



SST

Introduction to the SysML v2 Language

Textual Notation

This is a training presentation on the evolving SysML v2 language as it is being developed by the SysML v2 Submission Team (SST).

It is updated as appropriate for each release of the
SysML v2 Pilot Implementation.

Release: 2021-04

*Copyright © 2019-2021 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.*



SST

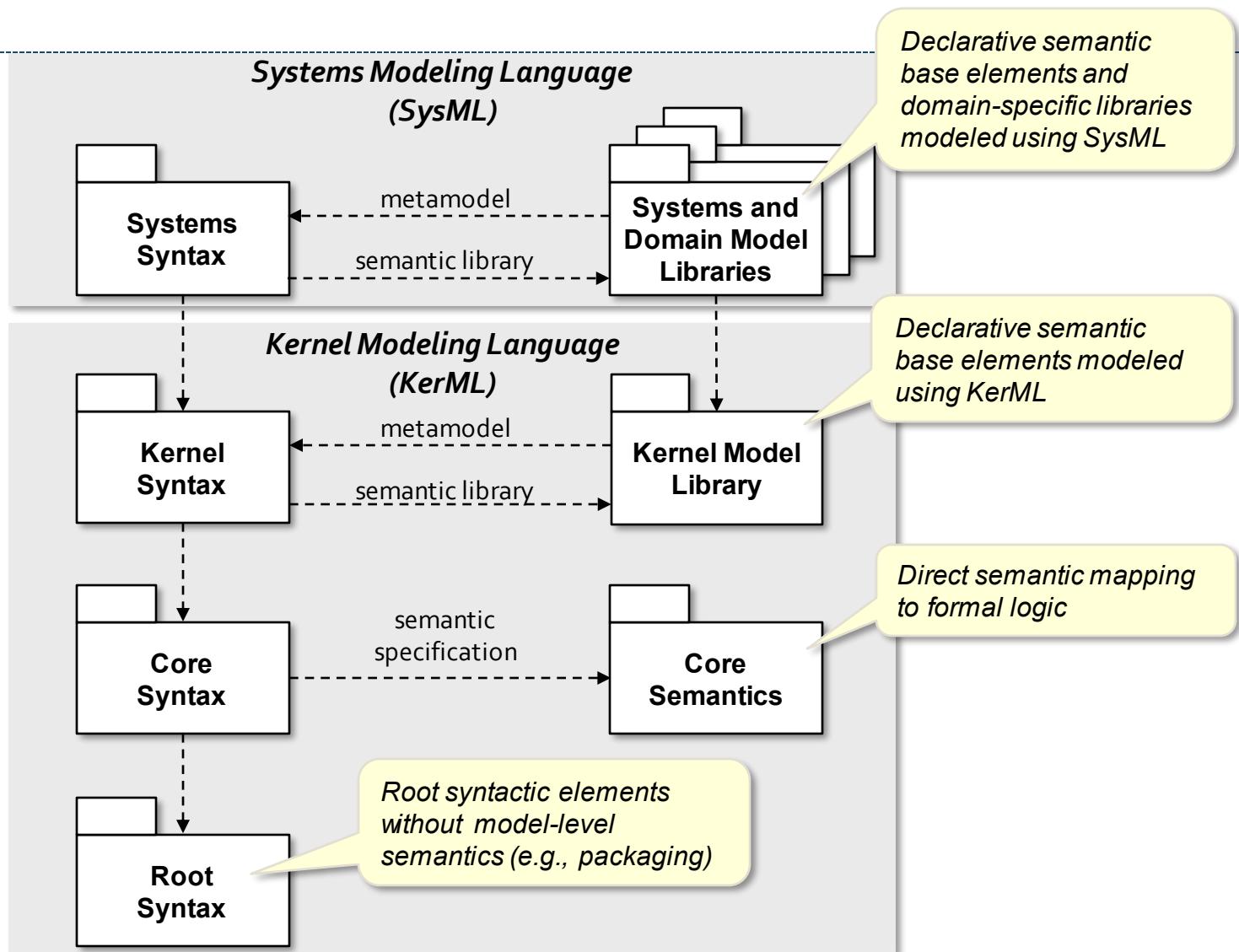
Changes in this Release

- Introduced dot notation for connector end feature paths and as expressions.
- Replaced the { ... } sequence construction notation with (...).
- Changed control functions to use new in-line expression body notation.

Note: Material on views, viewpoints, rendering, stakeholders and concerns will be added with the next release.

*To find slides that have changed recently, search for
Last changed: 2021-04
(and similarly for earlier releases).*

SysML v2 Language Architecture



Four-Layer Language Architecture

- Root – Root syntactic elements
 - Element, AnnotatingElement, Comment, TextualAnnotation, Namespace
 - Relationship, Ownership, Annotation, Documentation, Membership, Import
- Core – Fundamental semantic concepts – Formal declarative semantics
 - Type, Classifier, Feature, Multiplicity
 - FeatureMembership, EndFeatureMembership, Generalization, Superclassing, Subsetting, Redefinition, FeatureTyping, Conjugation, TypeFeaturing
- Kernel – Foundation for building modeling languages – Semantic kernel library
 - Class, DataType, Behavior, Function, Step, Expression, AnnotatingFeature, Package
 - Association, Interaction, Connector, BindingConnector, Succession, ItemFlow, SuccessionItemFlow
- Systems – Modeling language for systems engineering – Domain libraries
 - AttributeDefinition, EnumerationDefinition, ItemDefinition, PartDefinition, IndividualDefinition, PortDefinition, StakeholderDefinition, ActionDefinition, StateDefinition, ConstraintDefinition, RequirementDefinition, ConcernDefinition, CalculationDefinition, CaseDefinition, AnalysisCaseDefinition, VerificationCaseDefinition, ViewDefinition, ViewpointDefinition, RenderingDefinition
 - ReferenceUsage, AttributeUsage, EnumerationUsage, ItemUsage, PartUsage, IndividualUsage, PortUsage, StakeholderUsage, ActionUsage, StateUsage, ConstraintUsage, RequirementUsage, ConcernUsage, CalculationUsage, CaseUsage, AnalysisCaseUsage, VerificationCaseUsage, ViewUsage, ViewpointUsage, RenderingUsage
 - ConnectionDefinition, ConnectionUsage, InterfaceUsage, InterfaceDefinition, AllocationDefinition, AllocationUsage, Dependency

Packages – Members

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

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

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

```
package 'Package Example' {  
    import ScalarValues::*;

    part def Automobile;

    alias Automobile as Car;

    import ISO::TorqueValue as Torque;
}
```

An *import* adds all the members of the *imported* package to the *importing* package.

A package can introduce *aliases* for owned members or individual members of other packages, using *alias* or *import* keywords.

ⓘ 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

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.

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

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

    private part def Automobile;

    public alias Automobile as Car;

    import ISO::TorqueValue as Torque;
}
```

Comments

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

A comment can optionally be named.

A comment that begins with `/**` annotates the following element.

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 documentary 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 documentary comment,
         * annotating an explicitly specified element.
        */

    part def Automobile;

    /**
     * This is a documentary comment, annotating the
     * following element.
     */
    alias Automobile as Car;

    // This is a note. It is in the text, but not part
    // of the model.
    import ISO::TorqueValue as Torque;
}
```

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

Documentation

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

Because it is not a member, a documentation comment cannot be named.

Alias and import elements cannot have documentation comments.

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

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

    alias Automobile as Car;

    import ISO::TorqueValue as Torque;
}
```

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.

- ❶ **block** is a synonym for **part def**.
value type is a synonym for **attribute def**.
value is a synonym for **attribute**.

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

- ❶ Definitions and usages are also namespaces that allow **import**.

- ❶ The **attribute** keyword is optional on attribute usages.

```
part def Vehicle {  
    attribute mass : ScalarValues::Real;  
  
    part eng : Engine;  
  
    ref part driver : Person;  
}  
  
attribute def VehicleStatus  
import ScalarValues::*;

gearSetting : Integer;  
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 part usages.

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.

Subsetting

```
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;
```

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

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*.

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

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)

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

```
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 = "secr";  
        :>> color = TrafficLightColor::red;  
    }  
}  
  
enum def GradePoints :> Real {  
    A = 4.0;  
    B = 3.0;  
    C = 2.0;  
    D = 1.0;  
    F = 0.0;  
}
```

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.

Items

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;  
    }  
}
```

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.

Individuals and Snapshots

An *individual definition* is an item definition restricted to model a single individual and how it changes over its lifetime.

A *snapshot* is an individual usage at a single instant of time.

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

An individual definition will often specialize an item or part definition for the general class of things the individual is one of.

```
individual def PhysicalContext;
individual def Vehicle_1 :> Vehicle;

individual context : PhysicalContext {

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

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

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

Snapshot Succession

```
individual context : PhysicalContext {  
  
    snapshot vehicle_1_t0 : Vehicle_1 {  
        :>> mass = 2000.0;  
        :>> status {  
            :>> gearSetting = 0;  
            :>> acceleratorPosition = 0.0;  
        }  
    }  
  
    snapshot vehicle_1_t1 : Vehicle_1 {  
        :>> mass = 1500.0;  
        :>> status {  
            :>> gearSetting = 2;  
            :>> acceleratorPosition = 0.5;  
        }  
    }  
  
    succession vehicle_1_t0 then vehicle_1_t1;  
}
```

