SysML-Embeddable Ontology & Implementation v1.1

SysML Implementation

The concepts we described in the conceptual ontology are mapped to concrete implementation in SysML so that they can be used to actually model behavior. In this section we describe first the embedding of the ontology into SysML (so that the user can understand how the concepts are made concrete) and then provide a SysML example using the simple flashlight used in the conceptual example.

SysML-Embeddable Behavior Ontology - Description

The conceptual ontology presented the essential concepts to capture behavior. However, several reasons exist why we do not use this ontology directly as of now.

- IMCE has developed an infrastructure to generate a SysML profile from the OWL file that describes the ontology. This infrastructure and the tools used require for now that the ontology follows some characteristics. For example, the ontology should use "simple range class expressions", i.e., the relationships described in the ontology must have a unique source class and a unique target class. Accordingly, in the case where a relationship targets several classes, an abstract parent class is created to conform to that principle. Also, the mapping of a single conceptual class into multiple SysML elements is currently not supported by the IMCE tools. As a consequence, a one-to-one mapping to SysML lead to the introduction of additional classes in the ontology.
- Some decisions were made regarding the choice of embedding of the classes and relationships introduced in the conceptual ontology. These decisions resulted in the introduction of intermediate classes in SysML that should be captured in the ontology so that its existence is recognized and that validation rules can be written from an ontological perspective. For example, it was chosen to embed state variables as value properties in SysML. Value properties cannot exist on their own in SysML, they must have a "container". This container was in turn given an ontological existence by adding an associated class into the ontology.

The result of this process is an other ontology that we call "SysML-embeddable behavior ontology". This ontology is the closest possible to the original conceptual ontology while conforming to the constraints described above. Despite the introduction of a significant amount of new classes and relationships, the core of the behavior ontology remains the same than what was described in the conceptual ontology. **Note that, in the future, the generation of a pattern SysML profile for the behavior pattern will be done directly from the conceptual ontology, and that the implementation concerns about will be handled "behind-the-scene". In the meantime, the ontology presented on this page is the temporary solution that was selected to obtain a behavior SysML profile using the current infrastructure.** The introduced concepts and relationships are shown graphically with a blue fill.

The remainder of this page presents side-by-side the original conceptual ontology, the "SysML-embeddable behavior ontology", and the SysML example so that the reader can understand the relationships between the three representations.

Note that the concepts are represented by blocks and relationships are represented by ellipses on directed arrows.

State variable

In the conceptual ontology, `StateVariables` characterize `BehavingElements`. As mentioned above, it was chosen to embed `StateVariables` as value properties, hence requiring the introduction of a "container" for these value properties. The class that represent these containers is named `PropertyGroup`, and is shown in Figure 1. The relationship between `PropertyGroup` and the `StateVariables` it contains is named `hasAttribute`, and represents this containment. `PropertyGroup` is a specialization of the Analysis:Characterization concept (and is thus embedded as a Component.Block, i.e. a UML Component stereotyped by the SysML Block stereotype). The `hasAttribute` relationship is embedded using the associated UML concept of A_ownedAttribute_class: it represents the UML relation between a Class and the attributes (here `StateVariable`) owned by that Class. The SysML example provided in Figure 2 shows the case of the battery `StateVariables` (ignore for now the value types of the value properties in SysML, these are covered later in this page).

Note that the grouping of `StateVariables` into `PropertyGroups` is left to the modeler. All the `StateVariables` of a `PropertyGroup` characterizing the `BehavingElement` are applicable (no notion of only certain subsets available). The same scope applies if the `PropertyGroup` is provided by a library.

