
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: 2021-11


# Changes in this Release

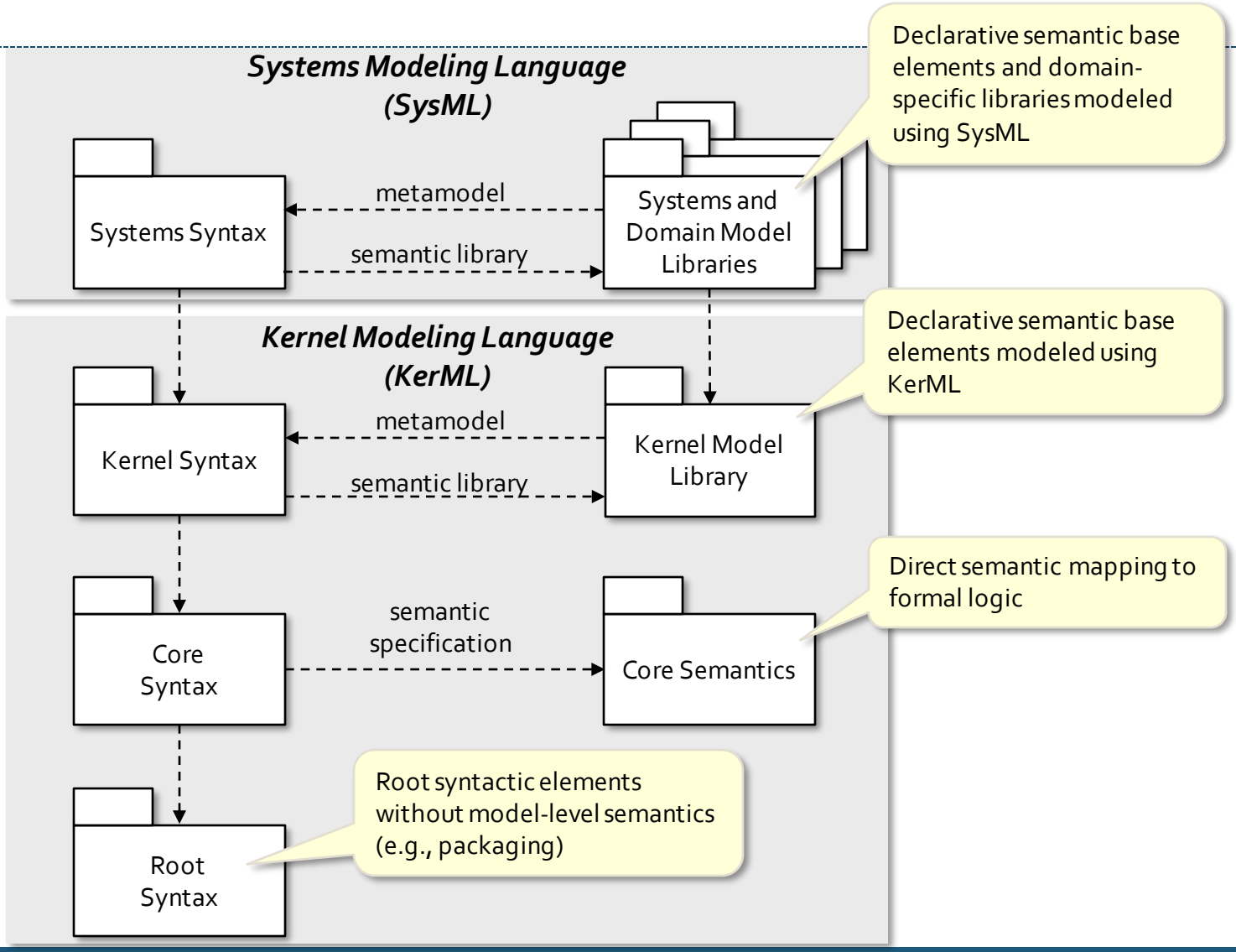
SST

- Changed RiskLevelEnum values from L, H, M to low, medium, high.
- Changed names of TradeStudies::ObjectiveFunction to EvaluationFunction and TradeStudies::TradeStudy::objectiveFunction to objectiveFunction.

*To find slides that have changed recently, search for*
Last changed: 2021-11
*(and similarly for earlier releases).*

# SysML v2 Language Architecture

SST



# Four-Layer Language Architecture

SST

- 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, ItemFow, SuccessionItemFlow
- Systems – Modeling language for systems engineering – Domain libraries
  - AttributeDefinition, EnumerationDefinition, OccurrenceDefinition, ItemDefinition PartDefinition, PortDefinition, ActionDefinition, StateDefinition, ConstraintDefinition, RequirementDefinition, ConcernDefinition, CalculationDefinition, CaseDefinition, AnalysisCaseDefinition, VerificationCaseDefinition , UseCaseDefinition, ViewDefinition, ViewpointDefinition, RenderingDefinition
  - ReferenceUsage, AttributeUsage, EnumerationUsage, OccurrenceUsage, ItemUsage, PartUsage, PortUsage, ActionUsage, StateUsage, ConstraintUsage, RequirementUsage, ConcernUsage, CalculationUsage, CaseUsage, AnalysisCaseUsage, VerificationCaseUsage, UseCaseUsage, ViewUsage, ViewpointUsage, RenderingUsage
  - ConnectionDefinition, ConnectionUsage, InterfaceUsage, InterfaceDefinition, AllocationDefinition, AllocationUsage, BindingConnectionAsUsage, SuccessionAsUsage, FlowConnectionUsage,SucessionFlowConnectionUsage, Dependency

# Packages – Members

SST

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.

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

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

   part def Automobile;

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

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

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

ⓘ 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

SST

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

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

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

  private part def Automobile;

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

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

# Comments

SST

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 Car for Automobile;

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

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

# Documentation

SST

*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 Car for Automobile;
  alias Torque for ISO::TorqueValue;
}
```

# Part and Attribute Definitions

SST

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

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

ⓘ Definitions and usages are also namespaces that allow import.

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

  part eng : Engine;

  ref part driver : Person;
}

attribute def VehicleStatus
  import ScalarValues::*;

  attribute gearSetting : Integer;
  attribute acceleratorPosition : Real;
}

part def Engine;
part def Person;
```

An *attribute usage 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

SST

```
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;
```

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

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

SST

```
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

SST

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

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

part def Cylinder;
```

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

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

A feature can also specify *multiplicity*.

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

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

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

# Enumeration Definitions (1)

SST

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)

SST

```
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.

The enum keyword is optional when declaring enumerated values.

⚠ 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)

SST

```
// 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];
  }
}
```

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

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)

SST

```
// 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

SST

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.

ⓘ 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.

ⓘ The default multiplicity for attributes and items is also `1..1`.

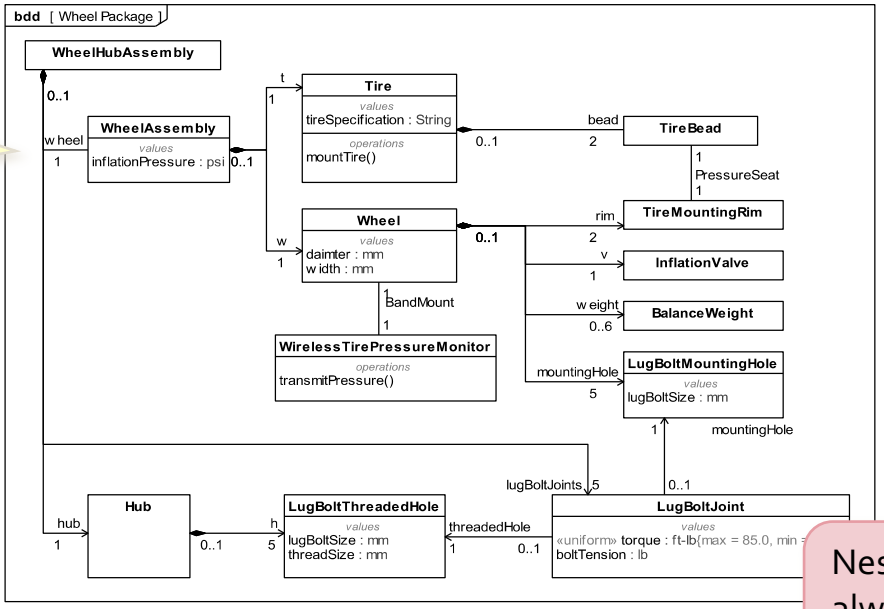
Items may also model continuous materials that are stored in and/or flow between parts of a system.

ⓘ 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.

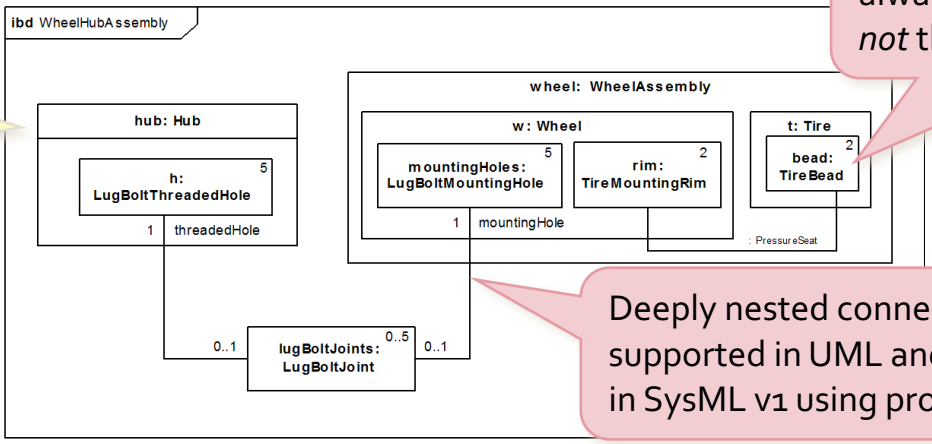
# Connectors
# Example from SysML v1.6 Spec

SST

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)

SST

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.