A *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 def Vehicle_1 :> Vehicle {  
    part leftFrontWheel : Wheel;  
    part rightFrontWheel : Wheel;  
}  
  
individual def Wheel_1 :> Wheel;  
  
individual vehicle_1 : Vehicle_1 {  
  
    snapshot vehicle_1_t0 {  
        snapshot leftFrontWheel_t0 : Wheel_1 :>> leftFrontWheel;  
    }  
  
    then snapshot vehicle_1_t1 {  
        snapshot rightFrontWheel_t1 : Wheel_1 :>> rightFrontWheel;  
    }  
}
```

By default, these are snapshots of the containing individual.

During the first snapshot of [Vehicle_1](#), [Wheel_1](#) has the role of the [leftFrontWheel](#).

This is a shorthand for a succession between the lexically previous snapshot and this snapshot.

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.

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

individual : Vehicle_1 {

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

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

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

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

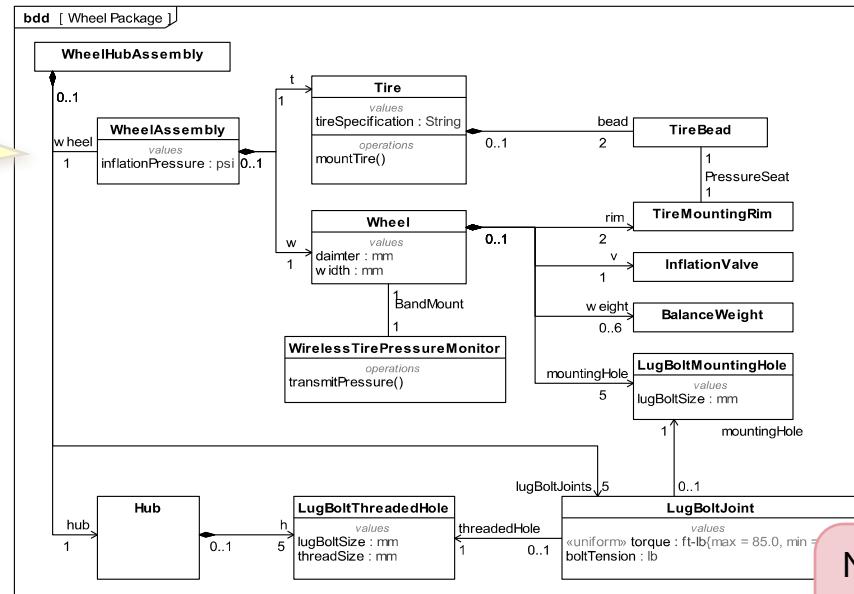
During this time slice of `Vehicle_1`, the `Alice` has the role of the `driver`.

During a latter time slice of `Vehicle_1`, `Bob` has the role of the `driver`.

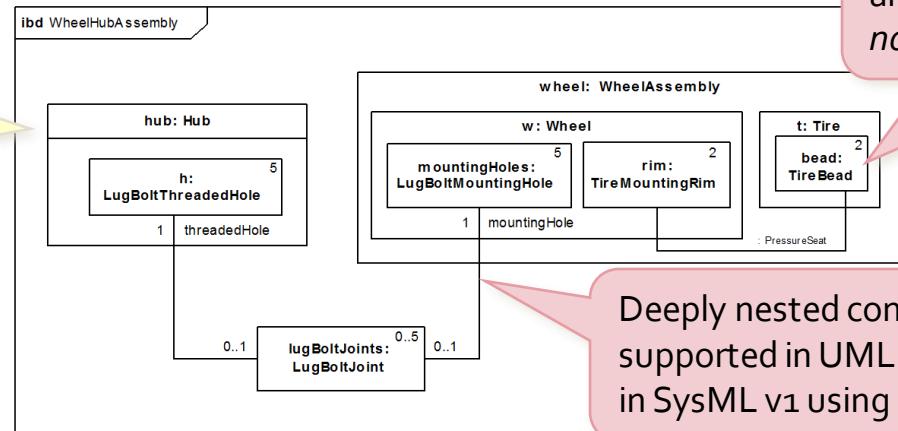
Connectors

Example from SysML v1.6 Spec

Decomposition and association are defined on the BDD.



Usage-level
interconnection is
defined on the IBD.



Nested parts on an IBD are always properties of the *type*, not the containing part.

Deeply nested connection is *not* supported in UML and must be added in SysML v1 using property paths.

Connections (1)

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

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

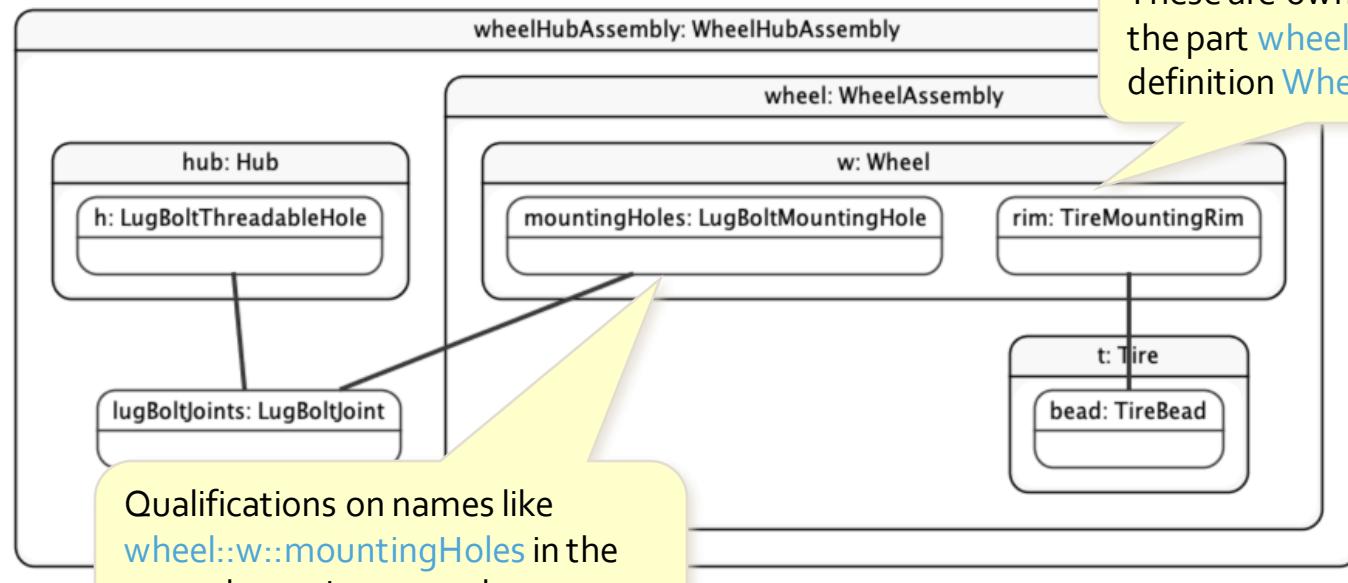
If a connection definition is not specified, a generic connection definition (called Connection) is used.

```
connection def PressureSeat {  
    end : TireBead[1];  
    end : TireMountingRim[1];  
}  
  
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 t.bead to w.rim;  
    }  
  
    part lugBoltJoints : LugBoltJoint[0..5];  
    part hub : Hub[1] {  
        part h : LugBoltThreadableHole[5];  
    }  
    connect lugBoltJoints[0..1]  
        to mountingHole => wheel.w.mountingHoles[1];  
    connect lugBoltJoints[0..1]  
        to threadedHole => hub.h[1];  
}
```

① **assoc block** is a synonym for **connection def**.
link is a synonym for **connection**.

“Dot” notation is used to specify paths to nested features to be connected.

Connections (2)



Ports

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

- ① Flow 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 {  
    value 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 *flow feature*.

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

Port Conjugation

Every port definition has an implicit *conjugate* port definition that reverses input and output flows. 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 short had for using the conjugate port definition (e.g., `FuelPort:'~FuelPort'`).

Interfaces

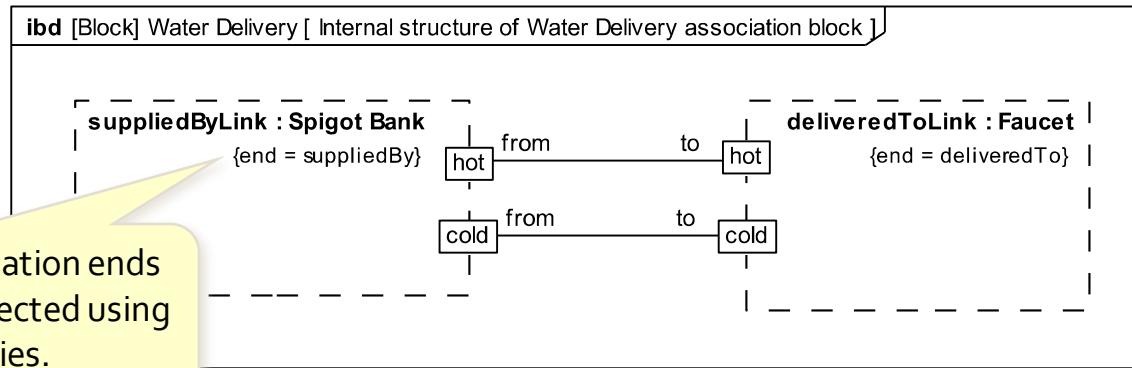
An *interface definition* is a connection definition whose ends are ports.

```
interface def FuelInterface {  
    end supplierPort : FuelOutPort;  
    end consumerPort : FuelInPort;  
}  
  
part vehicle : Vehicle {  
    part tankAssy : FuelTankAssembly;  
    part eng : Engine;  
  
    interface : FuelInterface connect  
        supplierPort => tankAssy.fuelTankPort to  
        consumerPort => eng.engineFuelPort;  
}
```

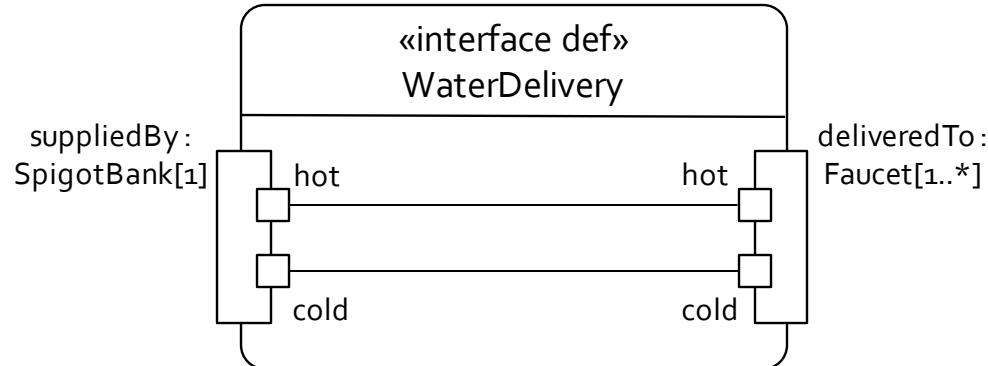
An *interface usage* is a connection usage defined by an interface definition, connecting two compatible ports.

Interface Decomposition

Example from SysML v1.6 Spec



Interface Decomposition



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

In SysML v2, connection ends have multiplicities corresponding to navigating across the connection...

...but they can be interconnected like participant properties.

Binding Connection (1)

```
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).