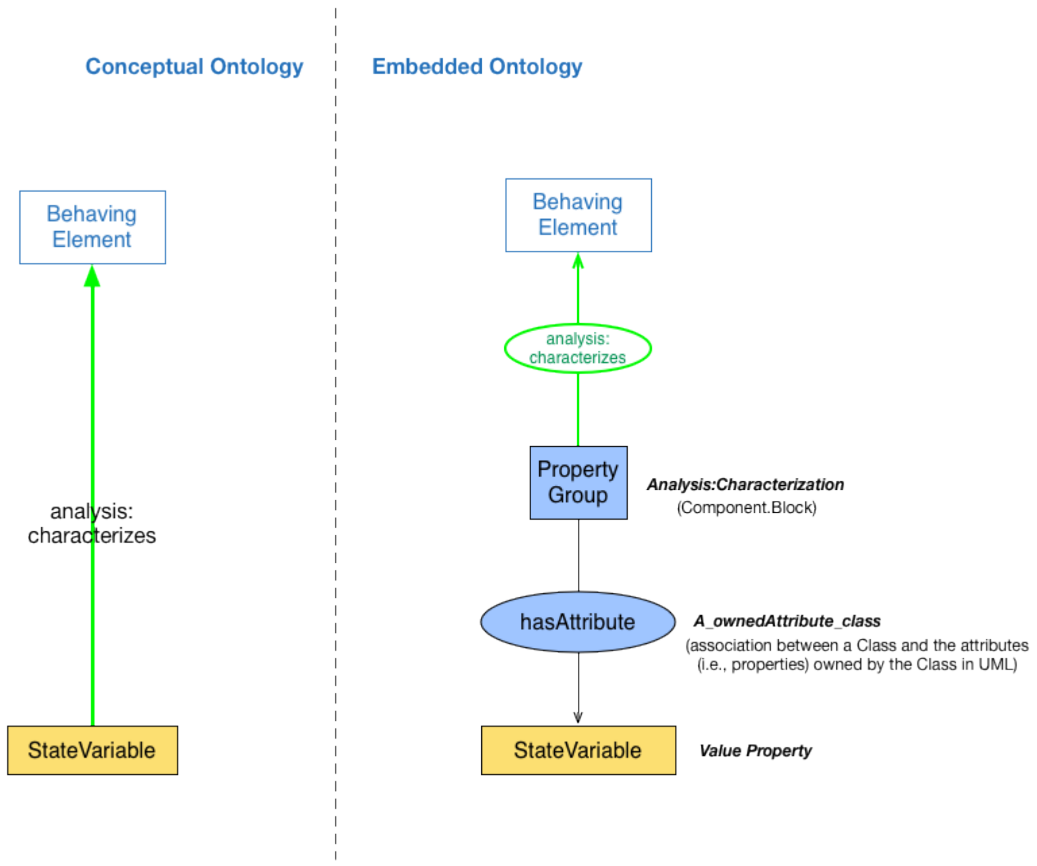


Figure 1. `StateVariable` embedding

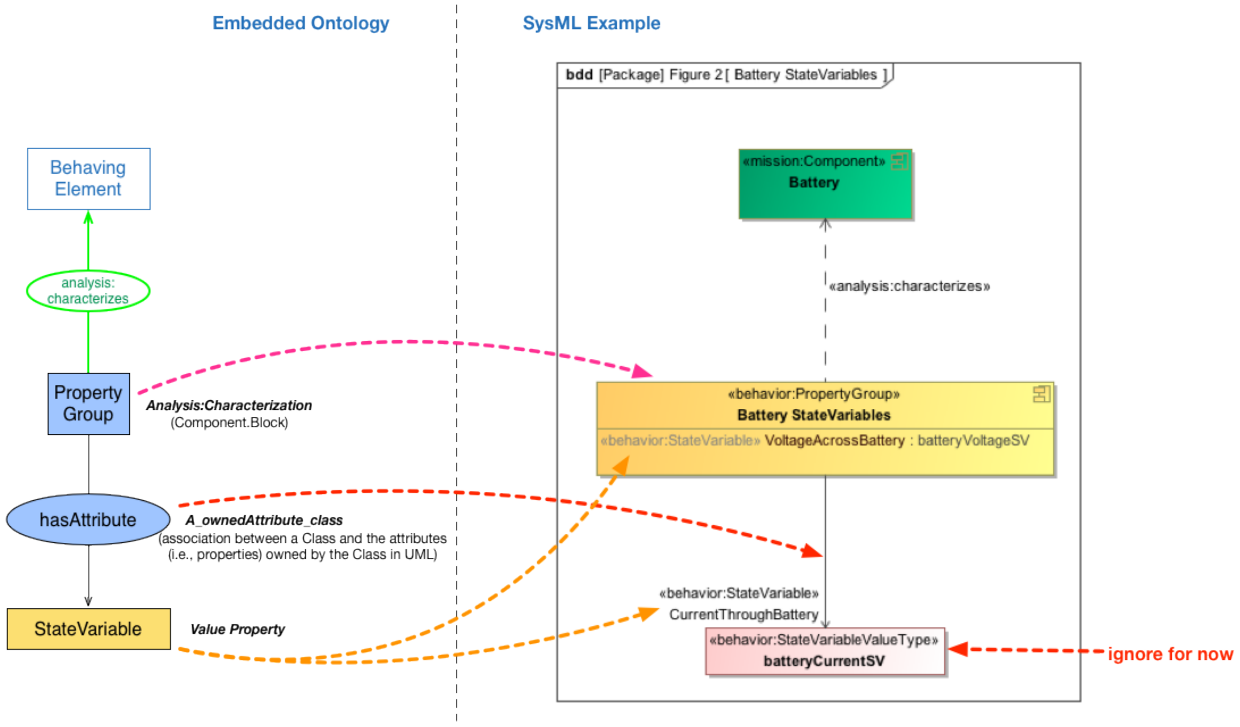


Figure 2. Battery `StateVariables`

Parameter

In the conceptual ontology, `Parameters` are related to `BehavingElements` in the same fashion than `StateVariables`. They are also embedded in a similar fashion, using value properties. `PropertyGroup` can own `StateVariables` and `Parameters`, as shown in Figure 3. The grouping of these properties into specific `PropertyGroups` is left to the modeler's decision. It can be seen in Figure 3 that the `hasAttribute` relationship has `PropertyGroup` as a source and `StateVariable` and `Parameter` as targets. To conform to the simple range class expression principle mentioned previously, an additional abstract class (i.e., a class that will not appear explicitly in models) is introduced as a generalization of `StateVariable` and `Parameter`, named `Property`. This class becomes in turn the target of the `hasAttribute` relationship. Figure 4 shows examples of this pattern in SysML, and with different options for grouping `StateVariables` and `Parameters` for the lamp and the battery. The grouping is left to the modelers and their intent for such groups.