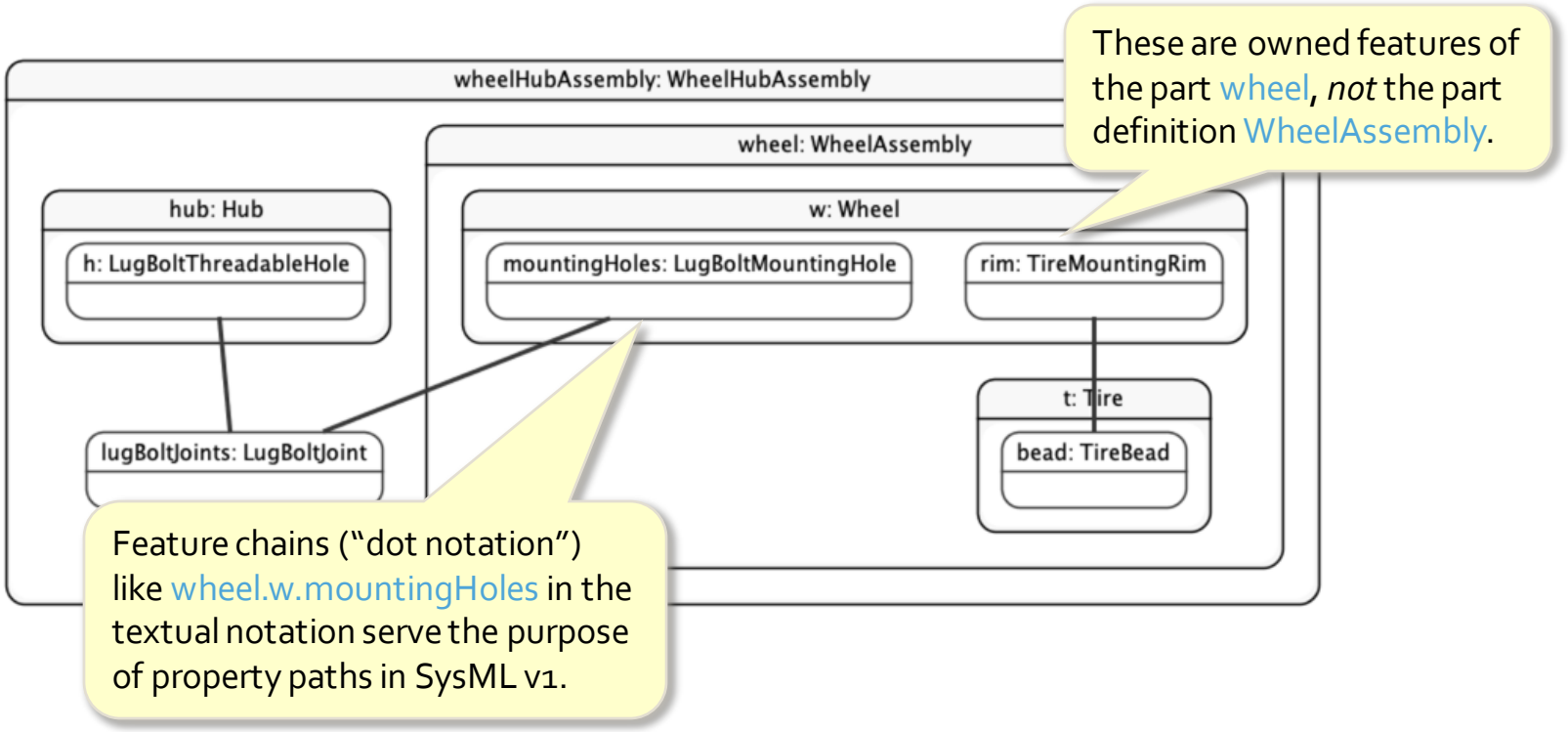
If a connection definition is not specified, a generic connection definition (called Connection) is used.

```
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];
}
```

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

# Connections (2)

SST



# Ports

SST

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

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

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

part def FuelTankAssembly {
  port fuelTankPort : FuelOutPort;
}

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

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

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

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

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 Conjugation

SST

```
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;
}
```

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

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

# Interfaces

SST

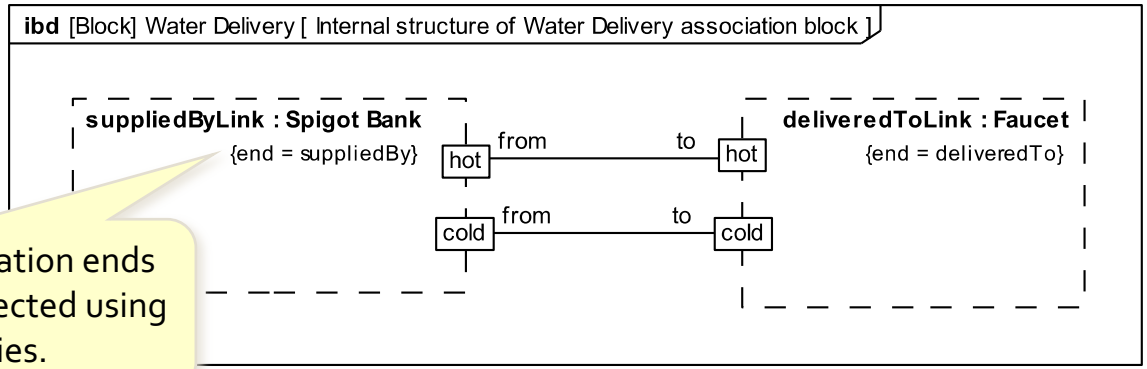
An *interface definition* is a connection definition whose ends are ports.

```
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

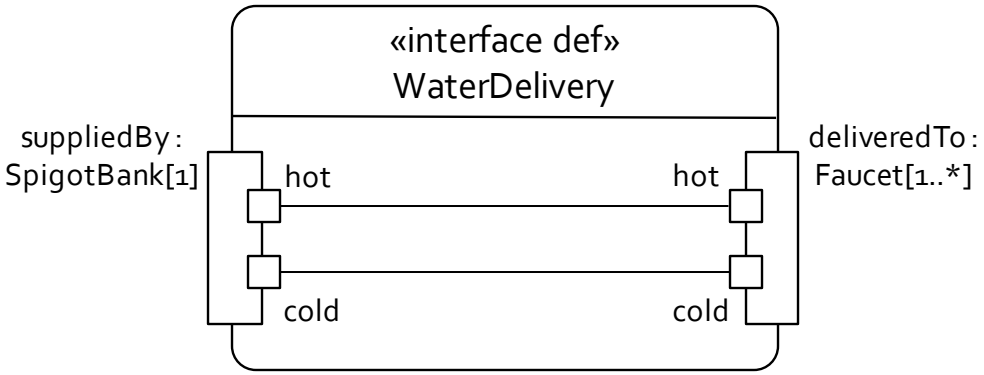
SST




In SysML v1, association ends could only be connected using participant properties.

# Interface Decomposition

SST




```
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)

SST

```
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)

SST

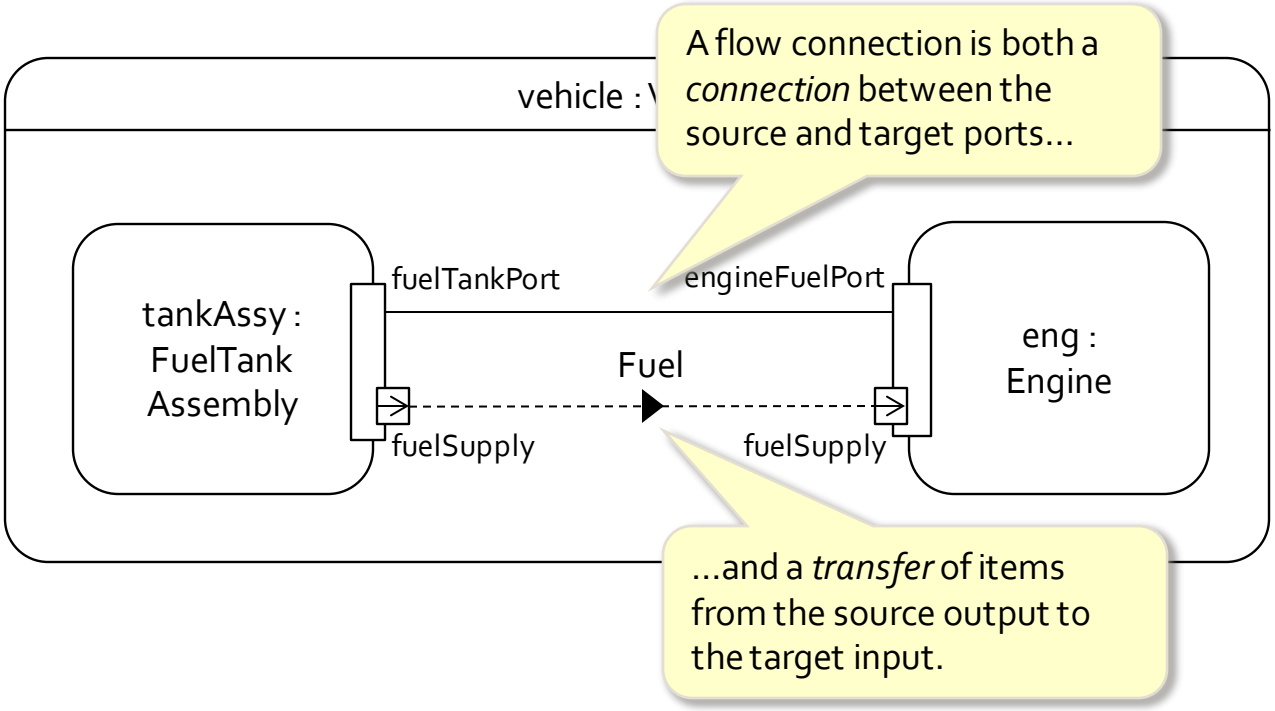
```
part tank : FuelTankAssembly {
  port redefines fuelTankPort {
    out item redefines fuelS
    in item redefines fuelR
  }

  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.

# Flow Connections

SST




ⓘ A flow connection 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 Flow Connections

SST

A *flow connection* is a transfer of something 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;
  part eng : Engine;

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

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

# Streaming Interfaces

SST

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

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

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

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

Flow connections can be defined within an interface definition.

The flows are established when the interface is used.