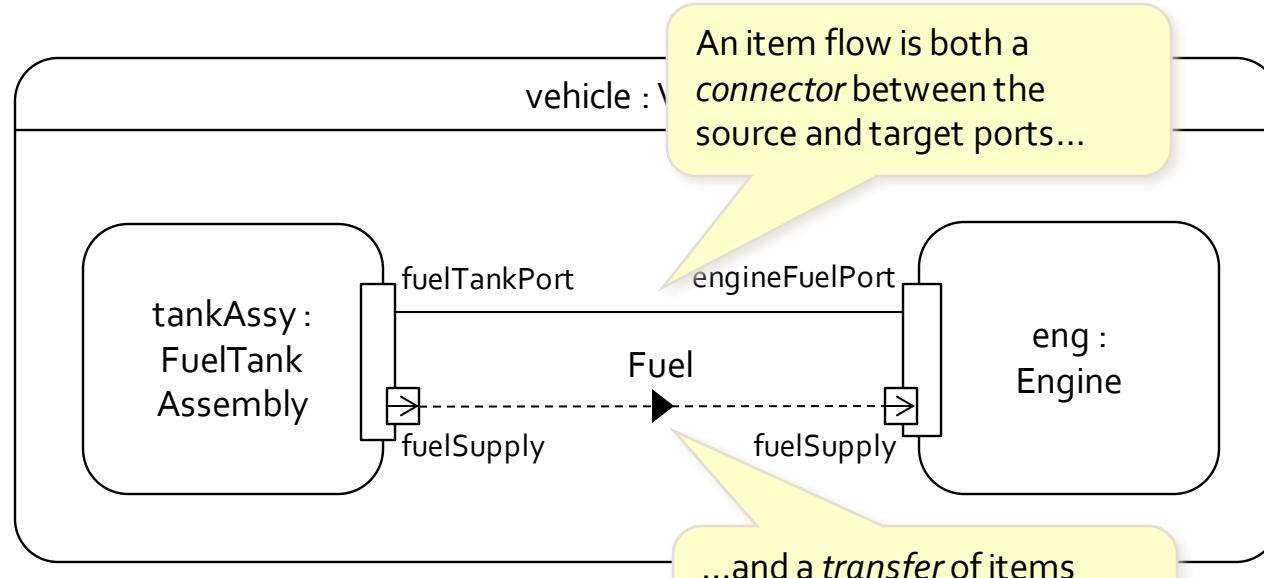
Binding Connections (2)

```
part tank : FuelTankAssembly {  
    port redefines fuelTankPort {  
        out item redefines fuelOut : Fuel;  
        in item redefines fuelIn : Fuel = fuelTankPort.fuelSupply;  
    }  
  
    part pump : FuelPump {  
        out item pumpOut : Fuel = fuelTankPort.fuelSupply;  
        in item pumpIn : Fuel;  
    }  
    part tank : FuelTank {  
        out item fuelOut : Fuel;  
        in item fuelIn : Fuel = fuelTankPort.fuelReturn;  
    }  
}
```

This shorthand notation combines the definition of a feature with a binding connection.

- This is *not* the same as an initial or default value, like in UML.

Item Flows



- ① An item 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.

Streaming Item Flows

- Currently feature paths (dot notation) *cannot* be used for item flows. This means that redefinitions must be used as necessary to provide “local” connection points.

An *item flow* is a transfer of items from an output of a source port to an input of a target port.

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

```
part vehicle : Vehicle {  
    part tankAssy : FuelTankAssembly {  
        port redefines fuelTankPort {  
            out item redefines fuelSupply;  
            in item redefines fuelReturn;  
        }  
    }  
}
```

```
part eng : Engine {  
    port redefines engineFuelPort {  
        in item redefines fuelSupply;  
        out item redefines fuelReturn;  
    }  
}
```

```
stream of Fuel  
from tankAssy::fuelTankPort::fuelSupply  
to eng::engineFuelPort::fuelSupply;
```

```
stream of Fuel  
from eng::engineFuelPort::fuelReturn  
to tankAssy::fuelTankPort::fuelReturn;  
}
```

- Ends must be specified using qualified names, *not* feature paths.

Streaming Interfaces

```
interface def FuelInterface {  
    end supplierPort : FuelOutPort {  
        out item redefines fuelSupply;  
        in item redefines fuelReturn;  
    }  
    end consumerPort : FuelInPort {  
        in item redefines fuelSupply;  
        out item redefines fuelReturn;  
    }  
  
    stream supplierPort::fuelSupply to consumerPort::fuelSupply;  
    stream consumerPort::fuelReturn to supplierPort::fuelReturn;  
}  
...  
  
interface : FuelInterface connect  
supplierPort => tankAssy.fuelTankPort to  
consumerPort => eng.engineFuelPort;
```

Item flows can be defined within an interface definition.

The flows are established when the interface is used.

Action Definitions (1)

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

① **activity** is a synonym for **action def**.

Action definitions may have **in**, **out** and **inout** *parameters* (the default is **in**).

```
action def Focus(in scene : Scene, out image : Image);
action def Shoot(in image : Image, out picture : Picture);
```

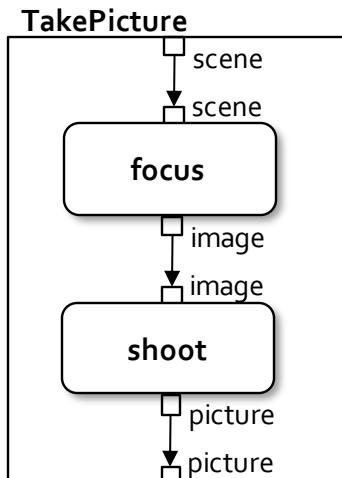
```
action def TakePicture
  (in scene : Scene,
   out picture : Picture) {
    bind focus.scene = scene;
```

```
action focus : Focus (in scene, out image);
```

```
flow focus::image to shoot::image;
```

```
action shoot : Shoot (in image, out picture);
```

```
bind shoot.picture = picture;
```



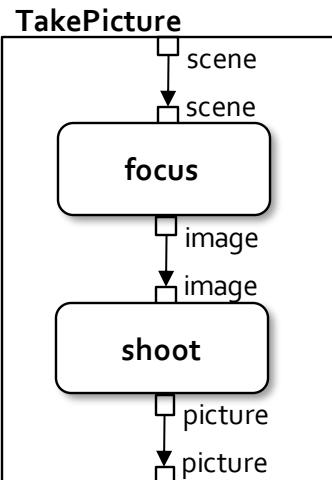
An *action* is a usage of an action definition performed in a specific context.

An item flow can be used to transfer items between actions.

An action has parameters corresponding to its action definition.

① The use of a non-streaming flow means that **focus** must finish producing its output before **shoot** can begin consuming it. (Streaming can also be used.)

Action Definitions (2)



```
action def Focus(in scene : Scene, out image : Image);  
action def Shoot(in image : Image, out picture : Picture);
```

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

Parameters can also be declared in the action body, similarly to flow features.

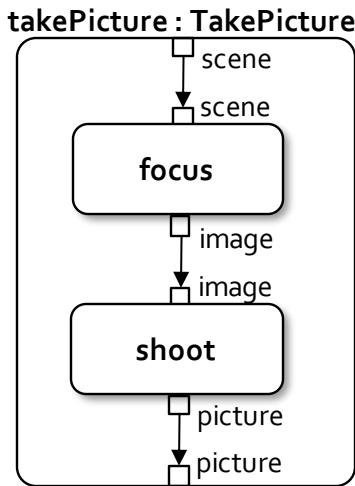
```
action focus : Focus {  
    in item scene = TakePicture::scene;  
    out item image;  
}
```

This is a shorthand for an item flow *into* the parameter.

```
action shoot : Shoot {  
    in item image flow from focus::image;  
    out item picture = TakePicture::picture;  
}
```

This is the same shorthand for binding used previously.

Action Decomposition

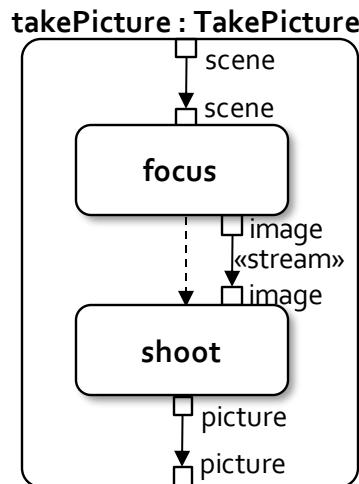


```
action def Focus(in scene : Scene, out image : Image);  
action def Shoot(in image : Image, out picture : Picture);  
action def TakePicture(in scene : Scene, out picture : Picture);  
  
action takePicture : TakePicture {  
    in item scene;  
    out item picture;  
  
    action focus : Focus {  
        in item scene = takePicture::scene;  
        out item image;  
    }  
  
    action shoot : Shoot {  
        in item image flow from focus::image;  
        out item picture = takePicture::picture;  
    }  
}
```

An action can also be directly decomposed into other actions.

- ⓘ This is just a composite structure as before, only typed by action definitions instead of part definitions. Unlike SysML v1, “adjunct properties” are not needed!

Action Succession (1)



```
action takePicture : TakePicture {
    in item scene;
    out item picture;

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

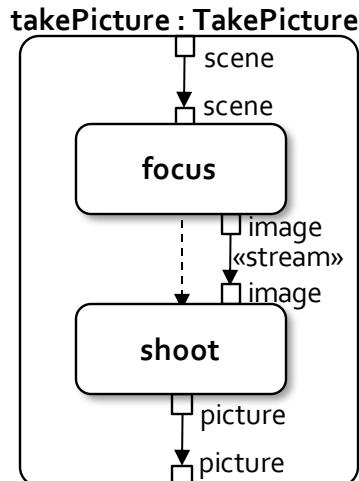
    succession focus then shoot;

    action shoot : Shoot {
        in item image stream from focus::image;
        out item picture = takePicture::picture;
    }
}
```

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

With the succession modeled explicitly, a stream item flow can be used here.

Action Succession (2)



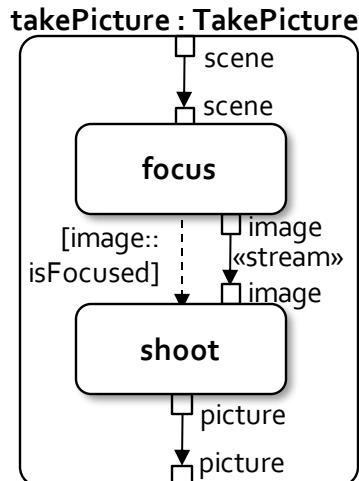
```
action takePicture : TakePicture {
    in item scene;
    out item picture;

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

    then action shoot : Shoot {
        in item image stream from focus::image;
        out item picture = takePicture::picture;
    }
}
```

This is a shorthand for a succession between the lexically previous action and this action.