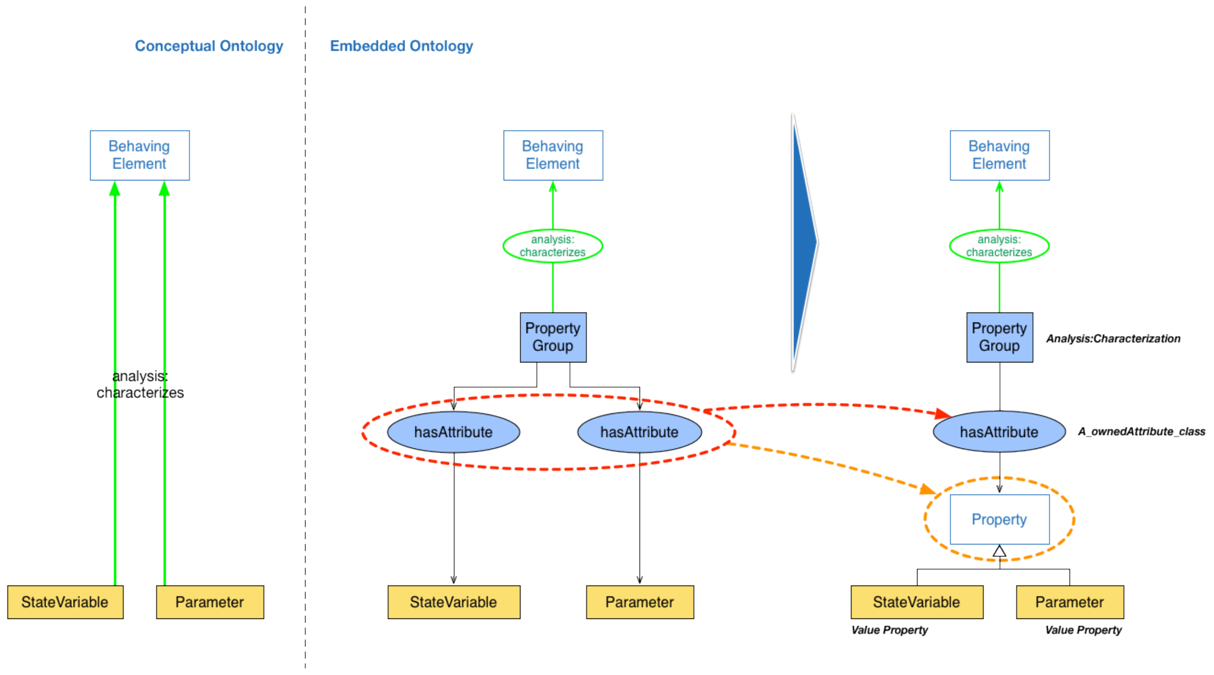


Figure 3. `Properties`

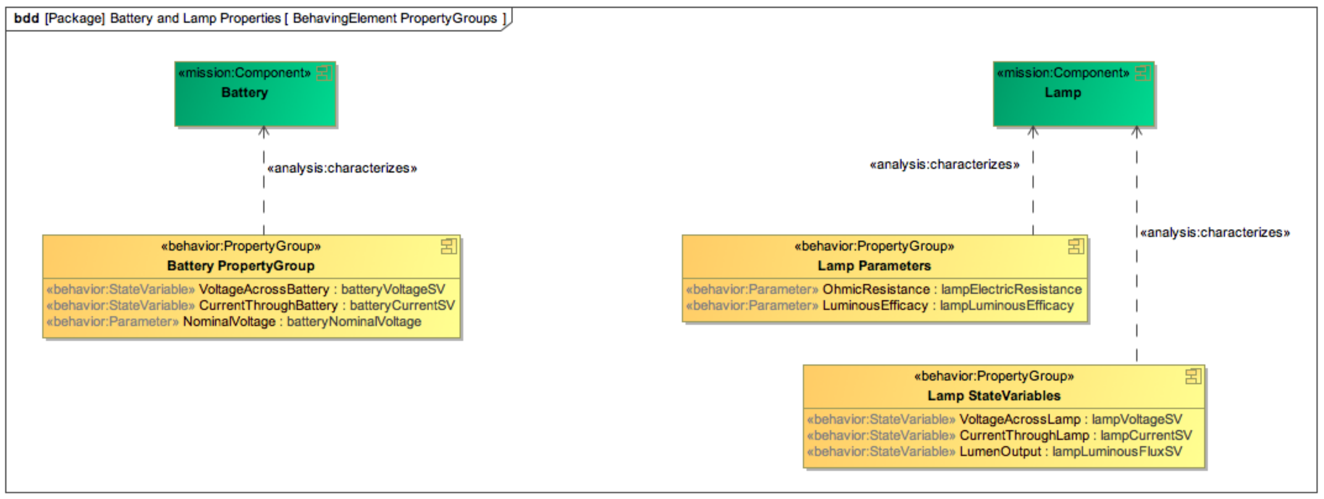

bdd [Package] Battery and Lamp Properties [BehavingElement PropertyGroups]
«mission:Component»
Battery
«analysis:characterizes»
«behavior:PropertyGroup»
Battery PropertyGroup
«behavior:StateVariable» VoltageAcrossBattery : batteryVoltageSV
«behavior:StateVariable» CurrentThroughBattery : batteryCurrentSV
«behavior:Parameter» NominalVoltage : batteryNominalVoltage
«mission:Component»
Lamp
«analysis:characterizes»
«analysis:characterizes»
«behavior:PropertyGroup»
Lamp Parameters
«behavior:Parameter» OhmicResistance : lampElectricResistance
«behavior:Parameter» LuminousEfficacy : lampLuminousEfficacy
«behavior:PropertyGroup»
Lamp StateVariables
«behavior:StateVariable» VoltageAcrossLamp : lampVoltageSV
«behavior:StateVariable» CurrentThroughLamp : lampCurrentSV
«behavior:StateVariable» LumenOutput : lampLuminousFluxSV

Figure 4. Battery and Lamp `Properties`

ElementBehavior

In the conceptual ontology, `ElementBehavior` characterizes its `BehavingElement` and constrains `StateVariable`(s), as shown in the left panel of Figure 5.

