}
}

```

```

standard library package TensorCalculations {
  private import ScalarValues::Boolean;
  private import ScalarValues::Number;
  private import Quantities::ScalarQuantityValue;
  private import Quantities::VectorQuantityValue;
  private import Quantities::TensorQuantityValue;
  private import MeasurementReferences::TensorMeasurementReference;
  private import MeasurementReferences::CoordinateTransformation;

  calc def '[' specializes BaseFunctions:: '[' { in elements: Number[1..n] ordered;
    in mRef: TensorMeasurementReference[1]; return quantity: TensorQuantityValue[1];
    private attribute n = mRef.flattenedSize;
  }
  ...
  calc def '+' :> DataFunctions:: '+'
    [{ in : TensorQuantityValue[1]; in : TensorQuantityValue[1]; return : TensorQuantityValue[1]; }
  calc def '-' :> DataFunctions:: '-'
    { in : TensorQuantityValue[1]; in : TensorQuantityValue[1]; return : TensorQuantityValue[1]; }
  calc def scalarTensorMult { in : Number[1]; in : TensorQuantityValue[1]; return : TensorQuantityValue[1]; }
  calc def tensorScalarMult { in : TensorQuantityValue[1]; in : Number[1]; return : TensorQuantityValue[1]; }
  calc def scalarQuantityTensorMult
    { in : ScalarQuantityValue[1]; in : TensorQuantityValue[1]; return : TensorQuantityValue[1]; }
  calc def tensorScalarQuantityMult
    { in : TensorQuantityValue[1]; in : ScalarQuantityValue[1]; return : TensorQuantityValue[1]; }
  calc def tensorVectorMult
    { in : TensorQuantityValue[1]; in : VectorQuantityValue[1]; return : VectorQuantityValue[1]; }
  calc def vectorTensorMult
    { in : VectorQuantityValue[1]; in : TensorQuantityValue[1]; return : VectorQuantityValue[1]; }
  calc def tensorTensorMult
    { in : TensorQuantityValue[1]; in : TensorQuantityValue[1]; return : TensorQuantityValue[1]; }
  calc def transform { in transformation : CoordinateTransformation; in sourceTensor : TensorQuantityValue;
    return targetTensor : TensorQuantityValue; }
}

```

Clocks is a Kernel Library model of the semantics of clocks.

A clock provides the current time as a quantity that advances monotonically over the lifetime of the clock.

TimeOf returns the time instant of the start of an occurrence relative to a given clock.

DurationOf returns the duration of a given occurrence relative to a given clock (the time of its end snapshot minus the time of its start snapshot).

```
standard library package Time {

  part universalClock : Clock[1] :>
    Clocks::universalClock;

  part def Clock :> Clocks::Clock {
    attribute :>> currentTime : TimeInstantValue;
  }

  calc def TimeOf :> Clocks::TimeOf {
    in o : Occurrence[1];
    in clock : Clock[1] default localClock;
    return timeInstant : TimeInstantValue[1];
  }

  calc def DurationOf :> Clocks::DurationOf {
    in o : Occurrence[1];
    in clock : Clock[1] default localClock;
    return duration : DurationValue;
  }
}
```

A singleton universal time reference.

Time (2)

Generic time scale to express a time instant.

Captures the specification of the time instant with value zero, also known as the (reference) epoch.

```
attribute def TimeScale :> IntervalScale {
  attribute :>> unit: DurationUnit[1];
  attribute definitionalEpoch: DefinitionalQuantityValue[1];
  attribute :>> definitionalQuantityValues = definitionalEpoch;
}
```

Representation of a time instant quantity.

```
attribute def TimeInstantValue :> ScalarQuantityValue {
  attribute :>> num: Real[1];
  attribute :>> mRef: TimeScale[1];
}
```

Generic representation of a time instant as a calendar date and time of day.

```
attribute timeInstant: TimeInstantValue :> scalarQuantities;
```

```
abstract attribute def DateTime :> TimeInstantValue;
abstract attribute def Date :> TimeInstantValue;
abstract attribute def TimeOfDay :> TimeInstantValue;
```

Representation of the Coordinated Universal Time (UTC) time scale.