Conditional Succession (1)



```
action takePicture : TakePicture {
    in item scene;
    out item picture;

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

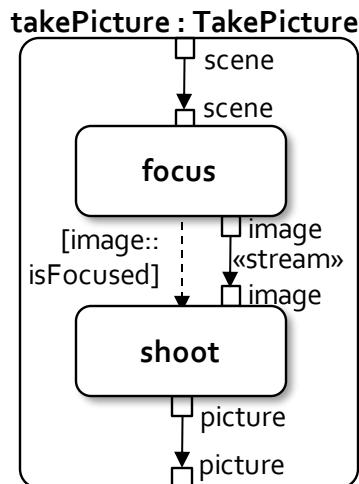
    succession focus
        if focus.image.isFocused then shoot;

    action shoot : Shoot {
        in item image stream from focus::image;
        out item picture = takePicture::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.

Conditional Succession (2)



```
action takePicture : TakePicture {
    in item scene;
    out item picture;

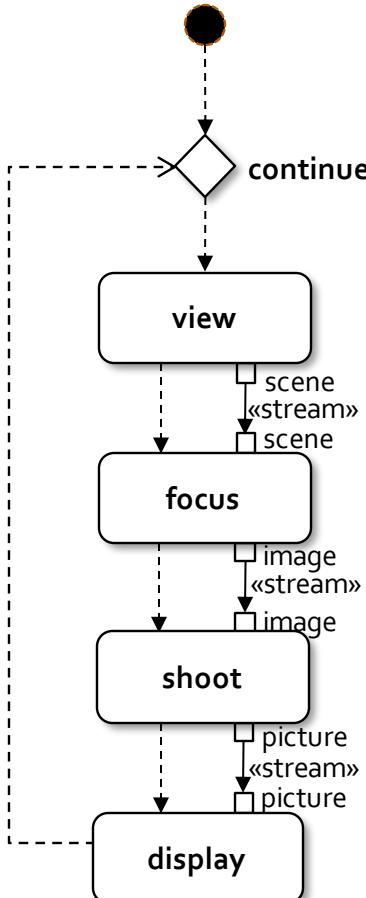
    action focus : Focus {
        in item scene = takePicture::scene;
        out item This is a shorthand for a conditional succession
    } following the lexically previous action.

    if focus::image::isFocused then shoot;

    action shoot : Shoot {
        in item image stream from focus::image;
        out item picture = takePicture::picture;
    }
}
```

Merge Nodes

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



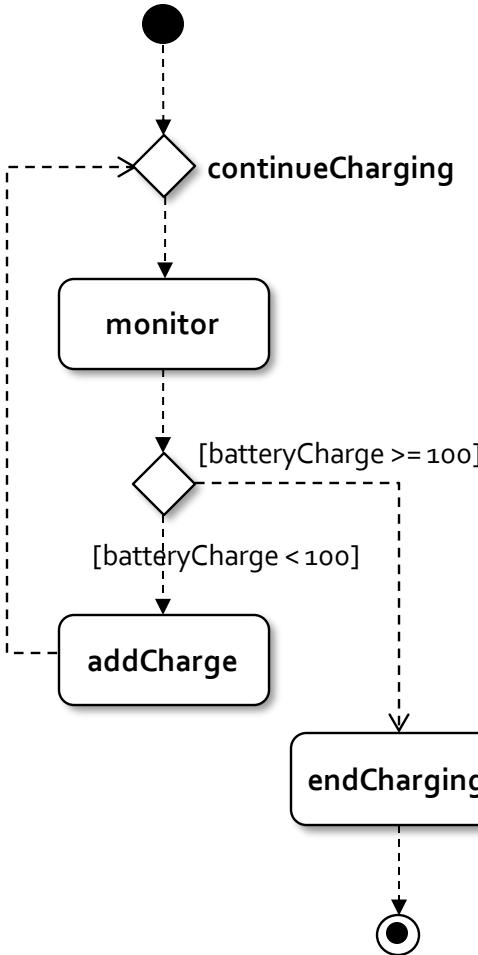
```

action takePicture : TakePicture {
    first start;
    then merge continue;
    then action trigger { out item scene; }
    then action focus : Focus {
        in item scene stream from trigger::scene;
        out item image;
    }
    then action shoot : Shoot {
        in image stream from focus::image;
        out picture;
    }
    then action display {
        in picture stream from shoot::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 item batteryCharge : Real;
    }

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

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

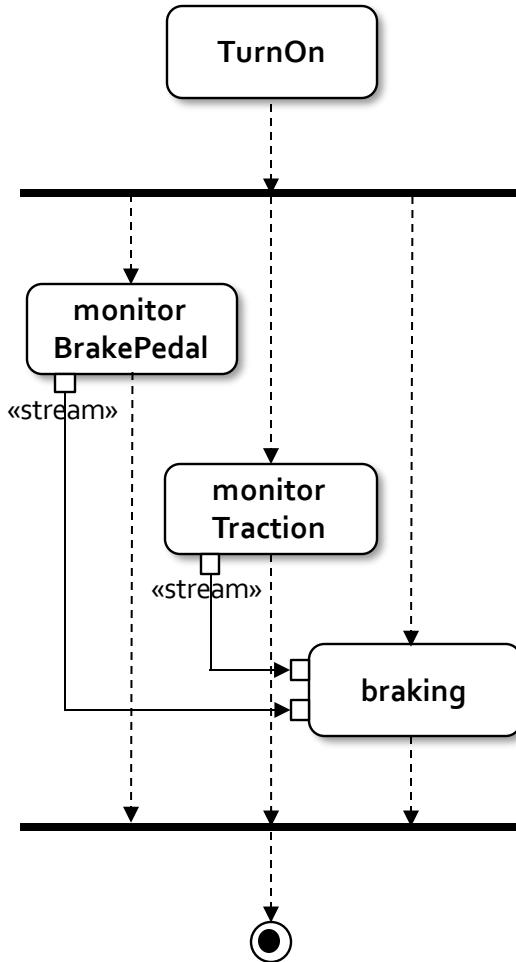
        action endCharging : EndCharging {
            in item charge = monitor::batteryCharge;
        }
        then done;
}
```

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.

Fork and Join Nodes



```

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;

action braking : Braking {
  in brakePressure stream from
    monitorBrakePedal::brakePressure;
  in modulationFrequency stream from
    monitorTraction::modulationFrequency; }
then joinNode;

join joinNode;
then done;
}
  
```

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

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

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

Performed Actions

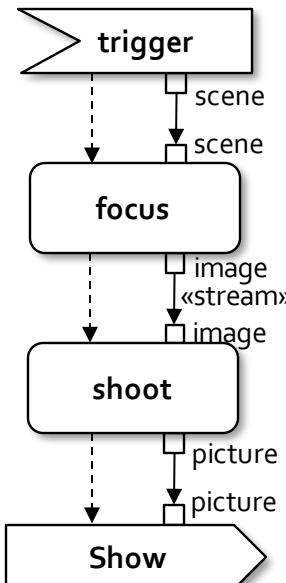
perform identifies the owner as the performer of an action.

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

```
part camera : Camera {  
  
    perform action takePhoto[*] ordered  
        :> takePicture;  
  
    part f : AutoFocus {  
        perform takePhoto::focus;  
    }  
  
    part i : Imager {  
        perform takePhoto::shoot;  
    }  
}
```

Asynchronous Messaging

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



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

This is the *name* of the action.

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

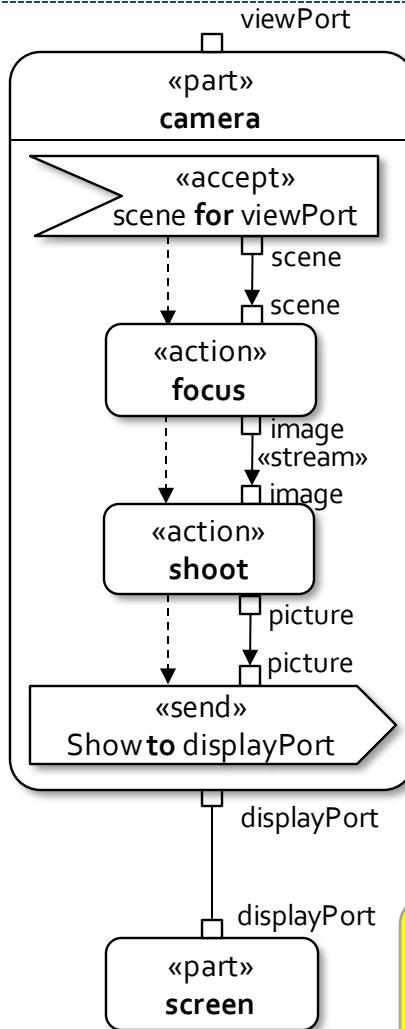
```
action takePicture : TakePicture {
    action trigger accept scene : Scene;
    then action focus : Focus {
        in item scene = trigger.scene;
        out item image;
    }
    then action shoot : Shoot {
        in item image stream from focus::image;
        out item picture;
    }
    then send Show(shoot.picture) to screen;
}
```

- ⓘ The notation “**Show(...)**” means to create an instance of the definition **Show** with the given value for its nested attribute.

This is an *expression* evaluating to the value to be sent.

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

Asynchronous Messaging through Ports



```

part camera : Camera {
  port viewPort;
  port displayPort;

  perform action takePicture : TakePicture {
    action trigger accept scene : Scene for viewPort;

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

    then send Show(shoot.picture) to displayPort;
  }
}
  
```

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

- An asynchronous transfer can propagate over a connection, but it is *not* currently modeled as an item flow.

A send action can target a specific port, meaning that the transfer is sent out over that port.

Opaque Actions

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

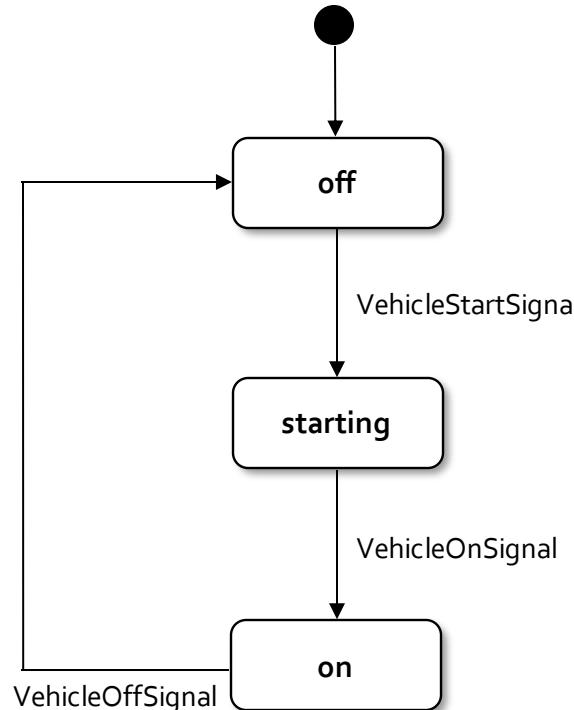
```
action def UpdateSensors (in sensors : Sensor[*]) {  
    language "Alf"  
    /*  
     * for (sensor in sensors) {  
     *     if (sensor.ready) {  
     *         Update(sensor);  
     *     }  
     * }  
     */  
}
```

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.

- ⓘ 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.



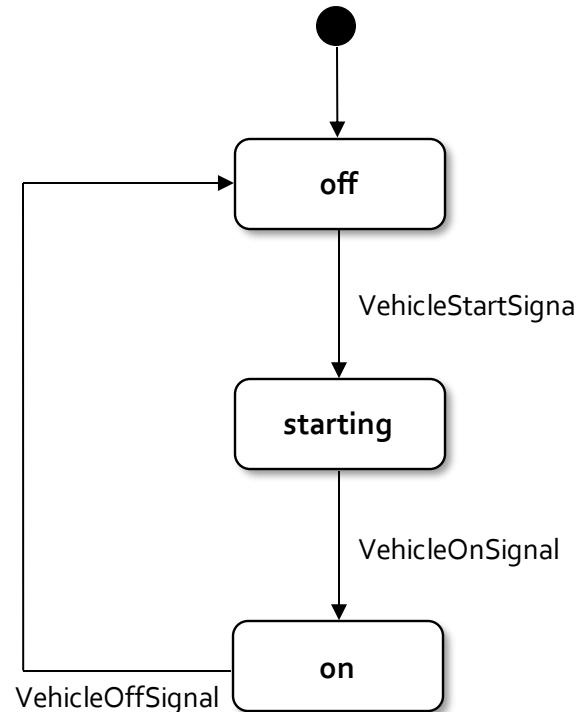
```
state def VehicleStates {  
    entry; then off;  
  
    state off;  
  
    transition off_to_starting  
        first off  
        accept VehicleStartSignal  
        then starting;  
  
    state starting;  
  
    transition starting_to_on  
        first starting  
        accept VehicleOnSignal  
        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)