# Action Definition

SST

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

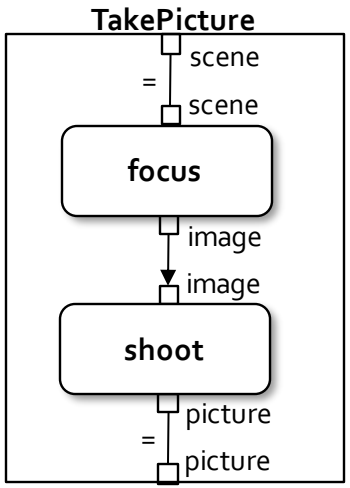
Action definitions may have in, out and inout *parameters* (the default is in).

```
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.

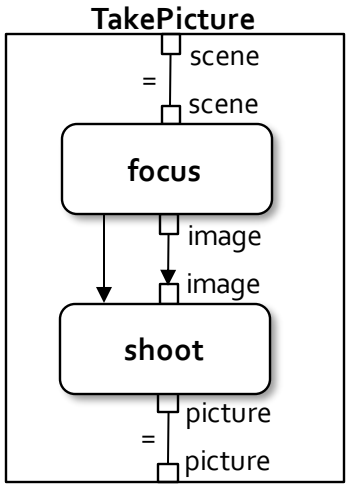
A flow connection can be used to transfer items between actions.

An action has parameters corresponding to its action definition.



# Action Succession (1)

SST



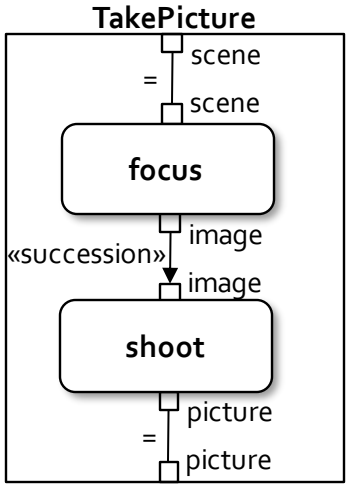

```
action def Focus(in scene : Scene, out image : Image);
action def Shoot(in image : Image, out picture : Picture);

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

  bind focus.scene = scene;

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

  flow focus.image to shoot.image;

  first focus then shoot;

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

  bind shoot.picture = picture;
}
```

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

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

# Action Succession (2)

SST

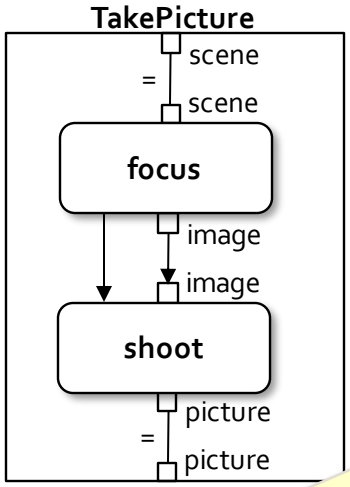



```
action def Focus(in scene : Scene, out image : Image);
action def Shoot(in image : Image, out picture : Picture);

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

  bind focus.scene = scene;

  action focus : Focus (in scene, out imag

  succession flow focus.image to shoot.image;

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

  bind shoot.picture = picture;
}
```

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

# Action Notation Shorthands

SST




```
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 : Pict

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

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

This is the same shorthand for binding used previously.

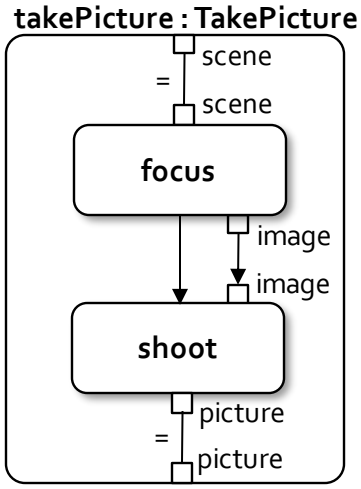
This is a shorthand for a flow connection *into* the parameter.

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

This qualified name refers to the picture parameter of TakePicture, rather than the parameter of shoot with the same name.

# Action Decomposition

SST



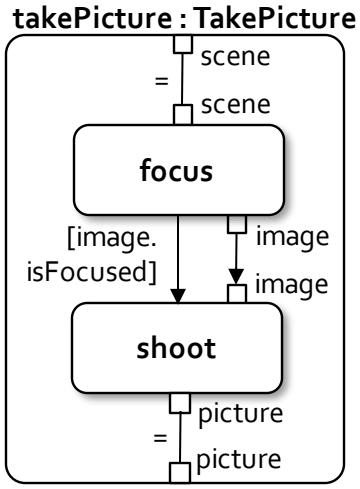

```
action def Focus(in scene : Scene, out image : Image);
action def Shoot(in image : Image, out picture : Picture);
action def TakePicture(in scene : Scene, out picture : Picture);

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

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

  then 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.

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

# Conditional Succession (1)

SST



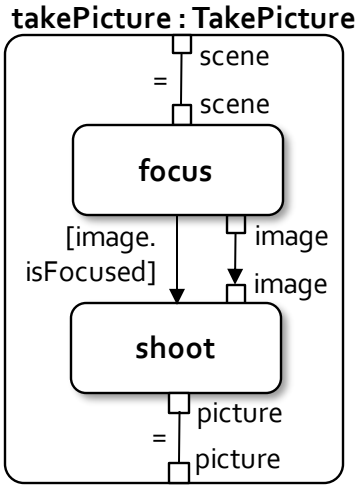

```
action takePicture : TakePicture {
  in item scene;
  out item picture;

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

  first focus
    if focus.image.isFocused then shoot;

  action shoot : Shoot {
    in item image flow 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 (no shorthand).

# Conditional Succession (2)

SST



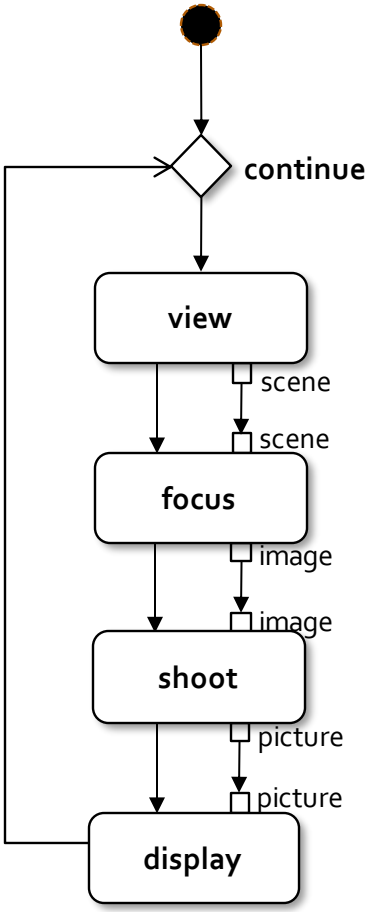

```
action takePicture : TakePicture {
  in item scene;
  out item picture;

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

  if focus.image.isFocused then shoot;

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

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

# Merge Nodes

SST

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



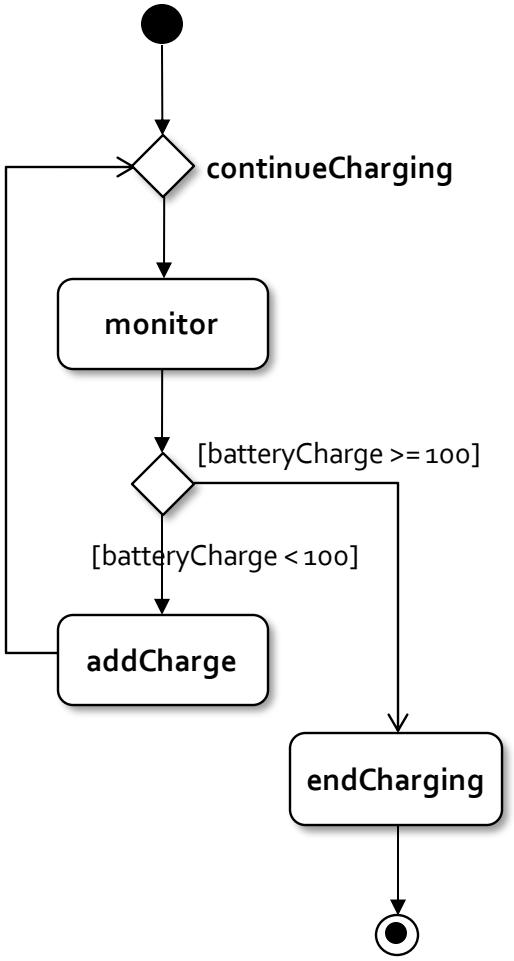

```
action takePicture : TakePicture {
  first start;

  then merge continue;

  then action trigger { out item scene; }

  then action focus : Focus {
    in item scene flow from trigger.scene;
    out item image;
  }

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

  then action display {
    in picture flow 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

SST




```
action def ChargeBattery {
  first start;

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

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

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

  action endCharging : EndCha

  then done;

}
```

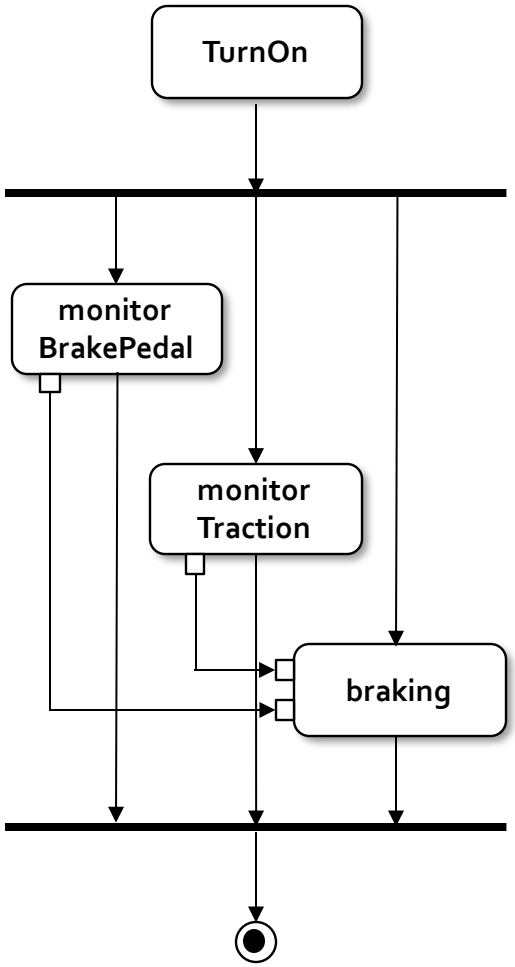
A decision node (decide keyword) chooses *exactly one successor* to happen after it.