The embedding of `ElementBehavior` depends on the intent of the modeler: two embedding are provided here, one for the capture of mathematical constraints, and one for the capture of state machines.

Mathematical constraints

This embedding of the `ElementBehavior` makes a distinction between the constraints and their owner. For this purpose, two classes are introduced: the `ElementBehaviorConstraint` is a (Component) Constraint Block, while the `ElementBehaviorCharacterization` is an Analysis:Characterization that owns the potentially many Constraint Blocks. The relationship between the `ElementBehaviorCharacterization` and the `ElementBehaviorConstraint`, named `hasConstraint`, is embedded as a binary composite association (or "black-diamond" association). An additional class is created to complete the picture: the constraint in the Constraint Block does not refer directly to the `StateVariables` of interest: the elements participating in the constraint are SysML Constraint Parameters owned by the Constraint Block (through A_ownedAttribute_structuredClassifier, the appropriate UML association in this case). The class that represents these constraint parameter is named `StateVariableConstraintParticipant`. In the constraint, the participants "play the role" of the relevant state variables: this relationship is captured through the `playsRoleOf` relationship that is embedded as a binding connector between `StateVariableConstraintParticipant` and `StateVariable`. This choice allows for potential reuse of the Constraint Block and their addition to libraries.

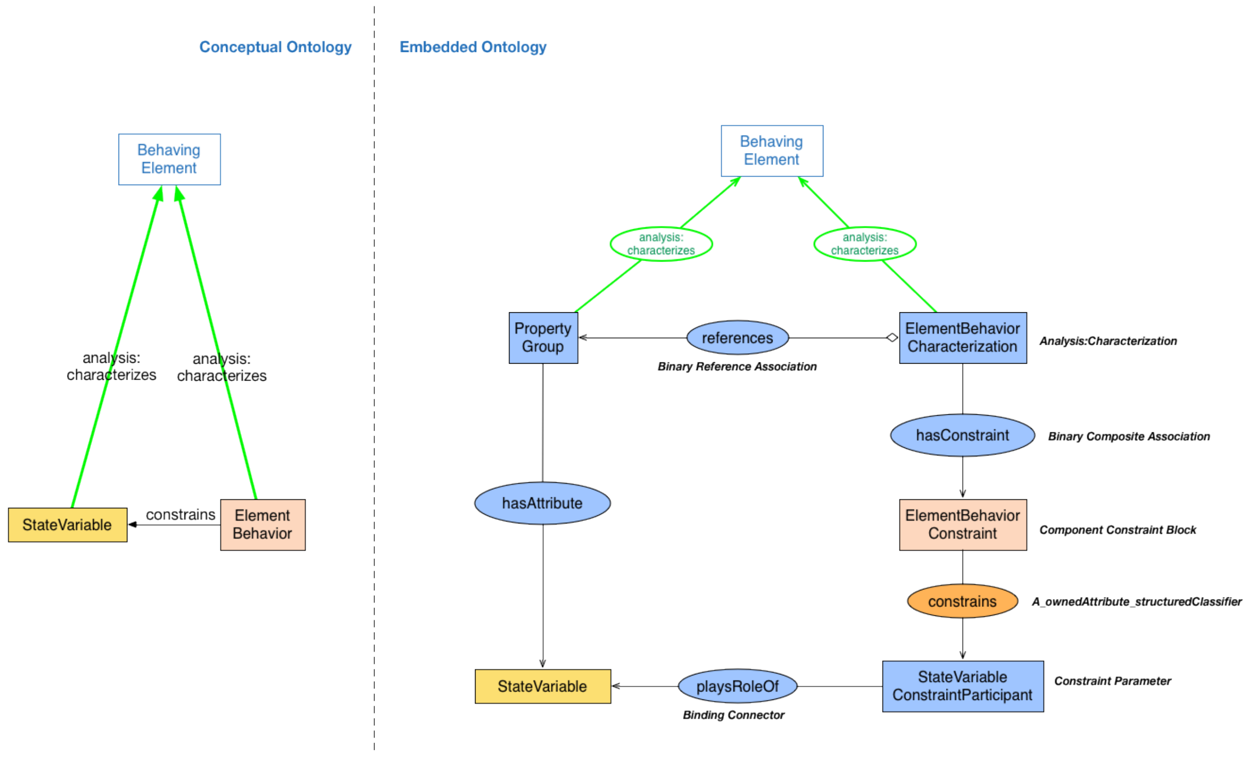


Figure 5. `ElementBehavior` and `StateVariables`

An `ElementBehaviorConstraint constrains StateVariables`, but also `uses` potentially `Parameters` in the expression of the constraint. This case is handled in a symmetrical fashion as shown in Figure 6, and the `ElementBehaviorConstraint` owns a `ParameterConstraintParticipant` (through the `uses` relationship) that plays the role of a `Parameter`.