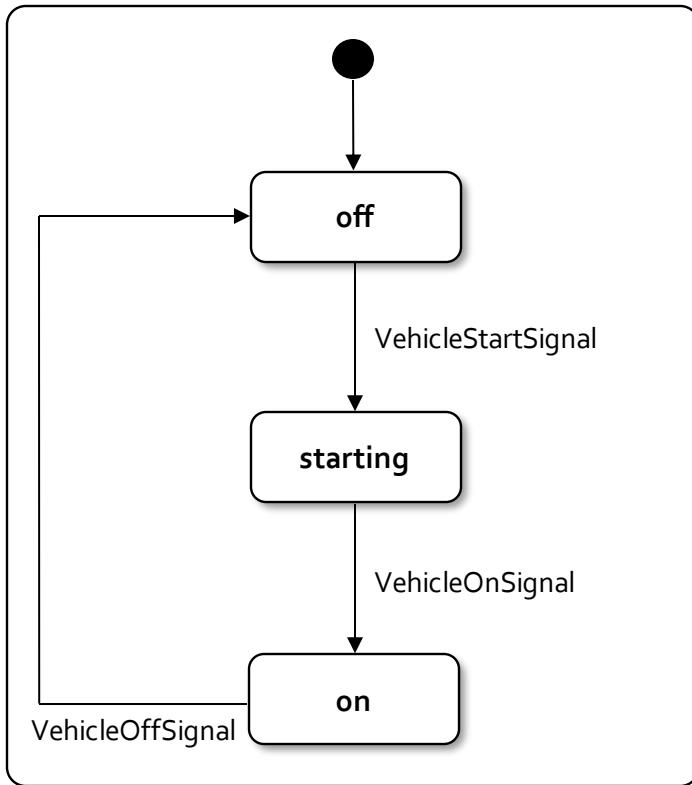


```
state def VehicleStates {  
    entry; then off;  
  
    state off;  
    accept VehicleStartSignal  
        then starting;  
  
    state starting;  
    accept VehicleOnSignal  
        then on;  
  
    state on;  
    accept VehicleOffSignal  
        then off;  
}
```

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

State Decomposition

vehicleStates : VehicleStates



```
state def VehicleStates;  
  
state vehicleStates : VehicleStates {  
    entry; then off;  
  
    state off;  
    accept VehicleStartSignal  
        then starting;  
  
    state starting;  
    accept VehicleOnSignal  
        then on;  
  
    state on;  
    accept VehicleOffSignal  
        then off;  
}
```

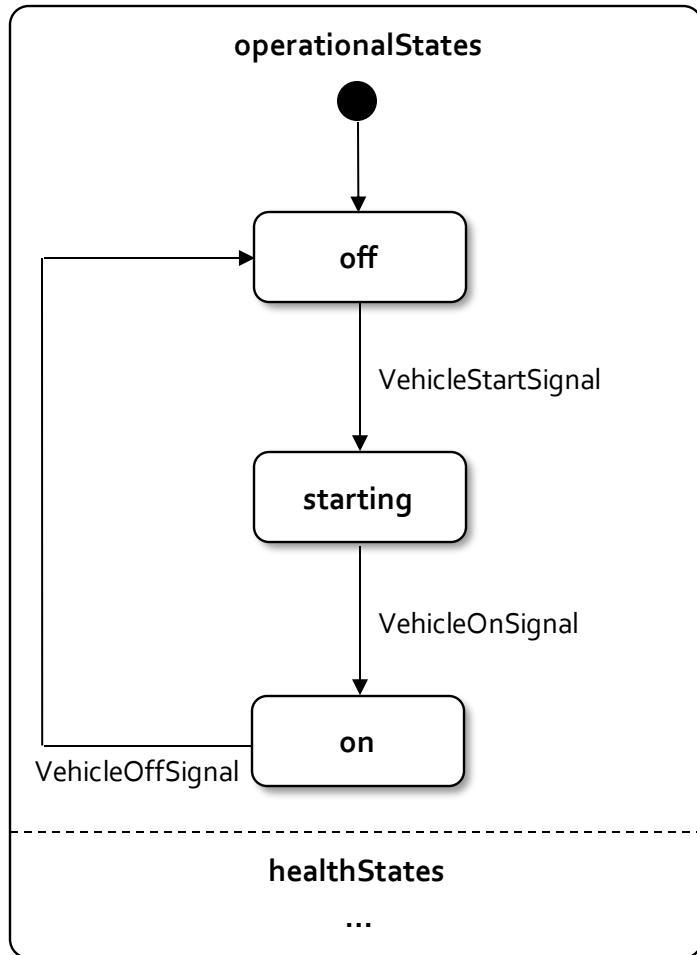
A state can be explicitly declared to be a usage of a state definition.

A state can also be directly decomposed into other states.

- ⓘ This is just a part structure as before, only typed by state definitions instead of blocks.

Concurrent States

vehicleStates : VehicleStates



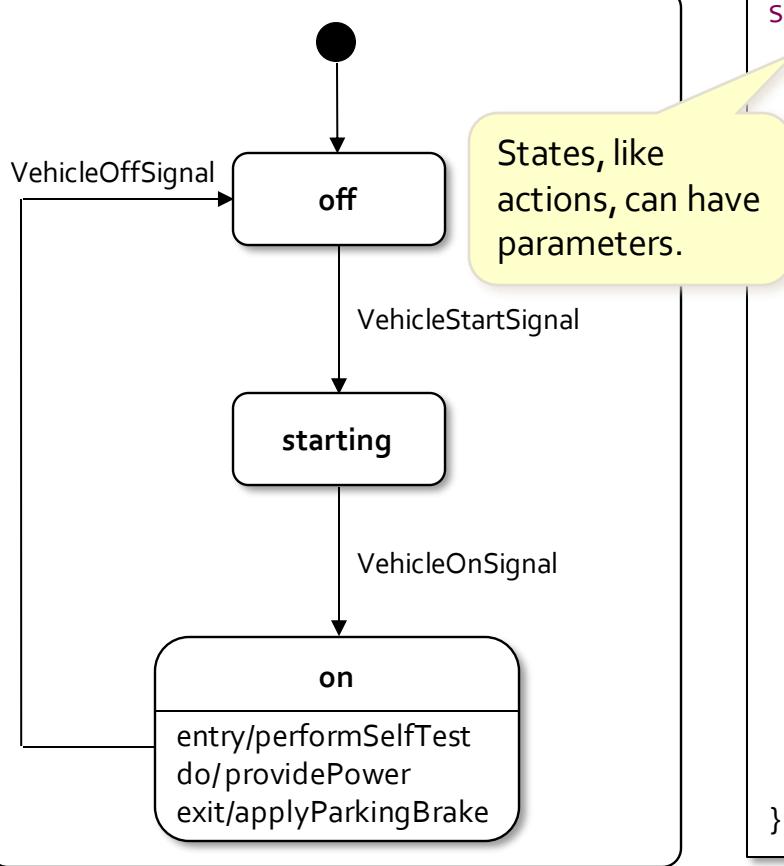
```
state def VehicleStates;  
  
state vehicleStates : VehicleStates {  
  
    state operationalStates {  
        entry; then off;  
  
        state off;  
        accept VehicleStartSignal  
            then starting;  
  
        state starting;  
        accept VehicleOnSignal  
            then on;  
  
        state on;  
        accept VehicleOffSignal  
            then off;  
    }  
  
    state healthStates {  
        ...  
    }  
}
```

If no initial state is specified, then nested states are presumed to be concurrent.

State Entry, Do and Exit Actions

- 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.

`vehicleStates : VehicleStates`



```

action performSelfTest(vehicle : Vehicle);

state def VehicleStates(operatingVehicle : Vehicle);

state vehicleStates : VehicleStates
(operatingVehicle : Vehicle) {

entry; 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;
}
  
```

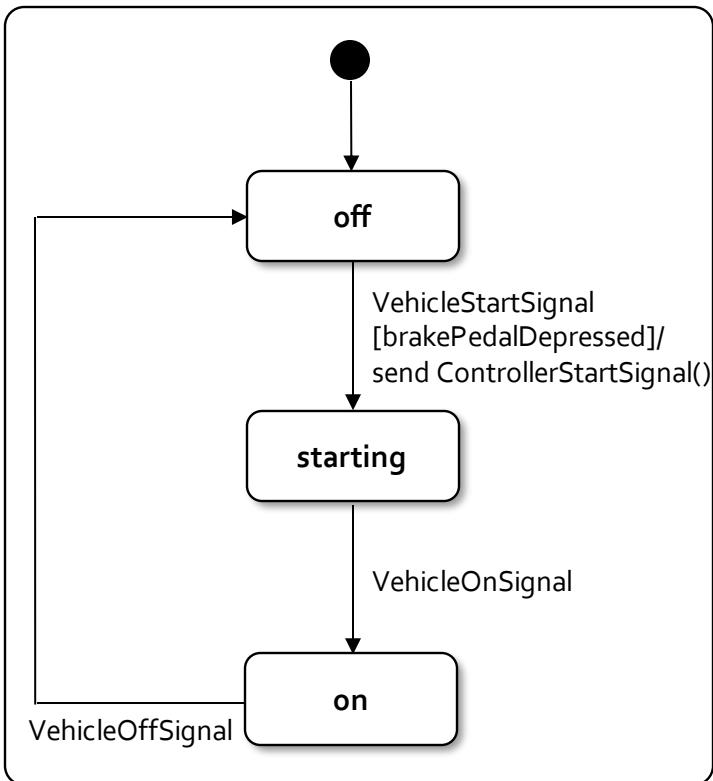
Entry, do and exit actions come before nested states and transitions.

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

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

Transition Guards and Effect Actions

`vehicleStates : VehicleStates`



```

action performSelfTest(vehicle : Vehicle);

state def VehicleStates(operatingVehicle : Vehicle);

state vehicleStates : VehicleStates (
  operatingVehicle : Vehicle,
  controller : VehicleController)

entry; then off;

state off;
accept VehicleStartSignal
  if operatingVehicle::brakePedalDepressed
  do send 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.

Exhibited States

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

```
part vehicle : Vehicle {  
  
    part vehicleController : VehicleController;  
  
    exhibit vehicleStates {  
        in ref operatingVehicle = vehicle;  
        in ref controller = vehicleController;  
    }  
}
```

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

Expressions

Mass Rollup Example (1)

```
package MassRollup {  
    import ScalarFunctions::*;

    part def MassedThing {  
        attribute mass subsets ISQ::mass;  
        attribute totalMass subsets ISQ::mass;  
    }

    part simpleThing : MassedThing {  
        attribute totalMass redefines MassedThing::totalMass = mass;  
    }

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

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

From the *International System of Quantities* library model.