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

SST




```
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 flow from
         monitorBrakePedal.brakePressure;
    in modulationFrequency flow 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

SST

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;
  }
}
```

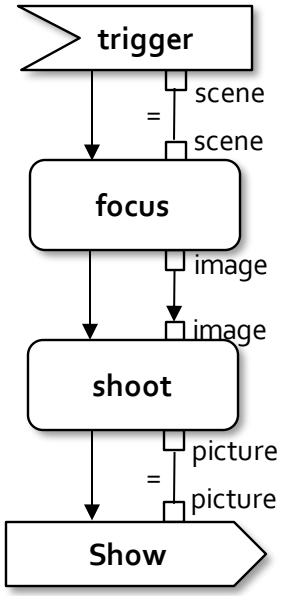
# Asynchronous Messaging

SST

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

This is the *name* of the action.

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




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

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

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

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

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

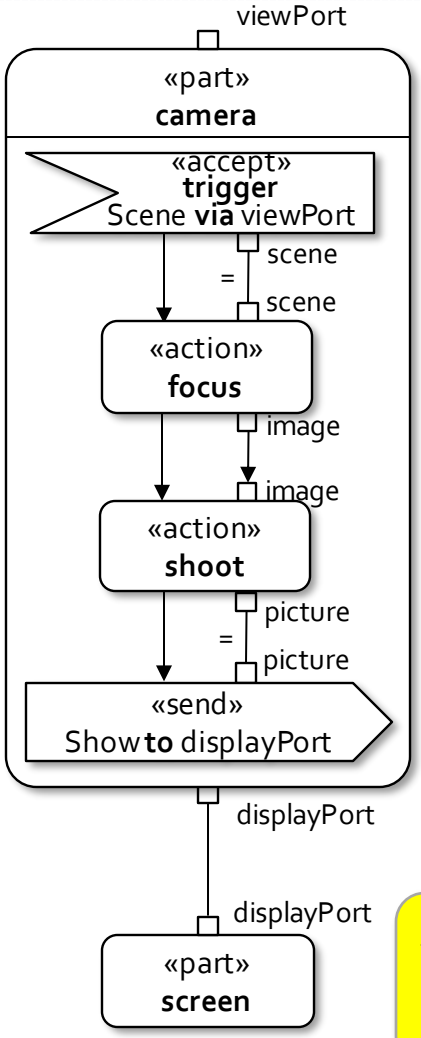
A *send action* is an outgoing transfer of values to a specific target.

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

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

# Asynchronous Messaging through Ports

SST




```
part camera : Camera {
  port viewPort;
  port displayPort;

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

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

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

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

⚠ An asynchronous transfer can propagate over a connection, but that is not currently modeled as a flow.

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

# Opaque Actions

SST

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);
     *     }
     * }
     */
}
```

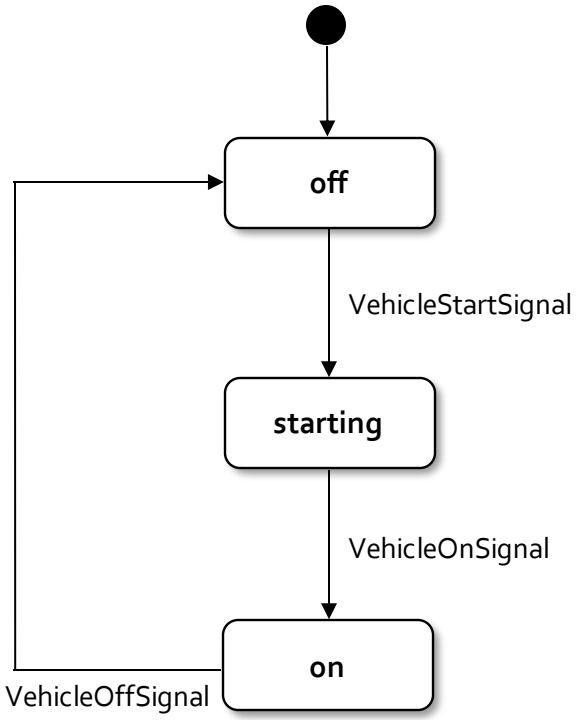
The textual representation body is written using comment syntax. The /*, */ and leading * symbols are *not* included in the body text. Note that support for referencing SysML elements from the body text is tool-specific.

ⓘ 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)

SST

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;
}
```

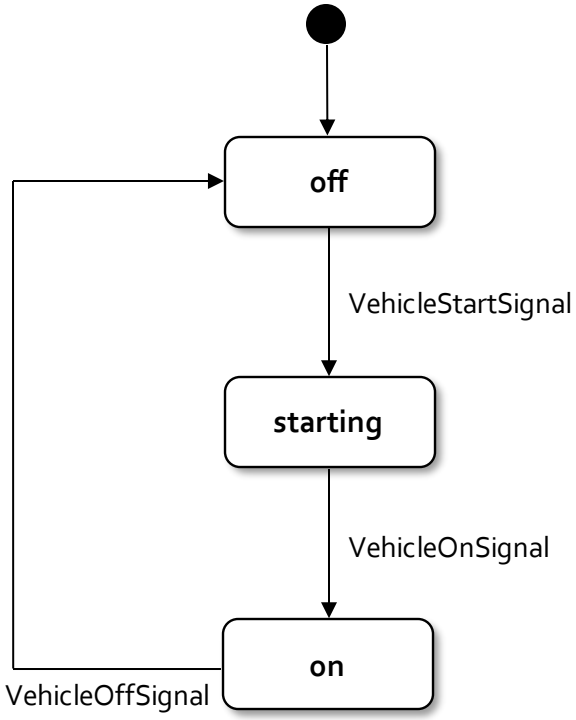
This indicates the the initial state after entry is "off".

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)

SST

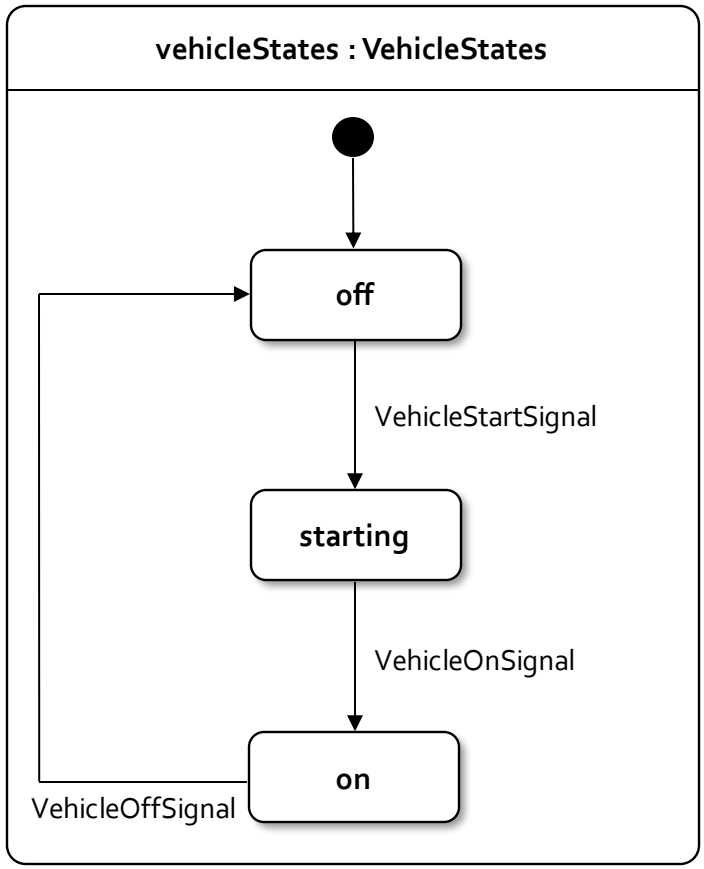



```
state def VehicleStates {
   entry; then off;

   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

SST




```
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;
}
```

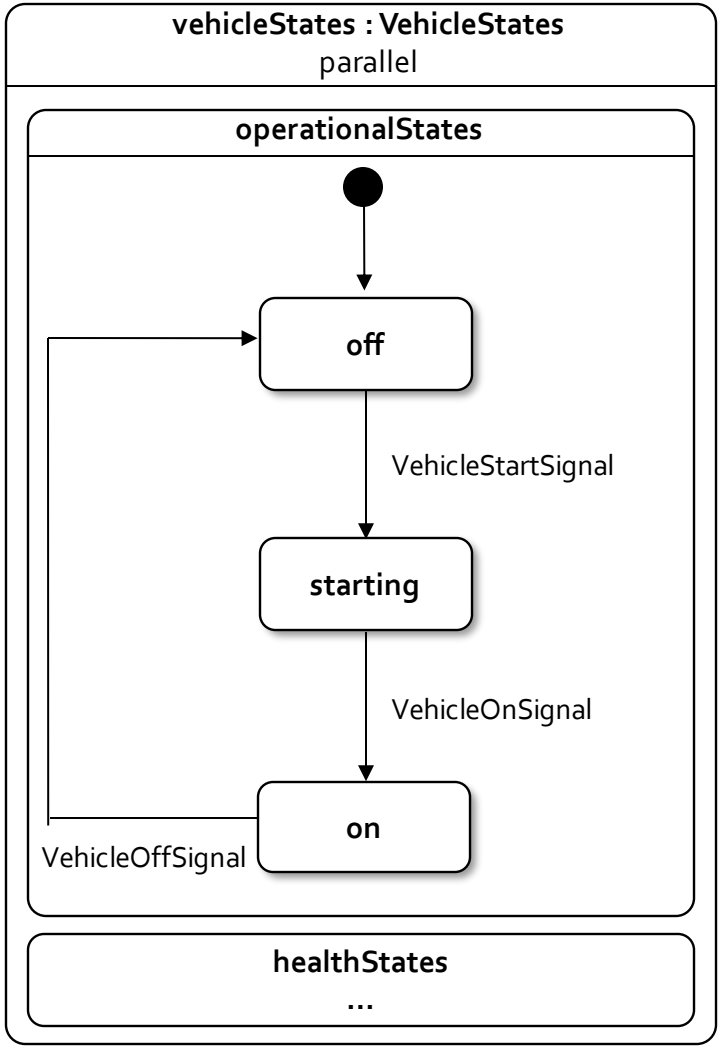
A state can be explicitly declared to be a usage of a state definition.