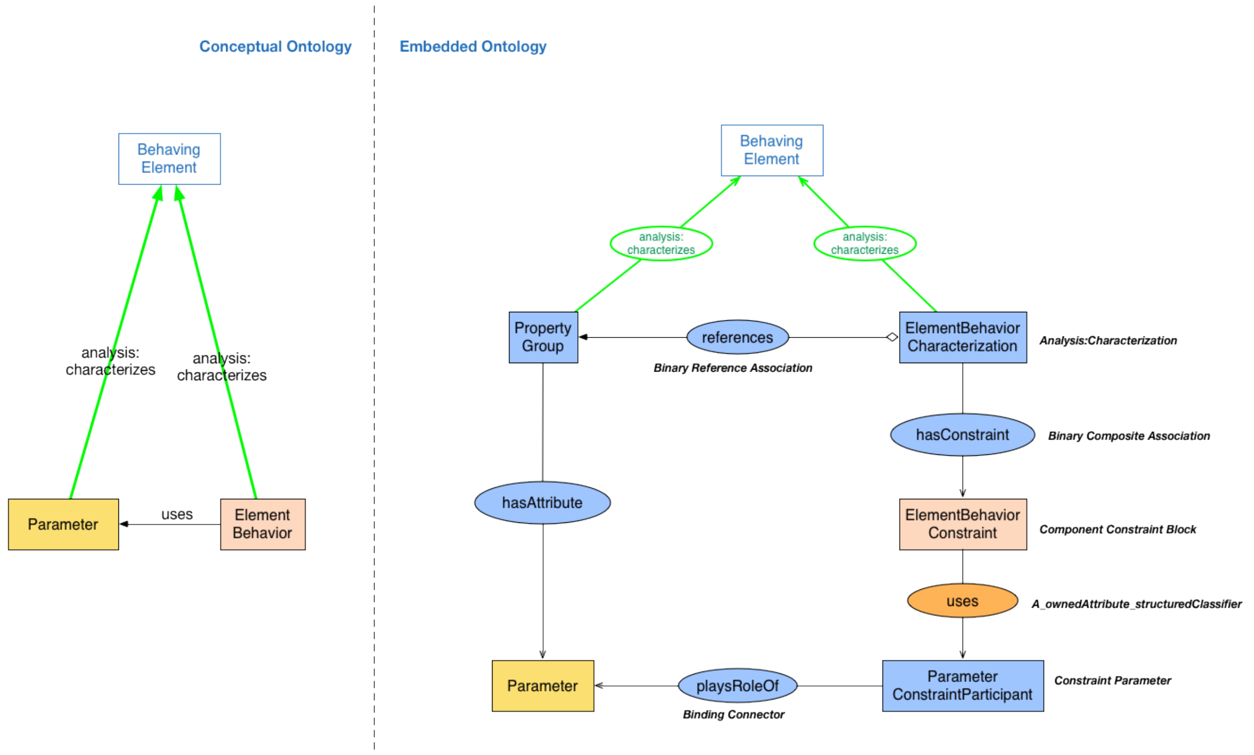
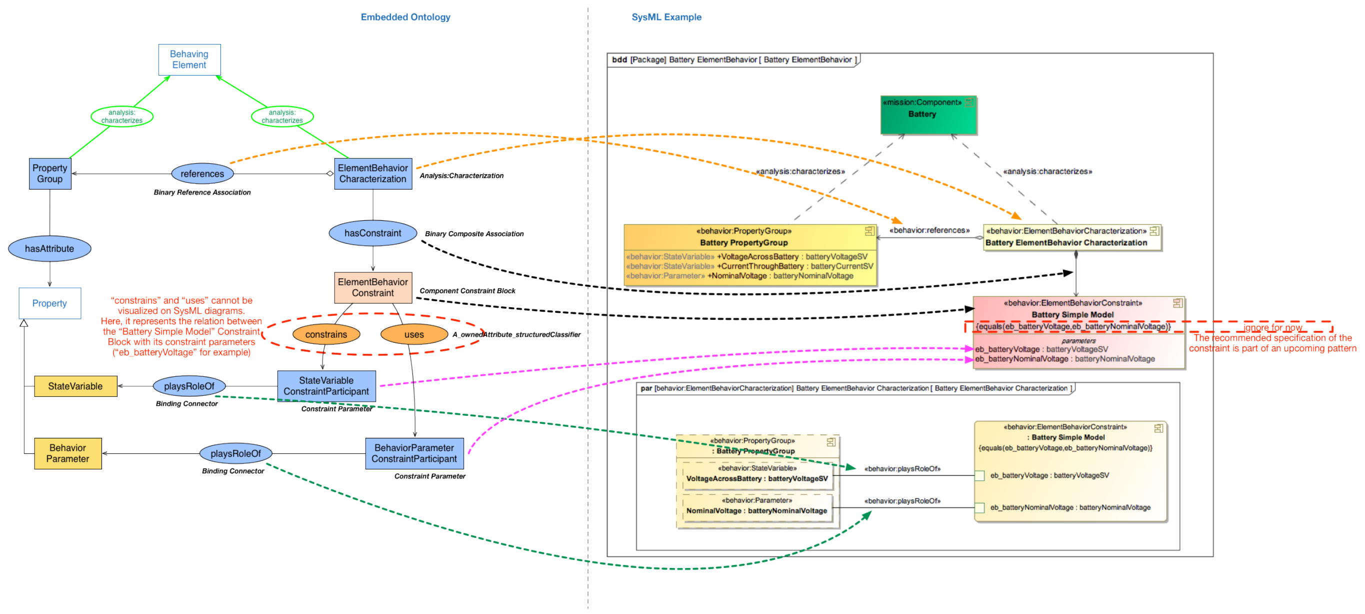