```
attribute UTC: TimeScale {
  attribute :>> longName = "Coordinated Universal Time";
  attribute :>> unit = SI::s;
  attribute :>> definitionalEpoch: DefinitionalQuantityValue {
    :>> num = 0;
    :>> definition = "UTC epoch at 1 January 1958 at 0 hour 0 minute 0 second";
  }
}
attribute def UtcTimeInstantValue :> DateTime { :>> mRef = UTC; }
attribute utcTimeInstant: UtcTimeInstantValue;
```

Representation of an ISO 8601-1 date and time in extended string format.

Representation of an ISO 8601 date and time with explicit date and time component attributes.

```

attribute def Iso8601DateTimeEncoding :> String;

attribute def Iso8601DateTime :> UtcTimeInstantValue {
  attribute :>> num = getElapsedUtcTime(val);
  attribute val: Iso8601DateTimeEncoding;
  private calc getElapsedUtcTime
    {in iso8601DateTime: Iso8601DateTimeEncoding; return
}
attribute def Iso8601DateTimeStructure :> UtcTimeInstantValue {
  attribute :>> num = getElapsedUtcTime(year, month, day, hour, minute, second,
    microsecond, hourOffset, minuteOffset);
  attribute :>> mRef = UTC;
  attribute year: Integer;
  attribute month: Natural;
  attribute day: Natural;
  attribute hour: Natural;
  attribute minute: Natural;
  attribute second: Natural;
  attribute microsecond: Natural;
  attribute hourOffset: Integer;
  attribute minuteOffset: Integer;
  private calc getElapsedUtcTime(year: Integer, month: Natural, day: Natural,
    hour: Natural, minute: Natural, second: Natural, microsecond: Natural,
    hourOffset: Integer, minuteOffset: Integer) : Real;
}
...
}

```

Geometry Library

Spatial Items (1)

```

standard library package SpatialItems {
...
  item def SpatialItem :> SpatialFrame {
    item :>> self : SpatialItem;
    item :>> localClock : Clock[1] default Time::universalClock;
    attribute coordinateFrame : ThreeDCoordinateFrame[1]
      default universalCartesianSpatial3dCoordinateFrame;

    item originPoint : Point[1] :> spaceShots;

    item subSpatialItems : SpatialItem[1..*] :> subitems {
      ref item :>> SpatialItem::localClock, subitems::localClock;
    }
    item componentItems : SpatialItem[0..*] ordered :> subSpatialItems {
      ref item :>> SpatialItem::localClock, subSpatialItems::localClock;
      attribute :>> coordinateFrame {
        attribute :>> transformation[1] default nullTransformation { ... }
      }
    }
    private attribute cunionNum: Natural [1] = if isEmpty(componentItems) ? 0 else 1;
    private attribute componentUnion[cunionNum] :> unionsOf {
      item :>> elements : SpatialItem[1..*] = componentItems;
    }
    ...
  }
  ...
}

```

A *spatial item* is an Item with a three-dimensional spatial extent that also acts as a spatial frame of reference.

A spatial item has local time and space references used within it.

A *compound* spatial item is a spatial item that is a (spatial) union of a collection of component spatial items.

```
standard library package SpatialItems {
```

```
...
calc def PositionOf :> SpatialFrames::PositionOf {
  in point : Point[1];
  in timeInstant : TimeInstantValue[1];
  in enclosingItem :>> 'frame' : SpatialItem[1];
  in clock : Clock[1] default enclosingItem.localClock;
  return positionVector : VectorQuantityValue[1];
}
calc def CurrentPositionOf :> SpatialFrames::CurrentPositionOf {
  in point : Point[1];
  in enclosingItem :>> 'frame' : SpatialItem[1];
  in clock : Clock[1] default enclosingItem.localClock;
  return positionVector : VectorQuantityValue[1];
}
calc def DisplacementOf :> SpatialFrames::DisplacementOf {
  in point1 : Point[1];
  in point2 : Point[1];
  in timeInstant : TimeInstantValue[1];
  in spacialItem :>> 'frame' : SpatialItem[1];
  in clock : Clock[1] default spacialItem.localClock;
  return displacementVector : VectorQuantityValue[1];
}
calc def CurrentDisplacementOf :> SpatialFrames::CurrentDisplacementOf {
  in point1 : Point[1];
  in point2 : Point[1];
  in spacialItem :>> 'frame' : SpatialItem[1];
  in clock : Clock[1] default spacialItem.localClock;
  return displacementVector : VectorQuantityValue[1];
}
}
```

The position of points in space can be determined relative to the space and time references of a spatial item.

The position of a point can move over time, with its "current" position being relative to the clock reference of the spatial item.

Shape Items (1)