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

SST



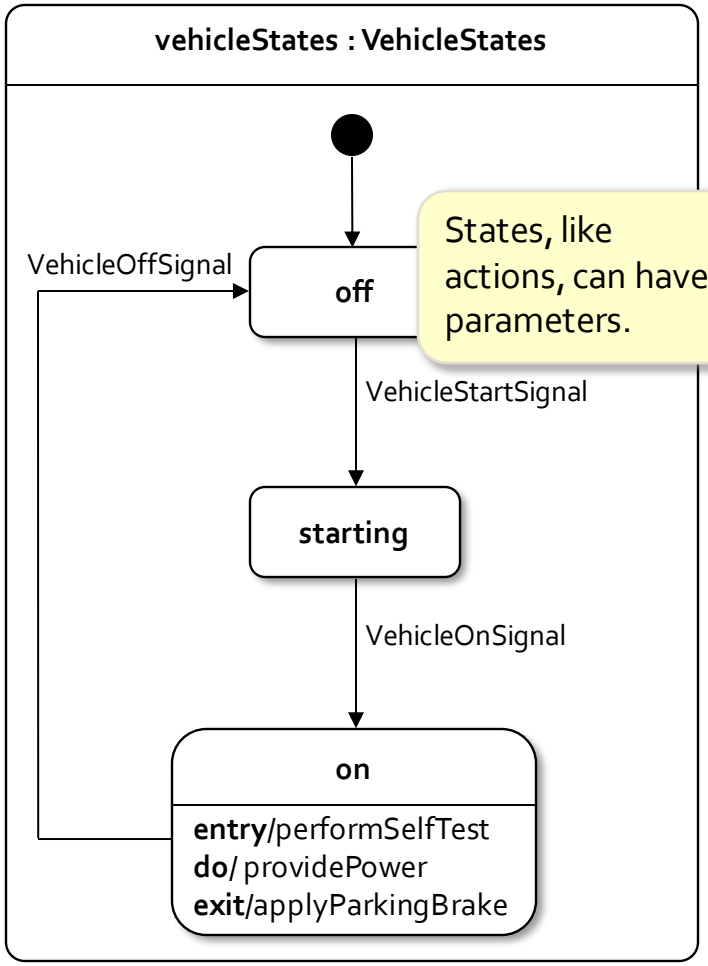

A *parallel state* is one whose nested states are concurrent.

```
state def VehicleStates;

state vehicleStates : VehicleStates parallel {

   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 {
      ...
   }
}
```

⚠ Transitions are not allowed between concurrent states.

# State Entry, Do and Exit Actions

SST




States, like actions, can have parameters.

```
action performSelfTest(vehicle : Vehicle);

state def VehicleStates(operatingVehicle : Vehicle);

state vehicleStates : VehicleStates
   (operatingVehicle : Vehicle) {

   entry; then off;

   state off;
   accept VehicleStartSi
      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;
}
```

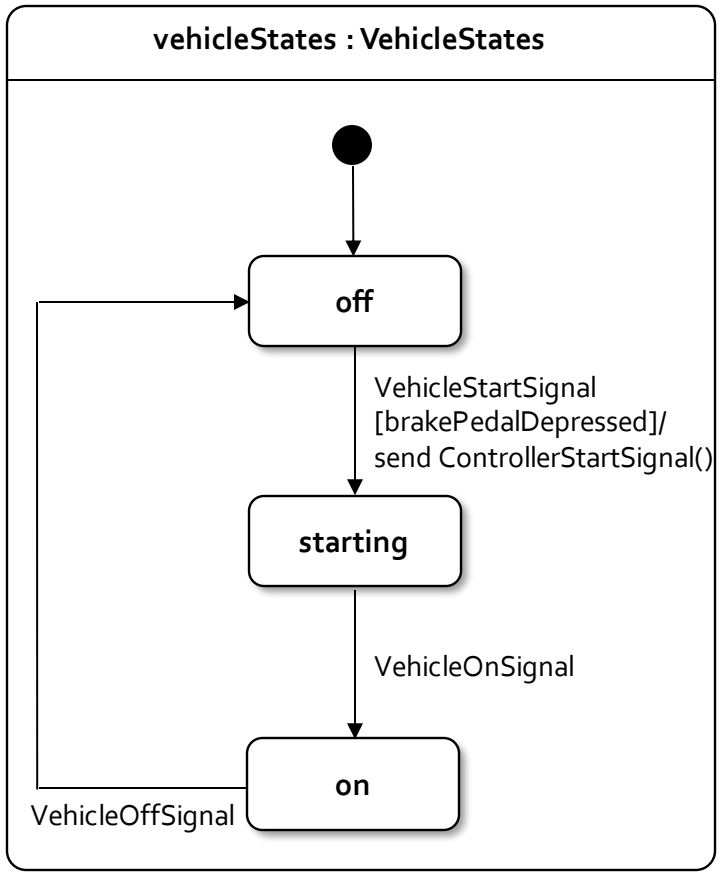
ⓘ An *entry action* is performed on entry to a state, a *do action* while in it, and an *exit action* on exit from it.

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

… or the action can be defined within the state.

# Transition Guards and Effect Actions

SST




```
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

SST

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

```
part vehicle : Vehicle {

  part vehicleController : VehicleController;

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

}
```

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

# Occurrences, Time Slices and Snapshots (1)

SST

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

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

```
attribute def Date;

item def Person;

part def Vehicle {
   timeslice assembly;

   first assembly then delivery;

   snapshot delivery;

   first delivery then ownership;

   timeslice ownership[0..*] ordered;

   snapshot junked = done;
}
```

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

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

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

ⓘ Timeslices and snapshots are *suboccurrences*.

# Occurrences, Time Slices and Snapshots (2)

SST

```
part def Vehicle {
  …

  snapshot delivery {
    attribute deliveryDate : Date;
  }

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

    ref item owner : Person[1];

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

  snapshot junked = done;
}
```

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

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

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

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

# Event Occurrences

SST

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

Event occurrences can be sequenced as usual using succession.