Figure 6. `ElementBehavior` and `Properties`

An example of the `ElementBehavior` of the battery is shown in Figure 7, and displays elements described above. The "white diamond" relationship `references` allows the construction of the parametric diagram of the `ElementBehaviorCharacterization` in which the `StateVariable`/`Parameter-ConstraintParticipant` are bound to their respective `StateVariables`/`Parameters`. Also note that the Constraint Block owns a constraint that specifies the relationship between the `StateVariables` / `Parameters`: **the recommended specification of a constraint is part of an upcoming pattern**.



Embedded Ontology
SysML Example
Behaving Element
analysis: characterizes
analysis: characterizes
Property Group
references
Binary Reference Association
ElementBehavior Characterization
Analysis:Characterization
hasConstraint
Binary Composite Association
hasAttribute
Property
ElementBehavior Constraint
Component Constraint Block
"constrains" and "uses" cannot be visualized on SysML diagrams. Here, it represents the relation between the "Battery Simple Model" Constraint Block with its constraint parameters ("eb_batteryVoltage" for example)
constrains
uses
A_ownedAttribute_structuredClassifier
StateVariable
playsRoleOf
Binding Connector
StateVariable ConstraintParticipant
Constraint Parameter
Behavior Parameter
playsRoleOf
Binding Connector
BehaviorParameter ConstraintParticipant
Constraint Parameter
bdd [Package] Battery ElementBehavior [Battery ElementBehavior]
«mission:Component» Battery
«analysis:characterizes»
«analysis:characterizes»
«behavior:PropertyGroup» Battery PropertyGroup
«behavior:StateVariable» +VoltageAcrossBattery : batteryVoltageSV
«behavior:StateVariable» +CurrentThroughBattery : batteryCurrentSV
«behavior:Parameter» +NominalVoltage : batteryNominalVoltage
«behavior:references»
«behavior:ElementBehaviorCharacterization» Battery ElementBehavior Characterization
«behavior:ElementBehaviorConstraint» Battery Simple Model
{equals(eb_batteryVoltage,eb_batteryNominalVoltage)}
ignore for now
The recommended specification of the constraint is part of an upcoming pattern
parameters
eb_batteryVoltage : batteryVoltageSV
eb_batteryNominalVoltage : batteryNominalVoltage
par [behavior:ElementBehaviorCharacterization] Battery ElementBehavior Characterization [Battery ElementBehavior Characterization]
«behavior:PropertyGroup» : Battery PropertyGroup
«behavior:StateVariable» VoltageAcrossBattery : batteryVoltageSV
«behavior:Parameter» NominalVoltage : batteryNominalVoltage
«behavior:playsRoleOf»
«behavior:playsRoleOf»
«behavior:ElementBehaviorConstraint» : Battery Simple Model
{equals(eb_batteryVoltage,eb_batteryNominalVoltage)}
eb_batteryVoltage : batteryVoltageSV
eb_batteryNominalVoltage : batteryNominalVoltage

Figure 7. Battery `ElementBehavior` example

State machines

There are cases where state machines are a more compact and efficient representation, for example to capture the discrete modes of a device. We have added to the embedding for the `ElementBehavior` of the conceptual ontology support for using UML state machines, as they fit in the "`StateVariable` constraint" definition of `ElementBehavior`. However, the only semantics of the UML state machines that are recognized in this embedding of the behavior pattern are the ones listed below. The remaining semantics of UML state machines are out of the purview of the behavior pattern and their usage is left to the agreements between modelers.

The `ElementBehaviorCharaterization` can own a UML state machine, represented by the `hasAutomaton` relationship and the `AttributeAutomatonConstraint`. The UML state machines can have regions, and states and these are captured in the ontology as `AutomatonRegion` and `AutomatonState`, with the appropriate `hasRegion` and `hasState` relationships. These concepts and relations are shown in Figure 8.

As in the flashlight example, some constraints can be applied while in some states: the switch voltage is constrained to be zero in the closed state and the switch current is constrained to be zero in the open state. To capture these constraints, we make use of the first embedding of `ElementBehavior` presented above. The constraints are described using an `ElementBehaviorConstraint`, and the domain of applicability to a specific state/region is handled by pointing a `analysis:characterizes` relationship from the `ElementBehaviorConstraint` to the appropriate `AutomatonState/Region` (these two concepts are generalized into the `AutomatonScope`). This convention allows to clearly state what is the scope of the constraint:

- if an `ElementBehaviorConstraint` points at an `AutomatonState` (or a set of `AutomatonStates`), then the constraint specified by this `ElementBehaviorConstraint` is valid only in the context of these `AutomatonStates`;
- if an `ElementBehaviorConstraint` points at an `AutomatonRegion` (or a set of `AutomatonRegions`), then the constraint specified by this `ElementBehaviorConstraint` is valid only in the context of these `AutomatonRegions`;
- if an `ElementBehaviorConstraint` does not point to `AutomatonState`(s) or `AutomatonRegion`(s) in the presence of an `AttributeAutomatonConstraint`, it means that the constraints specified by the `ElementBehaviorConstraint` is valid for all `AutomatonRegions` and `AutomatonStates` of that `AttributeAutomatonConstraint`.