```
standard library package ShapeItems {
  ...
  item def PlanarCurve :> Curve {
    attribute :>> length [1];
    ...
  }

  item def Line :> PlanarCurve {
    attribute :>> length;
    attribute :>> outerSpaceDimension = 1;
  }

  item def PlanarSurface :> Surface {
    attribute :>> area;
    attribute :>> outerSpaceDimension = 2;
    item :>> shape : PlanarCurve;
  }

  abstract item def Path :> StructuredCurve;
  abstract item def Shell :> StructuredSurface;

  ...
}
```

The ShapeItems package provides a model of items that represent basic geometric shapes.

Shape Items (2)

```

standard library package ShapeItems {
  ...
  item def ConicSection :> Path, PlanarCurve {
    ...
  }

  item def Ellipse :> ConicSection {
    attribute :>> semiMajorAxis;
    attribute :>> semiMinorAxis;
    ...
  }

  item def Circle :> Ellipse {
    attribute :>> radius;
    attribute :>> semiMajorAxis = radius;
    attribute :>> semiMinorAxis = radius;
    ...
  }
  ...

  item def Polygon :> Path, PlanarCurve {
    item :>> edges : Line { item :>> vertices [2]; }
    attribute :>> isClosed = true;
    ...
  }
  ...
}

```

Shape Items (3)

```

standard library package ShapeItems {
  ...

  item def Disc :> Shell, PlanarSurface {
    attribute :>> semiMajorAxis;
    attribute :>> semiMinorAxis;
    item :>> shape : Ellipse [1] {
      attribute :>> semiMajorAxis = Disc::semiMajorAxis;
      attribute :>> semiMinorAxis = Disc::semiMinorAxis;
    }
    ...
  }

  item def CircularDisc :> Disc {
    attribute :>> radius [1];
    attribute :>> semiMajorAxis [1] = radius;
    attribute :>> semiMinorAxis [1] = radius;
    item :>> shape : Circle { ... }
    ...
  }

  item def ConicSurface :> Shell {
    ...
  }
  ...
}

```

Shape Items (4)

```

standard library package ShapeItems {
  ...

  item def Ellipsoid :> ConicSurface {
    attribute semiAxis1 : LengthValue [1] :> scalarQuantities;
    attribute semiAxis2 : LengthValue [1] :> scalarQuantities;
    attribute semiAxis3 : LengthValue [1] :> scalarQuantities;
    ...
  }

  item def Sphere :> Ellipsoid {
    attribute :>> radius [1];
    attribute :>> semiAxis1 [1] = radius;
    attribute :>> semiAxis2 [1] = radius;
    attribute :>> semiAxis3 [1] = radius;
  }

  item def Paraboloid :> ConicSurface {
    attribute focalDistance : LengthValue [1] :> scalarQuantities;
    ...
  }

  ...
}

```

Shape Items (5)

```

standard library package ShapeItems {
  ...
  item def ConeOrCylinder :> Shell {
    attribute :>> semiMajorAxis [1];
    attribute :>> semiMinorAxis [1];
    attribute :>> height [1];
    ...
  }

  item def Cone :> ConeOrCylinder { ... }
  item def Cylinder :> ConeOrCylinder { ... }

  ...

  item def Polyhedron :>> Shell { ... }
  item def CuboidOrTriangularPrism :> Polyhedron { ... }
  item def Cuboid :> CuboidOrTriangularPrism { ... }
  item def RectangularCuboid :> Cuboid {
    attribute :>> length [1];
    attribute :>> width [1];
    attribute :>> height [1];
    ...
  }
  alias Box for RectangularCuboid;

  ...
}

```