```
part driver : Driver {
  event occurrence setSpeedSent;
}

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

  part speedometer : Speedometer {
    event occurrence sensedSpeedSent;
  }

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

# Messages (1)

SST

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

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

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

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

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

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

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

  first setSpeedMessage then sensedSpeedMessage;
}
```

# Messages (2)

SST

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

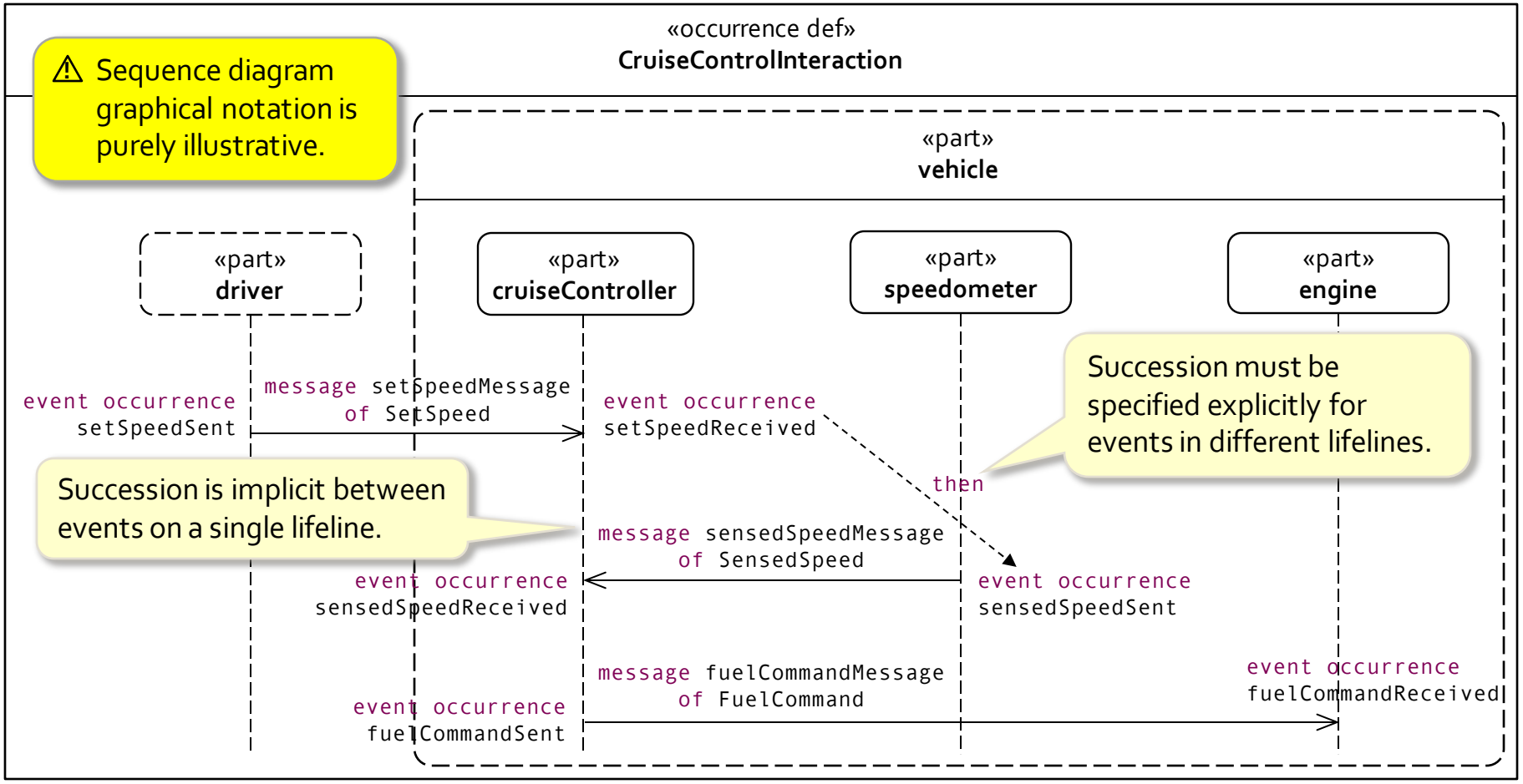
Each message has a source and target, even if it is not explicitly identified in the message declaration.

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

  ref part vehicle : Vehicle {
    part cruiseController : CruiseController {
      event setSpeedMessage::target;
      then event sensedSpeedMessage::target;
      then event fuelCommandMessage::source;
    }
    part speedometer : Speedometer {
      event sensedSpeedMessage::source;
    }
    part engine : Engine {
      event fuelCommandMessage::target;
    }
  }

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

# Interactions

SST

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



# Interaction Realization by Actions (1)

SST

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

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

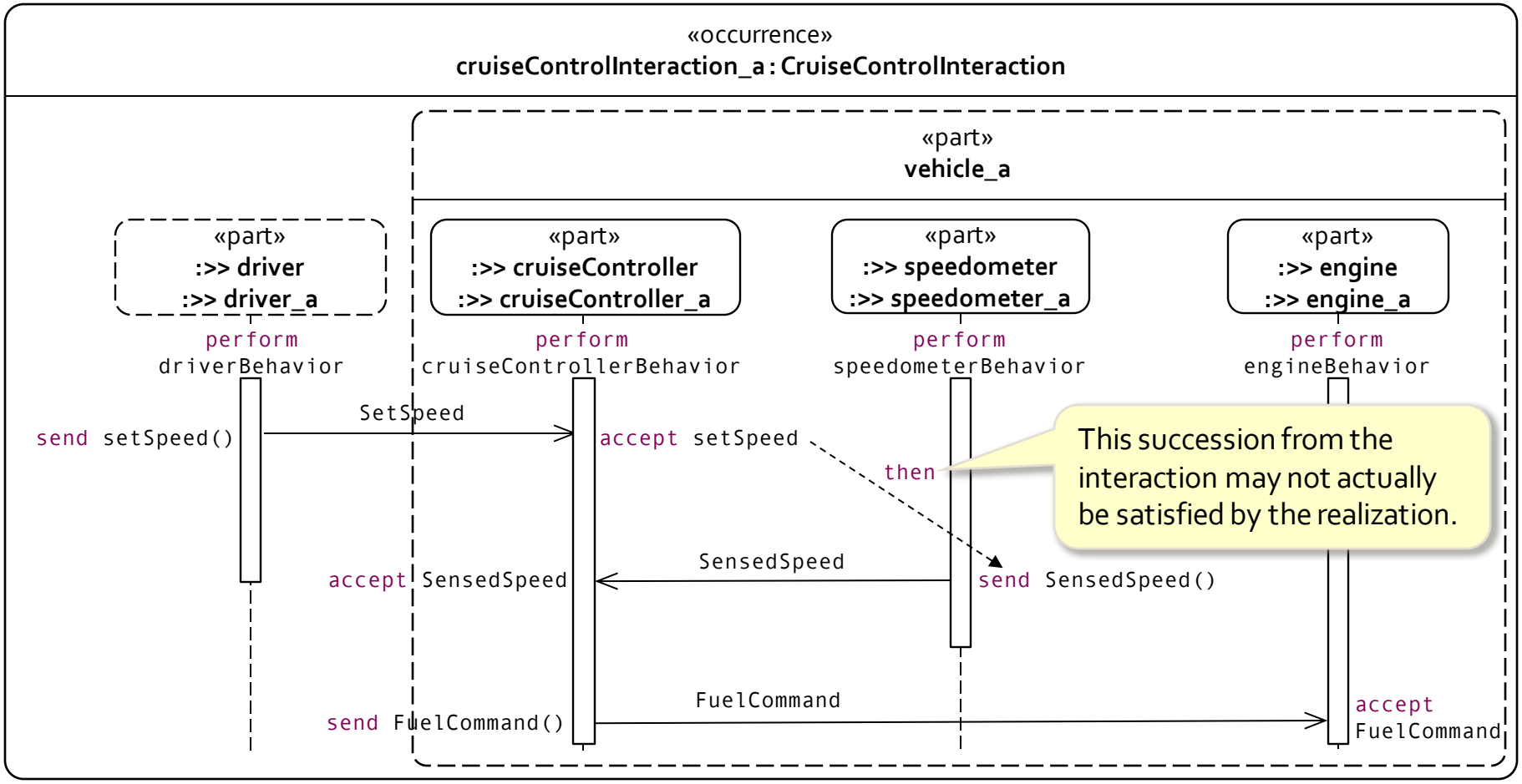
Messages are realized by the implicit transfers between send actions and corresponding accept actions.

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

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

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

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

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

# Interaction Realization by Actions (2)

SST



# Interaction Realization by Flows (1)

SST

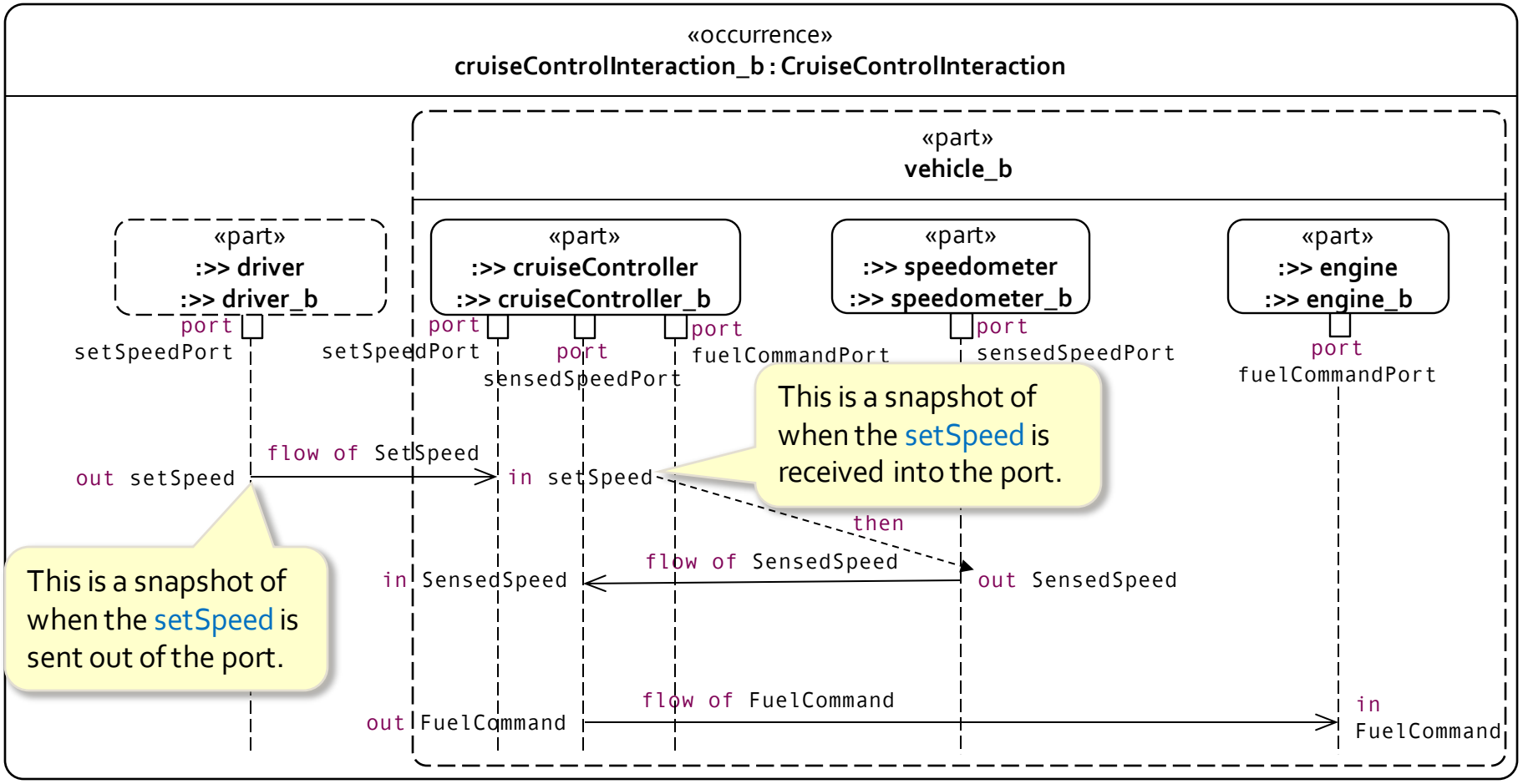
```
part driver_b : Driver {
   port setSpeedPort {
      out setSpeed : SetSpeed;
   }

nterface driver_b.setSpeedPort to vehicle_b.setSpeedPort {
   flow of SetSpeed from driver_b.setSpeedPort.setSpeed
      to vehicle_b.setSpeedPort.setSpeed;
}
part vehicle_b : Vehicle {
   port setSpeedPort {
      in setSpeed : SetSpeed;
   }
   part cruiseController_b : CruiseController {
      port setSpeedPort {
         in setSpeed : SetSpeed = vehicle_b.setSpeedPort.setSpeed;
      }
      port sensedSpeedPort {
         in sensedSpeed : SetSpeed;
      }
      …
   }
   flow of SensedSpeed from speedometer_b.sensedSpeedPort.sensedSpeed
     to cruiseController_b.sensedSpeedPort.sensedSpeed;
   part speedometer_b : Speedometer {
      port sensedSpeedPort {
         out sensedSpeed : SensedSpeed;
      }
   }
…
```

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

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

Messages are realized by explicit flows between features of ports.

# Interaction Realization by Flows (2)



# Individuals and Snapshots (1)

SST

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

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

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

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

individual item context : PhysicalContext {

   snapshot part 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.

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

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

# Individuals and Snapshots (2)

SST

```
individual item context : PhysicalContext {

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

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

  first vehicle_1_t0 then vehicle_1_t1;
}
```

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

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

# Individuals and Roles

SST

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

individual part def Wheel_1 :> Wheel;

individual part vehicle_1 : Vehicle_1 {

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

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

By default, these are snapshots of the containing individual.

During the first snapshot of Vehicle_1, Wheel_1 has the role of the leftFrontWheel.

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

# Individuals and Time Slices

SST

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

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

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

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

individual part : Vehicle_1 {

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

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

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

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

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

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

# Expressions and Feature Values

SST

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

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

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

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

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

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

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

# Car Mass Rollup Example (1)

SST

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

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


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

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

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

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

# Default Values

SST

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

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

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

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

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

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

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

# Car Mass Rollup Example (2)

SST

```
import ScalarValues::*;
import MassRollup2::*;

part def CarPart :> MassedThing {
   attribute serialNumber : String;
}
part car: CarPart :> compositeThing {
   attribute vin :>> serialNumber;
   part carParts : CarPart[*] :>> subcomponents;
   part engine :> carParts { … }
   part transmission :> carParts { … }
}


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

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

The default of totalMass to simpleMass is inherited.

The default for totalMass is overridden as specified in compositeThing.

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

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

# Calculation Definitions

SST

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

*Calculation parameters* are similar to the parameters on actions.