Binding connector.

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.

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

Expressions

Mass Rollup Example (2)

```
import ScalarValues::*;
import MassRollup::*;

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

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

// Example usage
import SI::*;
part c :> car {
    redefines car::mass = 1000[kg];
    part redefines engine {
        redefines engine::mass = 100[kg];
    }
    part redefines transmission {
        redefines transmission::mass = 50[kg];
    }
}

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

Units are identified on
the *value, not the type.*

- The previous `@[...]` for giving the units of a value is now *deprecated*. Use just `[...]` instead.

Calculation Definitions

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

```
calc def Power (  
    whlpwr : PowerValue,  
    Cd : Real,  
    Cf : Real,  
    tm : MassValue,  
    v : SpeedValue ) : PowerValue {
```

Calculation parameters are similar to the parameters on actions.

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.

```
calc def Acceleration (tp: PowerValue, tm : MassValue, v: SpeedValue) : AccelerationValue {  
    tp / (tm * v)  
}
```

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

```
calc def Velocity (dt : TimeValue, v0 : SpeedValue, a : AccelValue) : SpeedValue  
= v0 + a * dt;
```

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

If there are no intermediate computations, a shortened form can be used.

Calculation Usages (1)

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

- ➊ Note that the return result is *outside* the parentheses. The `return` keyword is optional.

```
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,
        out v_out : SpeedValue, out x_out : LengthValue) {

        calc acc : Acceleration (
            tp = Power(wheelPower, C_d, C_f, mass, v_in) ,
            tm = mass,
            v = v_in
        ) return a;

        calc vel : Velocity (
            dt = delta_t,
            v0 = v_in,
            a = acc.a
        ) return v = v_out;

        calc pos : Position (
            dt = delta_t,
            x0 = x_in,
            v = vel.v
        ) return x = x_out;
    }
}
```

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 and/or bound (like action 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 : Mass;  
}  
  
calc updateState  
(delta_t : TimeValue, currState : DynamicState) {  
  
    attribute totalPower : PowerValue = Power(wheelPower, C_d, C_f, mass, currState::v);  
  
    return attribute newState : DynamicState {  
        :>> v = Velocity(delta_t, currState::v, Acceleration(totalPower, mass, currState::v));  
        :>> x = Position(delta_t, currState::x, currState::v);  
    }  
}
```

A calculation can be specified without an explicit calculation definition.

Calculations can also handle structured values.

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

Constraint Definitions (1)

- 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.

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

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

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

```
constraint def MassConstraint (
    partMasses : MassValue[0..*],
    massLimit : MassValue) {

    sum(partMasses) <= massLimit
}
```

Constraint parameters are similar to the parameters on actions.

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

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

```
attribute chassisMass : MassValue;
```

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

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

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

Constraint Definitions (2)

```
import ISQ::*;
import SI::*;
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)

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

constraint def MassConstraint (
    partMasses : MassValue[0..*],
    massLimit : MassValue) {

    sum(partMasses) <= massLimit
}

part def Vehicle {
    assert constraint massConstraint : MassConstraint (
        partMasses = (chassisMass, engine.mass, transmission.mass),
        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.

ⓘ 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 (
    partMasses : MassValue[0..*],
    massLimit : MassValue);

constraint massConstraint : MassConstraint (
    partMasses : MassValue[0..*],
    massLimit : MassValue) {
    sum(partMasses) <= massLimit
}

part def Vehicle {
    assert massConstraint (
        partMasses = (chassisMass, engine.mass, transmission.mass),
        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

```
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 (  
    mass: MassValue,  
    initialSpeed : SpeedValue,  
    finalSpeed : SpeedValue,  
    deltaT : TimeValue,  
    force : ForceValue ) {  
  
    force * deltaT == mass * (finalSpeed - initialSpeed) &  
    mass > 0[kg]  
}
```

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

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(
    p : PowerValue, m : MassValue, dt : TimeValue,
    x_i : LengthValue, v_i : SpeedValue,
    x_f : LengthValue, v_f : VelocityValue,
    a : AccelerationValue
) {
    attribute v_avg : VelocityValue = (v_i + v_f)/2;
    a == Acceleration(p, m, v_avg) &
    v_f == Velocity(dt, v_i, a) &
    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 : VelocityValue,
    out x_out : LengthValue, out v_out : VelocityValue,
    out a_out : AccelerationValue
) {
    assert constraint dynamics : StraightLineDynamicsEquations (
        p = power, m = mass, dt = delta_t,
        x_i = x_in, v_i = v_in,
        x_f = x_out, v_f = v_out,
        a = a_out
    );
}
  
```

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

❶ 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.

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

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

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;  
    ...  
}  
  
requirement def id '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 may have a modeler specified *human id*, which is an alternate name for it.

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

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

Requirement Usages

A requirement may optionally have its own human ID.

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

```
requirement id '1.1' fullVehicleMassLimit : VehicleMassLimitationRequirement {  
    subject vehicle : Vehicle;  
    attribute :>> massReqd = 2000[kg];  
  
    assume constraint {  
        doc /* Fuel tank is full. */  
        vehicle.fuelMass == vehicle.fuelFullMass  
    }  
}  
  
requirement id '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.

Requirements can be grouped by reference...

...or by composition.

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

```
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;
    }
}
```

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

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

Requirement Satisfaction

```
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.

- ⓘ 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*.

```
analysis def FuelEconomyAnalysis {  
    subject vehicle : Vehicle;  
    return fuelEconomyResult : DistancePerVolumeValue;
```

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

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

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

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

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

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

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

Analysis Case Definitions (2)

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

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

```
analysis def FuelEconomyAnalysis {
    subject vehicle : Vehicle;
    return fuelEconomyResult : Distance
    ...
    in attribute scenario[*] {
        time : TimeValue;
        position : LengthValue;
        velocity : SpeedValue;
    }
    action solveForPower {
        out power: PowerValue[*];
        out acceleration: AccelerationValue[*];
        assert constraint {
            {1..size(scenario)-1} -> forAll i (
                StraightLineDynamicsEquations (
                    power[i], vehicle.mass,
                    scenario.time[i+1] - scenario.time[i],
                    scenario.position[i], scenario.velocity[i],
                    scenario.position[i+1], scenario.velocity[i+1],
                    acceleration[i+1]))
        }
    }
    then action solveForFuelEconomy {
        in power : PowerValue[*] = solveForPower.power;
        out fuelEconomy : DistancePerVolumeValue = fuelEconomyResult;
        ...
    }
}
```

Additional parameters can be specified in the case body.

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

Analysis Case Usages

```
part vehicleFuelEconomyAnalysisContext {  
  
    requirement vehicleFuelEconomyRequirements{subject vehicle : Vehicle; ... }  
  
    attribute cityScenario :> FuelEconomyAnalysis::scenario = { ... };  
    attribute highwayScenario :> FuelEconomyAnalysis::scenario = { ... };  
  
    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.

```
analysis engineTradeStudy : TradeStudies::TradeStudy {  
    subject : Engine = {engine4cyl, engine6cyl};  
    objective : MaximizeObjective;  
  
    calc :>> objectiveFunction {  
        in part engine :>> alternative : Engine;  
        calc powerRollup: PowerRollup (engine)  
            return power : ISQ::PowerValue;  
        calc massRollup: MassRollup (engine)  
            return mass : ISQ::MassValue;  
        calc efficiencyRollup: EfficiencyRollup (engine)  
            return efficiency : Real;  
        calc costRollup: CostRollup (engine)  
            return cost : Real;  
        return :>> result : Real = EngineEvaluation(  
            powerRollup.power, massRollup.mass,  
            efficiencyRollup.efficiency, costRollup.cost  
        );  
    }  
  
    return part :>> selectedAlternative : Engine;  
}
```

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

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

Redefine the `objectiveFunction` 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 = vehicle.mass;  
    doc /* The vehicle mass shall be less  
        * than or equal to 2500 kg. */  
    require constraint {  
        massActual <= 2500[SI::kg] }  
}  
  
verification def VehicleMassTest {  
    import Verifications::*;  
  
    subject testVehicle : Vehicle;  
  
    objective vehicleMassVerificationObjective {  
        verify vehicleMassRequirement;  
    }  
  
    return verdict : VerdictKind;  
}
```

Parameterizing the requirement allows it to be checked against a measured `massActual`, while asserting 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`.

□ Enumeration definition is not supported for user modeling yet.

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.

ⓘ The use of named parameter notation here is optional.

```
verification def VehicleMassTest {
    import Verifications::*;

    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.

Variation Definitions

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

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

```
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';
}
```

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.

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.

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.

```
abstract part vehicleFamily : Vehicle {
    part engine : EngineChoices[1];
    variation part transmission : Transmission[1] {
        variant manualTransmission;
        variant automaticTransmission;
    }
    assert constraint {
        (engine == engine::'4cylEngine' &
         transmission == transmission::manualTransmission) ^
        (engine == engine::'6cylEngine' &
         transmission == transmission::automaticTransmission)
    }
}
```

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

- ① The operator `&` means "and" and the operator `^` 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.

Dependencies

```
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 or 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.