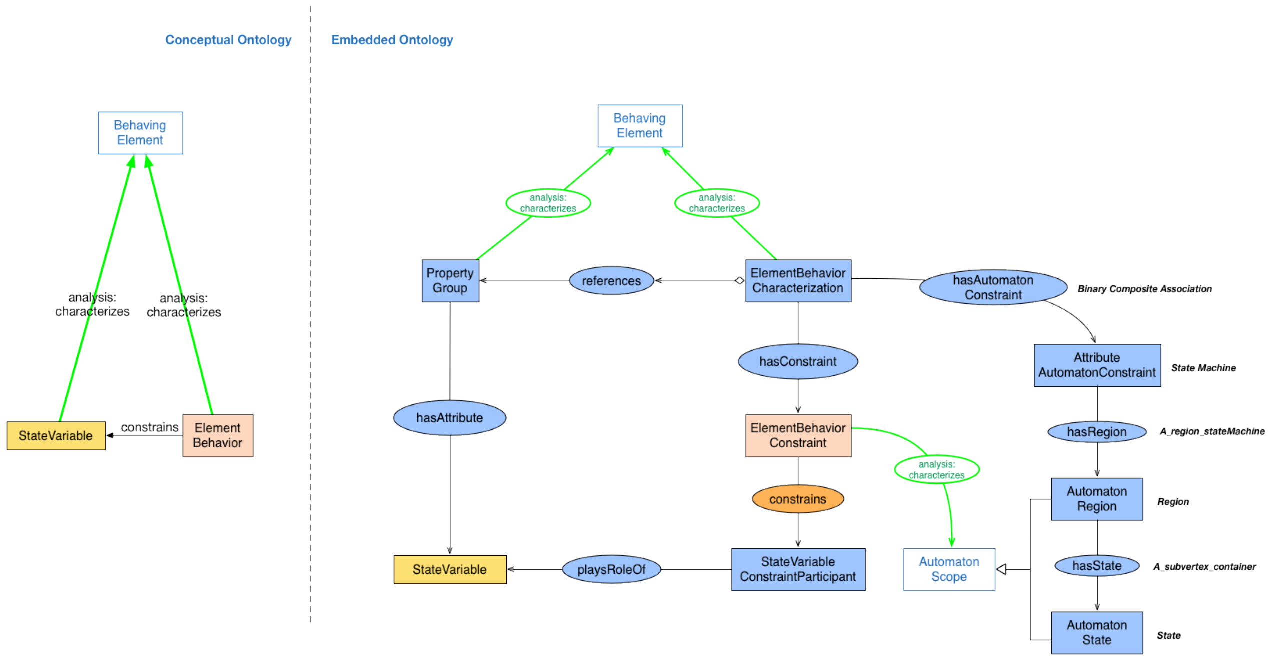


Figure 8. `ElementBehavior` and state machines

Figure 9 shows the switch `ElementBehavior` of the flashlight example captured using the embedding presented above. Note that the `Codomain` of the Switch position `StateVariable` is defined using the state machine, as seen in the structure of the value type typing the Switch position `StateVariable`. More details on value types is provided later in this page. The Switch position has two `States`: OPEN and CLOSED, and represented using states in the state machine. The constraint on the voltage for the CLOSED state is captured using an `ElementBehaviorConstraint` that points an `analysis:characterizes` dependency to the CLOSED state. The constraint on the voltage for the OPEN state is captured using an `ElementBehaviorConstraint` that points an `analysis:characterizes` dependency to the OPEN state.

Note that in Figure 9, no triggers were modeled, but they could have been captured using the semantics of UML State Machine.