A calculation has a single *return result*.

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

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

   whlpwr - drag - friction
}

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

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;
```

The calculation can include the computation of intermediate values.

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

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

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

# Calculation Usages (1)

SST

```
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;
   }
}
```

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

ⓘ Note that the return result is *outside* the parentheses. The return keyword is optional.

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)

SST

```
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 : MassVa

   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)

SST

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

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

   sum(partMasses) <= massLimit
}

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

   attribute chassisMass : MassValu

   part engine : Engine {
      attribute mass : MassValue;
   }

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

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

*Constraint parameters* are similar to the parameters on actions.

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

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

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

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

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

# Constraint Definitions (2)

SST

```
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)

SST

```
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)

SST

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

SST

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

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

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

constraint def AveragedDynamics (
   mass: MassValue,
   initialSpeed : SpeedValue,
   finalSpeed : SpeedValue,
   deltaT : TimeValue,
   force : ForceValue ) {


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

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

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

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

# Analytical Constraints

SST

```
constraint def StraightLineDynamicsEquations(
   p : PowerValue, m : MassValue, dt : TimeValue,
   x_i : LengthValue, v_i : SpeedValue,
   x_f : LengthValue, v_f : SpeedValue,
   a : AccelerationValue
) {
   attribute v_avg : SpeedValue = (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 : SpeedValue,
   out x_out : LengthValue, out v_out : SpeedValue,
   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.

ⓘ Note the use of the calculation definitions defined earlier.

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

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

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

# Requirement Definitions (1)

SST

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)

SST

```
part def Vehicle {
   attribute dryMass: MassValue;
   attribute fuelMass: MassV
   attribute fuelFullMass: Ma
   …
}

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

   subject vehicle : Vehicle;


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

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

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

ⓘ Actually, any identifiable element can have a human id, not just requirements.

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)

SST

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

```
port def ClutchPort;
action def GenerateTorque;

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

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

# Requirement Usages

SST

A requirement may optionally have its own human id.

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

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

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

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

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

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

# Requirement Groups

SST

A requirement may also be used to group other requirements.

ⓘ 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.

Requirements can be grouped by reference…

…or by composition.

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

# Requirement Satisfaction

SST

```
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)

SST

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;

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

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

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

   …
}
```

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

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

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

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

# Analysis Case Definitions (2)

SST

```
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 {in i : Positive;
            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 *steps* of an analysis case are actions that, together, compute the analysis result.

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

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

# Analysis Case Usages

SST

```
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.fuel
   }

   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

SST

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

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

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

   calc :>> evaluationFunction {
      in part 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;
}
```

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

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

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

# Verification Case Definitions (1)

SST

```
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.

# Verification Case Definitions (2)

SST

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 argument 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)

SST

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)

SST

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

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

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

individual testSystem : TestSystem :> massVerificationSystem {

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

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

The test on the individual TestVehicle1 should pass.

The test on the individual TestVehicle2 should fail.

# Use Case Definition

SST

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

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

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

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

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

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

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

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

# Use Case Usage

SST

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

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

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

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

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

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

use case 'add fuel' {
   subject vehicle : Vehicle;
   actor fueler : Person;
   actor 'fuel station' : 'Fuel Station';
}
```

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

# Variation Definitions

SST

```
attribute def Diameter :> Real;

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

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

// Variability model

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

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

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

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

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

# Variation Usages

SST

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.

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

```
abstract part vehicleFamily : Vehicle {

   part engine : EngineChoices[1];

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

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

}
```

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

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

# Variation Configuration

SST

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

SST

```
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;
}
```

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

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

A dependency can have multiple clients and/or suppliers.

# Allocation

SST

```
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;
   }
}
```

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

ⓘ 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

SST

⚠ 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)

SST

ⓘ 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;
```

ⓘ 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)

SST

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

ⓘ The AnalysisTooling package is part of the Analysis domain library.

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

```
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";}}
}
```

The @ symbol is equivalent to the metadata keyword.

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

# Element Import Filtering (1)

SST

```
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.

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

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

package 'Mandatory Safety Features' {
   import vehicle::**;
   filter @Safety and Safety::isMandatory;
}
```

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)

SST

```
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;}}
   }
}
```

ⓘ The filter keyword is only allowed in a package, but a filtered import can be used anyplace an import is allowed.

Filter conditions can also be combined with the import itself.

```
package 'Safety Feature' {
   import vehicle::**[@Safety];
}

package 'Mandatory Safety Features' {
   import vehicle::**
      [@Safety and 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.

# Stakeholders and Concerns

SST

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

concern 'system breakdown' {
  doc /*
   * To ensure that a system covers all its required capabilities,
   * it is necessary to understand how it is broken down into
   * subsystems and components that provide those capabilities.
   */
  stakeholder se: 'systems engineer';
  stakeholder ivv: 'IV&V';
}

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

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

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

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

# Viewpoints

SST

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

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

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

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

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

# Views (1)

SST

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

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

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

  filter @SysML::PartUsage;
}

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

  expose vehicle::**;

  render asTreeDiagram;
}
```

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

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

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

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

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

# Views (2)

SST

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

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

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

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

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

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

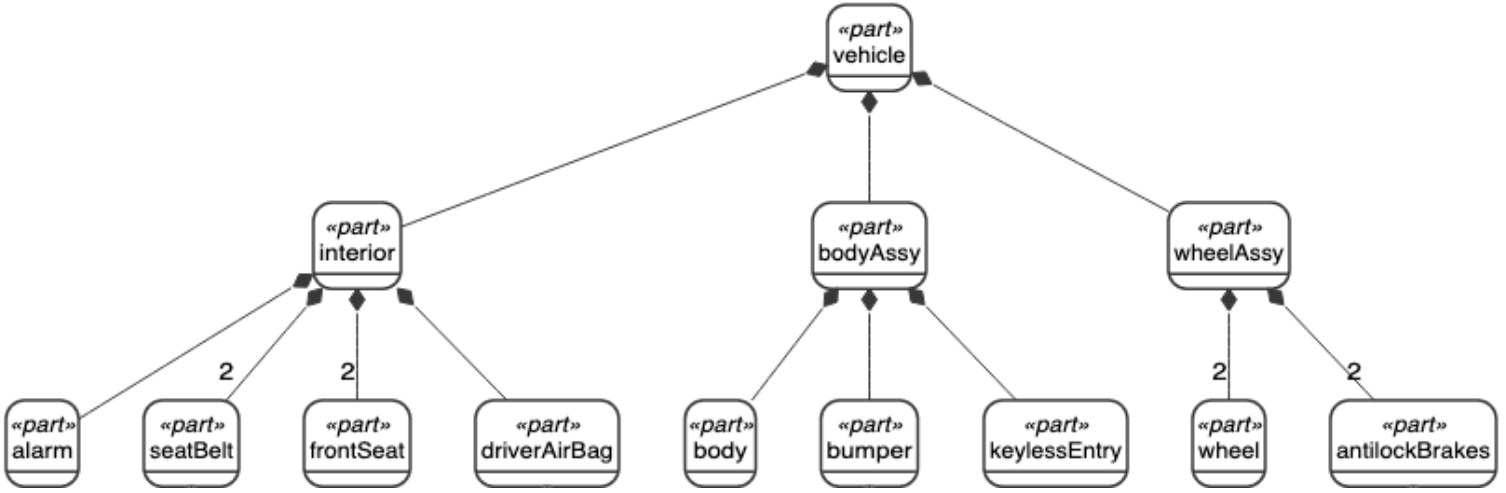
Views can have subviews.

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

# Example View Rendering

SST

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



# Kernel Model Library (selected models)

# ScalarValues

SST

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

  abstract datatype ScalarValue specializes Value;
  datatype Boolean specializes ScalarValue;
  datatype String specializes ScalarValue;
  abstract datatype NumericalValue specializes ScalarValue;

  abstract datatype Number specializes NumericalValue;
  datatype Complex specializes Number;
  datatype Real specializes Complex;
  datatype Rational specializes Real;
  datatype Integer specializes Rational;
  datatype Natural specializes Integer;
  datatype Positive specializes Natural;
}
```

# Base Functions

SST

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

  abstract function '=='(x: Anything[0..1], y: Anything[0..1]): Boolean[1];
  function '!='(x: Anything[0..1], y: Anything[0..1]): Boolean[1] {
    !(x == y)
  }

  function ToString(x: Anything[0..1]): String[1];

  …
}
```

# Sequence Functions

SST

```
package SequenceFunctions {
  private import Base::Anything;
  private import ScalarValues::*;

  function equals
    (x: Anything[0..*] ordered nonunique,
     y: Anything[0..*] ordered nonunique): Boolean[1];
  function size(seq: Anything[0..*] ordered nonunique): Natural[1];
  function isEmpty(seq: Anything[0..*] ordered nonunique): Boolean[1];
  function notEmpty(seq: Anything[0..*] ordered nonunique): Boolean[1];
  function includes(
    seq: Anything[0..*] ordered nonunique,
    value: Anything[1]): Boolean[1];
  function excludes(
    seq: Anything[0..*] ordered nonunique,
    value: Anything[1]): Boolean[1];

  function head(seq: Anything[0..*] ordered nonunique): Anything[0..1];
  function tail(seq: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique;
  function last(seq: Anything[0..*]): Anything[0..1];
  …
}
```

# Data Functions

SST

```
package DataFunctions {
  private import Base::DataValue;
  private import ScalarValues::Boolean;

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

ⓘ Generalizes previous ScalarFunctions package.

# Boolean Functions

SST

```
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

SST

```
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

SST

```
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

SST

```
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

SST

```
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

SST

```
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;
}
```

# Metadata Domain Library

# Risk Metadata

SST

General risk level in terms of probability and impact.

Standard low, medium and high risk levels.