Allocation

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 {  
    import LogicalModel::*;

    part powerTrain : PowerTrain {  
        part engine : Engine {  
            perform generateTorque;  
        }  
    }  
  
    allocate torqueGenerator to powerTrain {  
        allocate torqueGenerator.generateTorque  
            to powerTrain.engine.generateTorque;  
    }  
}
```

- ➊ 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.

Allocation Definition

- 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 {  
    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 can be defined using regular attribute definitions.

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;  
    }  
  
    attribute def SafetyFeature;  
    attribute def SecurityFeature;  
  
    metadata SafetyFeature about  
        vehicle::interior::seatBelt,  
        vehicle::interior::driverAirBag,  
        vehicle::bodyAssy::bumper;  
  
    metadata SecurityFeature about  
        vehicle::interior::alarm,  
        vehicle::bodyAssy::keylessEntry;
```

- 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)

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

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

```
action computeDynamics {  
    import AnalysisTooling::*;  
  
    metadata ToolExecution {  
        toolName = "ModelCenter";  
        uri = "aserv://localhost/Vehicle/Equation1";  
    }  
  
    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";}}  
}
```

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

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

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

Element Import Filtering (1)

```
attribute 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' {  
    import vehicle::**;  
    filter @Safety;  
}  
  
package 'Mandatory Safety Features' {  
    import vehicle::**;  
    filter @Safety && 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. In this context, `@` is a shorthand for checking if an element has the given metadata annotation.

Element Import Filtering (2)

```
attribute 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;} }  
    }  
}
```

- ❶ The `filter` keyword is only allowed in a package, 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' {  
    import vehicle::**[@Safety];  
}  
  
package 'Mandatory Safety Features' {  
    > import vehicle::**  
        [@Safety && 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.

Kernel Model Library (selected models)

Scalar Values

```
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 Integer specializes Number;
    datatype UnlimitedNatural specializes Number;
    datatype Natural specializes Integer, UnlimitedNatural;
    datatype Rational specializes Number;
    datatype Real specializes Number;
    datatype Complex specializes Number;
}
```

Base Functions

```
package BaseFunctions {
    private import Base::*;
    private import ScalarValues::*;

    function =='(x: Anything, y: Anything): Boolean;
    function !='(x: Anything, y: Anything): Boolean;

    function ToString(x: Anything): String;

    function size(seq: Anything[*]): Natural;
    function isEmpty(seq: Anything[*]): Boolean;
    function notEmpty(seq: Anything[0..*]): Boolean;
    function head(seq: Anything[0..*]): Boolean;
    function tail(seq: Anything[0..*]): Boolean;
    function last(seq: Anything[0..*]): Boolean;

    ...
}
```

Data Functions

```
package DataFunctions {  
    private import Base:::DataValue;  
    private import ScalarValues::Boolean;  
  
    abstract function '+'(x: DataValue, y: DataValue[0..1]): ;  
    abstract function '-'(x: DataValue, y: DataValue[0..1]): DataValue;  
    abstract function '*'(x: DataValue, y: DataValue): DataValue;  
    abstract function '/'(x: DataValue, y: DataValue): DataValue;  
    abstract function '**'(x: DataValue, y: DataValue): DataValue;  
    abstract function '%' (x: DataValue, y: DataValue): DataValue;  
    abstract function '!'(x: DataValue): DataValue;  
    abstract function '~~'(x: DataValue): DataValue;  
    abstract function '|'(x: DataValue, y: DataValue): DataValue;  
    abstract function '^'(x: DataValue, y: DataValue): DataValue;  
    abstract function '&'(x: DataValue, y: DataValue): DataValue;  
    abstract function '<'(x: DataValue, y: DataValue): Boolean;  
    abstract function '>'(x: DataValue, y: DataValue): Boolean;  
    abstract function '<='(x: DataValue, y: DataValue): Boolean;  
    abstract function '>='(x: DataValue, y: DataValue): Boolean;  
    abstract function Max(x: DataValue, y: DataValue): DataValue;  
    abstract function Min(x: DataValue, y: DataValue): DataValue;  
    abstract function '..'(lower: DataValue, upper: DataValue): DataValue[0..*];  
    abstract function sum(collection: DataValue[0..*]): DataValue;  
    abstract function product(collection: DataValue[0..*]);  
}
```

ⓘ Generalizes previous
ScalarFunctions package.

Boolean Functions

```
package BooleanFunctions {  
    import ScalarValues::*;

    function '!' specializes ScalarFunctions:'!'(x: Boolean): Boolean;

    function '|' specializes ScalarFunctions:'|'(x: Boolean, y: Boolean): Boolean;
    function '^' specializes ScalarFunctions:'^'(x: Boolean, y: Boolean): Boolean;
    function '&' specializes ScalarFunctions:'&'(x: Boolean, y: Boolean): Boolean;

    function '==' specializes BaseFunctions:'=='(x: Boolean, y: Boolean): Boolean;
    function '!=' specializes BaseFunctions:'!='(x: Boolean, y: Boolean): Boolean;

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

String Functions

```
package StringFunctions {  
    import ScalarValues::*;

    function '+' specializes ScalarFunctions::'+'(x: String, y:String): String;

    function Size(x: String): Natural;
    function Substring(x: String, lower: Integer, upper: Integer): String;

    function '<' specializes ScalarFunctions::'<'(x: String, y: String): Boolean;
    function '>' specializes ScalarFunctions::'>'(x: String, y: String): Boolean;
    function '<=' specializes ScalarFunctions::'<='(x: String, y: String): Boolean;
    function '>=' specializes ScalarFunctions::'>='(x: String, y: String): Boolean;

    function '=' specializes BaseFunctions::'=='(x: String, y: String): Boolean;
    function '!=' specializes BaseFunctions::'!='(x: String, y: String): Boolean;
    function ToString specializes BaseFunctions::ToString(x: String): String;
}
```

Integer Functions

```

package IntegerFunctions {
  import ScalarValues::*;

  function Abs specializes NumericalFunctions::Abs (x: Integer): Natural;

  function '+' specializes NumericalFunctions::'+' (x: Integer, y: Integer[0..1]): Integer;
  function '-' specializes NumericalFunctions::'-' (x: Integer, y: Integer[0..1]): Integer;
  function '*' specializes NumericalFunctions::'*' (x: Integer, y: Integer): Integer;
  function '/' specializes NumericalFunctions::'/' (x: Integer, y: Integer): Rational;
  function '**' specializes NumericalFunctions::'**' (x: Integer, y: Natural): Integer;
  function '%' specializes NumericalFunctions::'%' (x: Integer, y: Integer): Integer;

  function '<' specializes NumericalFunctions::'<' (x: Integer, y: Integer): Boolean;
  function '>' specializes NumericalFunctions::'>' (x: Integer, y: Integer): Boolean;
  function '<=' specializes NumericalFunctions::'<=' (x: Integer, y: Integer): Boolean;
  function '>=' specializes NumericalFunctions::'>=' (x: Integer, y: Integer): Boolean;

  function Max specializes NumericalFunctions::Max (x: Integer, y: Integer): Integer;
  function Min specializes NumericalFunctions::Min (x: Integer, y: Integer): Integer;

  function '==' specializes BaseFunctions::'==' (x: Integer, y: Integer): Boolean;
  function '!=' specializes BaseFunctions::'!=' (x: Integer, y: Integer): Boolean;
  function '..' specializes ScalarFunctions::'..' (lower: Integer, upper: Integer): Integer[0..*];

  function ToString specializes BaseFunctions::ToString (x: Integer): String;
  function ToNatural(x: Integer): Natural;
  function ToInteger(x: String): Integer;
  function ToRational(x: Integer): Rational;
  function ToReal(x: Integer): Real;

  function sum specializes ScalarFunctions::sum (collection: Integer[0..*]): Integer;
  function product specializes ScalarFunctions::product (collection: Integer[0..*]): Integer;
}

```



Unlimited Natural Functions

SST

```
package UnlimitedNaturalFunctions {
    import ScalarValues::*;

    function '<' specializes NumericalFunctions::'<'(x: UnlimitedNatural, y: UnlimitedNatural): Boolean;
    function '>' specializes NumericalFunctions::'>'(x: UnlimitedNatural, y: UnlimitedNatural): Boolean;
    function '<=' specializes NumericalFunctions::'<='(x: UnlimitedNatural, y: UnlimitedNatural):
        Boolean;
    function '>=' specializes NumericalFunctions::'>='(x: UnlimitedNatural, y: UnlimitedNatural):
        Boolean;

    function Max specializes NumericalFunctions::Min(x: UnlimitedNatural, y: UnlimitedNatural):
        UnlimitedNatural;
    function Min specializes NumericalFunctions::Max(x: UnlimitedNatural, y: UnlimitedNatural):
        UnlimitedNatural;

    function '=' specializes BaseFunctions::'=='(x: UnlimitedNatural, y: UnlimitedNatural): Boolean;
    function '!=' specializes BaseFunctions::'!='(x: UnlimitedNatural, y: UnlimitedNatural): Boolean;

    function ToString specializes BaseFunctions::ToString(x: UnlimitedNatural): String;
    function ToNatural(x: UnlimitedNatural): Natural;
    function ToUnlimitedNatural(x: String): UnlimitedNatural;
}
```

Natural Functions

```

package NaturalFunctions {
  import ScalarValues::*;

  function '+' specializes IntegerFunctions::'+' (x: Natural, y: Natural[0..1]): Natural;
  function '*' specializes IntegerFunctions::'*' (x: Natural, y: Natural): Natural;
  function '/' specializes IntegerFunctions::'/' (x: Natural, y: Natural): Natural;
  function '%' specializes IntegerFunctions::'%' (x: Natural, y: Natural): Natural;

  function '<' specializes IntegerFunctions::'<', UnlimitedNaturalFunctions::'<'
    (x: Natural, y: Natural): Boolean;
  function '>' specializes IntegerFunctions::'>', UnlimitedNaturalFunctions::'>'
    (x: Natural, y: Natural): Boolean;
  function '<=' specializes IntegerFunctions::'<=', UnlimitedNaturalFunctions::'<='
    (x: Natural, y: Natural): Boolean;
  function '>=' specializes IntegerFunctions::'>=', UnlimitedNaturalFunctions::'>='
    (x: Natural, y: Natural): Boolean;

  function Max specializes IntegerFunctions::Max, UnlimitedNaturalFunctions::Max
    (x: Natural, y: Natural): Natural;

  function '==' specializes IntegerFunctions::'==', UnlimitedNaturalFunctions::'='
    (x: UnlimitedNatural, y: UnlimitedNatural): Boolean;
  function '/=' specializes IntegerFunctions::'!=', UnlimitedNaturalFunctions::'!='
    (x: UnlimitedNatural, y: UnlimitedNatural): Boolean;

  function ToString specializes IntegerFunctions::ToString, UnlimitedNaturalFunctions::ToString
    (x: Natural): String;
  function ToNatural(x: String): Natural;
}

```

Rational Functions

```

package RationalFunctions {
    import ScalarValues::*;

    function Rat(numer: Integer, denum: Integer): Rational;
    function Numer(rat: Rational): Integer;
    function Denom(rat: Rational): Integer;

    function Abs specializes NumericalFunctions::Abs (x: Rational): Rational;
    function '+' specializes NumericalFunctions::'+' (x: Rational, y: Rational[0..1]): Rational;
    function '-' specializes NumericalFunctions::'-' (x: Rational, y: Rational[0..1]): Rational;
    function '*' specializes NumericalFunctions::'*' (x: Rational, y: Rational): Rational;
    function '/' specializes NumericalFunctions::'/' (x: Rational, y: Rational): Rational;
    function '**' specializes NumericalFunctions:: '**' (x: Rational, y: Rational): Rational;
    function '<' specializes NumericalFunctions::'<' (x: Rational, y: Rational): Boolean;
    function '>' specializes NumericalFunctions::'>' (x: Rational, y: Rational): Boolean;
    function '<=' specializes NumericalFunctions::'<=' (x: Rational, y: Rational): Boolean;
    function '>=' specializes NumericalFunctions::'>=' (x: Rational, y: Rational): Boolean;
    function Max specializes NumericalFunctions::Max (x: Rational, y: Rational): Rational;
    function Min specializes NumericalFunctions::Min (x: Rational, y: Rational): Rational;
    function '==' specializes BaseFunctions::'==' (x: Rational, y: Rational): Boolean;
    function '!=' specializes BaseFunctions::'!=' (x: Rational, y: Rational): Boolean;
    function GCD(x: Rational, y: Rational): Integer;
    function Floor(x: Rational): Integer;
    function Round(x: Rational): Integer;

    function ToString specializes BaseFunctions::ToString (x: Rational): String;
    function ToInteger(x: Rational): Integer;
    function ToRational(x: String): Rational;
    function ToReal(x: Rational): Real;
    function ToComplex(x: Rational): Complex;

    function sum specializes ScalarFunctions::sum(collection: Rational[0..*]): Rational;
    function product specializes ScalarFunctions::product(collection: Rational[0..*]): Rational;
}

```

Real Functions

```
package RealFunctions {
    import ScalarValues::*;

    function Abs specializes NumericalFunctions::Abs (x: Real): Real;

    function '+' specializes NumericalFunctions::'+' (x: Real, y: Real[0..1]): Real;
    function '-' specializes NumericalFunctions::'-' (x: Real, y: Real[0..1]): Real;
    function '*' specializes NumericalFunctions::'*' (x: Real, y: Real): Real;
    function '/' specializes NumericalFunctions::'/' (x: Real, y: Real): Real;
    function '**' specializes NumericalFunctions::'**' (x: Real, y: Real): Real;
    function '<' specializes NumericalFunctions::'<' (x: Real, y: Real): Boolean;
    function '>' specializes NumericalFunctions::'>' (x: Real, y: Real): Boolean;
    function '<=' specializes NumericalFunctions::'<=' (x: Real, y: Real): Boolean;
    function '>=' specializes NumericalFunctions::'>=' (x: Real, y: Real): Boolean;

    function Max specializes NumericalFunctions::Max (x: Real, y: Real): Real;
    function Min specializes NumericalFunctions::Min (x: Real, y: Real): Real;

    function '==' specializes BaseFunctions::'==' (x: Real, y: Real): Boolean;
    function '!=' specializes BaseFunctions::'!=' (x: Real, y: Real): Boolean;
    function Sqrt(x: Real): Real;

    function Floor(x: Real): Integer;
    function Round(x: Real): Integer;

    function ToString specializes BaseFunctions::ToString (x: Real): String;
    function ToInteger(x: Real): Integer;
    function ToRational(x: Real): Rational;
    function ToReal(x: String): Real;
    function ToComplex(x: Real): Complex;

    function sum specializes ScalarFunctions::sum (collection: Real[0..*]): Real;
    function product specializes ScalarFunctions::product (collection: Real[0..*]): Real;
}
```

Analysis Domain Library (selected models)

Trade Studies

```
package TradeStudies {  
    import Base::Anything;  
    import ScalarValues::*;  
    import ScalarFunctions::*;  
    abstract calc def ObjectiveFunction(alternative) result : ScalarValue[1];  
    abstract requirement def TradeStudyObjective {  
        subject selectedAlternative : Anything;  
        in ref alternatives : Anything[1..*];  
        in calc fn : ObjectiveFunction;  
        out attribute best : ScalarValue;  
        require constraint { fn(selectedAlternative) == best }  
    }  
    requirement def MinimizeObjective :> TradeStudyObjective {  
        out attribute :>> best = alternatives->minimize {in x; fn(x)},  
    }  
    requirement def MaximizeObjective :> TradeStudyObjective {  
        out attribute :>> best = alternatives->maximize {in x; fn(x)};  
    }  
    abstract analysis def TradeStudy {  
        subject studyAlternatives : Anything[1..*];  
        abstract calc objectiveFunction : ObjectiveFunction;  
        objective tradeStudyObjective : TradeStudyObjective {  
            in ref :>> alternatives = studyAlternatives;  
            in calc :>> fn = objectiveFunction;  
        }  
        return ref selectedAlternative =  
            studyAlternatives->selectOne {in ref a; tradeStudyObjective(a)};  
    }  
}
```

An **ObjectiveFunction** 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 **ObjectiveFunction**.

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 **ObjectiveFunction** such that it satisfies the objective of the **TradeStudy**.

Analysis Tooling Annotations

```
package AnalysisTooling {  
    private import ScalarValues::*;

    attribute def ToolExecution {
        attribute toolName : String;
        attribute uri : String;
    }

    attribute 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)

```
package StateSpaceRepresentation {  
    import ISQ::TimeValue;  
  
    abstract attribute def StateSpace;  
    abstract attribute def Input;  
    abstract attribute def Output;  
  
    abstract calc def NextState  
        (input: Input, stateSpace: StateSpace, timeStep: TimeValue): StateSpace;  
    abstract calc def GetOutput(input: Input, stateSpace: StateSpace): Output;  
  
    abstract attribute def StateSpaceEvent;  
    attribute def ZeroCrossingEvent :> StateSpaceEvent;  
    item def StateSpaceItem;  
  
    abstract action def StateSpaceDynamics {  
        in attribute input: Input;  
        abstract calc nextState: NextState;  
        abstract calc getOutput: GetOutput;  
        attribute stateSpace: StateSpace;  
        out attribute output: Output = getOutput(input, stateSpace);  
    }  
    abstract attribute def StateDerivative;  
    abstract calc def Derivative(input: Input, stateSpace: StateSpace): StateDerivative;  
    abstract calc def Integrate (derivative: Derivative, input: Input,  
        initialState: StateSpace, timeInterval: TimeValue) result: StateSpace;  
    ...  
}
```

StateSpaceDynamics is the simplest form of state space representation, and nextState 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 derivative: Derivative;  
    calc :>> nextState: NextState {  
        calc integrate: Integrate(  
            derivative = ContinuousStateSpaceDynamics::derivative,  
            input = NextState::input,  
            initialState = NextState::stateSpace,  
            timeInterval = NextState::timeStep  
        ) return resultState = result;  
    }  
}  
  
abstract calc def Difference(input: Input, stateSpace: StateSpace) :> StateSpace  
abstract calc def AddDiff(stateSpace: StateSpace, diff: StateSpace) :> StateSpace
```

`DiscreteStateSpaceDynamics` represents discrete behavior. The `difference` needs to return difference of the `stateSpace` for each timestep.

```
abstract action def DiscreteStateSpaceDynamics :> StateSpaceDynamics {  
    abstract calc difference: Difference;  
    calc :>> nextState: NextState {  
        attribute diff: StateSpace = difference(input, stateSpace);  
        calc addDiff: AddDiff(stateSpace, diff) return result;  
    }  
}
```

Quantities and Units Domain Library (selected models)

Quantities

```
package Quantities {
```

```
    abstract attribute def TensorQuantityValue :> Collections::Array {
```

```
        attribute num : ScalarValues::Number[1..*] ordered nonunique :>> elements;
        attribute mRef : UnitsAndScales::TensorMeasurementReference;
        attribute :>> dimensions = mRef::dimensions;
```

```
    }
```

```
    abstract attribute def VectorQuantityValue :> TensorQuantityValue {
        attribute :>> mRef : UnitsAndScales::VectorMeasurementReference;
    }
```

```
    abstract attribute def ScalarQuantityValue :> ScalarQuantityValue {
        attribute :>> mRef : UnitsAndScales::ScalarMeasurementReference;
    }
```

```
    abstract attribute tensorQuantities: TensorQuantityValue[*]
    abstract attribute vectorQuantities: VectorQuantityValue[*]
    abstract attribute scalarQuantities: ScalarQuantityValue[*]
```

```
    alias TensorQuantityValue as QuantityValue;
    alias tensorQuantities as quantity;
```

```
}
```

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.

The value of a tensor quantity is relative to a multi-dimensional *tensor measurement reference*.

A *vector* is a tensor with a single dimension.

A *scalar* is a vector with a single element. Its measurement reference may be a *measurement unit* if the scale is a *ratio scale*.

A *quantity* is a usage of a quantity value to represent a feature of something.

Measurement Reference

A *tensor measurement reference* is defined as an array of *scalar measurement references*, one for each element of the tensor.

```
package UnitsAndScales {
    attribute def TensorMeasurementReference :> Collections::Array {
        attribute longName : ScalarValues::String;
        attribute mrefs : ScalarMeasurementReference[1..*] :>> elements;
        ...
    }
    alias TensorMeasurementReference as MeasurementReference;

    attribute def VectorMeasurementReference
        :> TensorMeasurementReference {
        attribute :>> dimensions[1];
        ...
    }

    attribute def ScalarMeasurementReference
        :> VectorMeasurementReference {
        attribute :>> dimensions[1] = 1;
        attribute scaleValueDefinitions: ScaleValueDefinition[0..*];
        ...
    }
    ...
}
```

Units

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 powerfactor* is a representation of a reference unit raised to an exponent.

```
package UnitsAndScales {  
    ...  
  
    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;  
    }  
    ...  
}
```

International System of Quantities (ISQ)

```
package ISQ {  
    import ISQBase::*;

    ...
}
```

```
package ISQBase {
```

```
    attribute def LengthUnit :> SimpleUnit {...}
```

```
    attribute def LengthValue :> ScalarQuantityValue {  
        attribute redefines num : ScalarValues::Real;  
        attribute redefines mRef : LengthUnit;  
    }
```

```
    attribute length: LengthValue :> quantity;  
    ...  
}
```

```
package ISQSpaceTime {  
    import ISQBase::*;

    ...
}
```

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.

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.

The ISO standard (ISO 80000) is divided into several parts. For example, Part 3 defines units related to space and time.

International System of Units / Système International (SI)

```
package SI {  
    import ISQ::*;

    import SIPrefixes::*;

    attribute g : MassUnit { redefines longName = "gram"; }

    attribute m : LengthUnit { redefines longName = "metre"; }

    attribute kg : MassUnit {  
        attribute redefines name = "kilogram";  
        attribute redefines unitConversion : ConversionByPrefix {  
            attribute redefines prefix = kilo;  
            attribute redefines referenceUnit = g  
        }  
    }  
    attribute s : TimeUnit { redefines longName = "second"; }  
    ...  
  
    attribute N: ForceUnit = kg * m / s ** 2  
    { redefines longName = "newton"; }
```

The *International System of Units* defines base units for the seven abstract ISQ unit types.

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

```
package USCustomaryUnits {  
    import ISQ::*;

    attribute ft : LengthUnit {  
        attribute redefines longName = "foot";  
        attribute redefines unitConversion : ConversionByConvention {  
            attribute redefines referenceUnit = m,  
            attribute redefines conversionFactor = 3048/10000;  
        }  
    }  
    attribute mi : LengthUnit {  
        attribute redefines longName = "mile",  
        attribute redefines unitConversion : ConversionByConvention {  
            attribute redefines referenceUnit = ft,  
            attribute redefines conversionFactor = 5280;  
        }  
    }  
  
    attribute 'mi/hr' : SpeedUnit = mi / hr  
    { redefines longName = "mile per hour"; }  
    alias 'mi/hr' as mph;  
    ...  
}
```

US customary units are defined by conversion from SI units.

An alias for mile per hour.