To be used to annotate model elements as to their risk level in typical risk areas.

```
package RiskMetadata {
   import ScalarValues::Real;
   attribute def Level :> Real {
      private attribute level : Level :>> self;
      assert constraint { level >= 0.0 && level <= 1.0 }
   }
   enum def LevelEnum :> Level {
      low = 0.25;
      medium = 0.50;
      high = 0.75;
   }
   attribute def RiskLevel {
      attribute probability : Level;
      attribute impact : Level [0..1];
   }
   enum def RiskLevelEnum :> RiskLevel {
      low = RiskLevel(probability = LevelEnum::L);
      medium = RiskLevel(probability = LevelEnum::M);
      high = RiskLevel(probability = LevelEnum::H);
   }
   attribute def Risk {
      attribute totalRisk : RiskLevel [0..1];
      attribute technicalRisk : RiskLevel [0..1];
      attribute scheduleRisk : RiskLevel [0..1];
      attribute costRisk : RiskLevel [0..1];
   }
}
```

# Modeling Metadata

SST

```
package ModelingMetadata {
   private import Base::Anything;
   private import ScalarValues::String;
   private import RiskMetadata::Risk;

   enum def StatusKind {
      open;
      tbd; // To be determined
      tbr; // To be resolved
      tbc; // To be confirmed
      done;
      closed;
   }
   attribute def StatusInfo {
      attribute originator : String [0..1];
      attribute owner : String [0..1];
      attribute status : StatusKind;
      attribute risk : Risk [0..1];
   }
   attribute def Rationale {
      attribute text : String;
      ref explanation : Anything [0..1];
   }
   attribute def Issue {
      attribute text : String;
   }
}
```

To be used to annotate model elements with status information.

To be used to give a rationale for a model element.

Generic issue annotation.

This is an optional reference to a feature that further explains this rationale (e.g., a trade study analysis).

# Analysis Domain Library

# Trade Studies

SST

```
package TradeStudies {
   import Base::Anything;
   import ScalarValues::*;
   import ScalarFunctions::*;
   abstract calc def EvaluationFunction(alternative) result : ScalarValue[1];
   abstract requirement def TradeStudyObjective {
      subject selectedAlternative : Anything;
      in ref alternatives : Anything[1..*];
      in calc fn : EvaluationFunction;
      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 evaluationFunction : EvaluationFunction
      objective tradeStudyObjective : TradeStudyObjective {
         in ref :>> alternatives = studyAlternatives;
         in calc :>> fn = evaluationFunction;
      }
      return ref selectedAlternative =
         studyAlternatives->selectOne {in ref a; tradeStudyObjective(a)};
}
```

An EvaluationFunction is a calculation that evaluates a TradeStudy alternative.

A TradeStudyObjective is the base definition for the objective of a TradeStudy, requiring the selectedAlternative to have best evaluation according to a given EvaluationFunction.

A TradeStudy is an analysis case whose subject is a set of alternatives and whose result is a selection of one of those alternatives, based on a given EvaluationFunction such that it satisfies the objective of the TradeStudy.

# Analysis Tooling Annotations

SST

```
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)

SST

```
package StateSpaceRepresentation {
   import ISQ::DurationValue;
   import Quantities::VectorQuantityValue;
   import VectorCalculations::*;

   abstract attribute def StateSpace :> VectorQuantityValue;
   abstract attribute def Input :> VectorQuantityValue;
   abstract attribute def Output :> VectorQuantityValue;
   abstract calc def GetNextState
      (input: Input, stateSpace: StateSpace, timeStep: DurationValue): StateSpace;
   abstract calc def GetOutput(input: Input, stateSpace: StateSpace): Output;
   abstract action def StateSpaceEventDef;
   action def ZeroCrossingEventDef :> StateSpaceEventDef;
   item def StateSpaceItem;

   abstract action def StateSpaceDynamics {
      in attribute input: Input;
      abstract calc getNextState: GetNextState;
      abstract calc getOutput: GetOutput;
      attribute stateSpace: StateSpace;
      out attribute output: Output = getOutput(input, stateSpace);
   }
   abstract attribute def StateDerivative :> VectorQuantityValue { … }
   abstract calc def GetDerivative(input: Input, stateSpace: StateSpace): StateDerivative;
   abstract calc def Integrate (getDerivative: GetDerivative, input: Input,
      initialState: StateSpace, timeInterval: DurationValue) result: StateSpace;
   …
```

StateSpaceDynamics is the simplest form of state space representation, and getNextState directly computes the stateSpace of the next timestep.

# State Space Representation (2)

SST

```
   …

   abstract action def ContinuousStateSpaceDynamics :> StateSpaceDynamics {
      abstract calc getDerivative: GetDerivative;
      calc :>> getNextState: GetNextState {
         calc integrate: Integrate(
            derivative = ContinuousStateSpaceDynamics::getDerivative,
            input = GetNextState::input,
            initialState = GetNextState::stateSpace,
            timeInterval = GetNextState::timeStep
         ) return resultState = result;
      }
      event occurrence zeroCrossingEvents[0..*] : ZeroCr
   }

   abstract calc def GetDifference(input: Input, sta

   abstract action def DiscreteStateSpaceDynamics :> StateSpaceDynamics {
      abstract calc getifference: GetDifference;
      calc :>> getNextState: GetNextState {
         attribute diff: StateSpace = getDifference(input, stateSpace);
         stateSpace + diff
      }
   }
}
```

ContinuousStateSpaceDynamics represents continuous behavior. The derivative needs to return a time derivative of stateSpace, i.e. dx/dt.

DiscreteStateSpaceDynamics represents discrete behavior. getDifference returns the difference of the stateSpace for each timestep.

# Sampled Functions

SST

SamplePair is a key-value pair of a domain-value and a range-value, used as a sample element in SampledFunction.

SampleFunction is a variable-size, ordered collection of SamplePair elements representing a discretely sampled mathematical function.

Domain and Range return all the domain values and range values of a sampled function.

Sample a calculation on given domain values to create a SampledFunction.

Interpolate a SampledFunction to compute a result for a given domain value.

The function is constrained to be strictly increasing or strictly decreasing.

```
package SampledFunctions {
  …

  attribute def SamplePair :> KeyValuePair {
    attribute domainValue :>> key;
    attribute rangeValue :>> val;
  }

  attribute def SampledFunction :> OrderedMap {
    attribute samples: SamplePair[0..*] ordered :>> elements;
    assert constraint { … }
  }

  calc def Domain(fn : SampledFunction)
    return Anything[0..*] = fn.samples.domainValue;
  calc def Range(fn : SampledFunction) :
    return Anything[0..*] = fn.samples.rangeValue;

  calc def Sample
    (in calc calculation (x), in attribute domainValues [0..*])
    return sampling { … }

  calc def Interpolate
    (attribute fn : SampledFunction, attribute value)
    return attribute result;
  calc interpolateLinear : Interpolate { … }
}
```

# Quantities and Units Domain Library (selected models)

# Quantities

SST

```
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

SST

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

SST

A *measurement unit* is a measurement scale defined as a sequence of unit power factors.

A *simple unit* is a measurement unit with no power factor dependencies on other units.

A *derived unit* is any unit that is not simple.

A *unit power factor* is a representation of a reference unit raised to an exponent.

```
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)

SST

```
package ISQ {
   import ISQBase::*;
   import ISQSpaceTime::*;
   …
}
```

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.

```
package ISQBase {

   attribute def LengthUnit :> SimpleUnit {…}

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

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

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.

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

The ISQ 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)

SST

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

  attribute g : MassUnit { redefines longName = "gr    , }

  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

SST

```
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.

# Vector Calculations

SST

```
package VectorCalculations {
  import ScalarValues::*;
  import Quantities::*;

  calc def '+' specializes NumericalFunctions::'+'
    (x: VectorQuantityValue, y: VectorQuantityValue):
    VectorQuantityValue;

  calc def '-' specializes NumericalFunctions::'-'
    (x: VectorQuantityValue, y: VectorQuantityValue):
    VectorQuantityValue;

  calc def scalarVectorMult specializes NumericalFunctions::'*'
    (x: Real, y: VectorQuantityValue): VectorQuantityValue;

  calc def vectorScalarMult specializes NumericalFunctions::'*'
    (x: VectorQuantityValue, y: Real): VectorQuantityValue;

alias scalarVectorMult as '*';
}
```

# Time (1)

```
package Time {
   attribute def TimeScale :> IntervalScale {
      attribute :>> unit: DurationUnit[1];
      attribute definitionalEpoch: DefinitionalQuantityValue[1];
      attribute :>> definitionalQuantityValues = definitionalEpoch;
   }

   attribute def TimeInstantValue :> ScalarQuantityValue {
      attribute :>> num: Real[1];
      attribute :>> mRef: TimeScale[1];
   }
   attribute timeInstant: TimeInstantValue :> scalarQuantities;

   abstract attribute def DateTime :> TimeInstantValue;
   abstract attribute def Date :> TimeInstantValue;
   abstract attribute def TimeOfDay :> TimeInstantValue;

   attribute UTC: TimeScale {
      attribute :>> longName = "Coordinated Universal Time";
      attribute :>> unit = SI::s;
      attribute :>> definitionalEpoch: DefinitionalQuantityValue {
         :>> num = 0;
         :>> definition = "UTC epoch at 1 January 1958 at 0 hour 0 minute 0 second";
      }
   }
   attribute def UtcTimeInstantValue :> DateTime { :>> mRef = UTC; }
   attribute utcTimeInstant: UtcTimeInstantValue;
```

Generic time scale to express a time instant.

Captures the specification of the time instant with value zero, also known as the (reference) epoch.

Representation of a time instant quantity.

Generic representation of a time instant as a calendar date and time of day.

Representation of the Coordinated Universal Time (UTC) time scale.

# Time (2)

SST

```
   attribute def Iso8601DateTimeEncoding :> String;

   attribute def Iso8601DateTime :> UtcTimeInstantValue {
      attribute :>> num = getElapsedUtcTime(val);
      attribute val: Iso8601DateTimeEncoding;
      private calc getElapsedUtcTime(iso8601DateTime: Iso8601Da
   }

   attribute def Iso8601DateTimeStructure :> UtcTimeInstantValue {
      attribute :>> num = getElapsedUtcTime(year, month, day, hour, minute, second,
           microsecond, hourOffset, minuteOffest);
      attribute :>> mRef = UTC;
      attribute year: Integer;
      attribute month: Natural;
      attribute day: Natural;
      attribute hour: Natural;
      attribute minute: Natural;
      attribute second: Natural;
      attribute microsecond: Natural;
      attribute hourOffset: Integer;
      attribute minuteOffest: Integer;
      private calc getElapsedUtcTime(year: Integer, month: Natural, day: Natural,
         hour: Natural, minute: Natural, second: Natural, microsecond: Natural,
         hourOffset: Integer, minuteOffest: Integer) : Real;
   }
   …
}
```

Representation of an ISO 8601-1 date and time in extended string format.

Representation of an ISO 8601 date and time with explicit date and time component attributes.