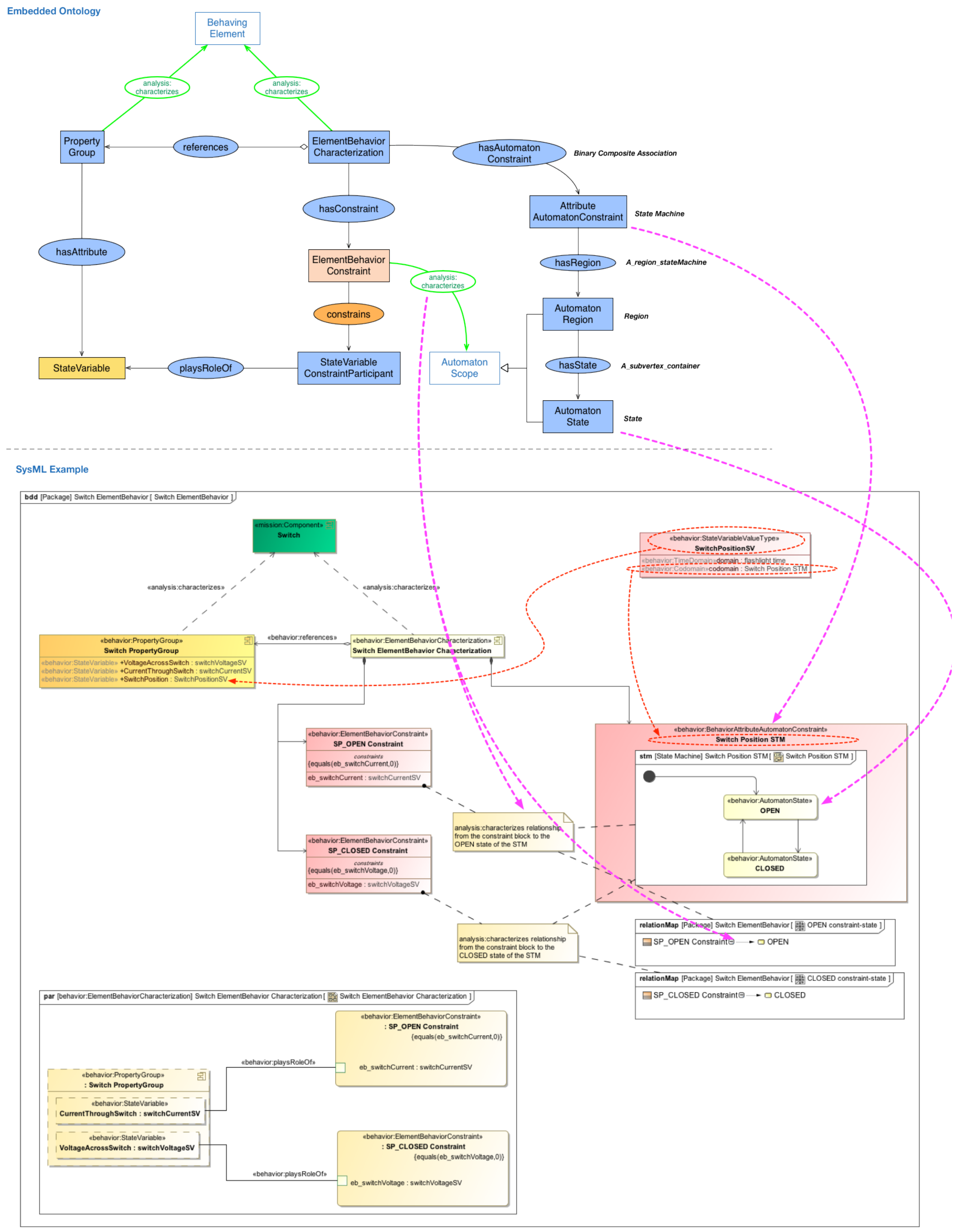

Embedded Ontology
Behaving Element
analysis: characterizes
analysis: characterizes
Property Group
references
ElementBehavior Characterization
hasAutomaton Constraint
Binary Composite Association
Attribute AutomatonConstraint
State Machine
hasConstraint
hasAttribute
ElementBehavior Constraint
analysis: characterizes
hasRegion
A_region_stateMachine
Automaton Region
Region
constrains
StateVariable
playsRoleOf
StateVariable ConstraintParticipant
Automaton Scope
hasState
A_subvertex_container
Automaton State
State
SysML Example
bdd [Package] Switch ElementBehavior [Switch ElementBehavior]
«mission:Component» Switch
«analysis:characterizes»
«analysis:characterizes»
«behavior:PropertyGroup» Switch PropertyGroup
«behavior:StateVariable» +VoltageAcrossSwitch : switchVoltageSV
«behavior:StateVariable» +CurrentThroughSwitch : switchCurrentSV
«behavior:StateVariable» +SwitchPosition : SwitchPositionSV
«behavior:references»
«behavior:ElementBehaviorCharacterization» Switch ElementBehavior Characterization
«behavior:StateVariableValueType» SwitchPositionSV
«behavior:TimeDomain»domain : flashlight time
«behavior:Codomain»codomain : Switch Position STM
«behavior:ElementBehaviorConstraint» SP_OPEN Constraint
constraints
{equals(eb_switchCurrent,0)}
eb_switchCurrent : switchCurrentSV
«behavior:ElementBehaviorConstraint» SP_CLOSED Constraint
constraints
{equals(eb_switchVoltage,0)}
eb_switchVoltage : switchVoltageSV
analysis:characterizes relationship from the constraint block to the OPEN state of the STM
analysis:characterizes relationship from the constraint block to the CLOSED state of the STM
«behavior:BehaviorAttributeAutomatonConstraint» Switch Position STM
stm [State Machine] Switch Position STM [Switch Position STM]
«behavior:AutomatonState» OPEN
«behavior:AutomatonState» CLOSED
relationMap [Package] Switch ElementBehavior [OPEN constraint-state]
SP_OPEN Constraint → OPEN
relationMap [Package] Switch ElementBehavior [CLOSED constraint-state]
SP_CLOSED Constraint → CLOSED
par [behavior:ElementBehaviorCharacterization] Switch ElementBehavior Characterization [Switch ElementBehavior Characterization]
«behavior:ElementBehaviorConstraint» : SP_OPEN Constraint
{equals(eb_switchCurrent,0)}
eb_switchCurrent : switchCurrentSV
«behavior:playsRoleOf»
«behavior:PropertyGroup» : Switch PropertyGroup
«behavior:StateVariable» CurrentThroughSwitch : switchCurrentSV
«behavior:StateVariable» VoltageAcrossSwitch : switchVoltageSV
«behavior:ElementBehaviorConstraint» : SP_CLOSED Constraint
{equals(eb_switchVoltage,0)}
«behavior:playsRoleOf»
eb_switchVoltage : switchVoltageSV

Figure 9. Switch `ElementBehavior`

Interation aspects

Interaction

Another constraint type for `StateVariable` is in the context of an `Interaction`. The right part of of Figure 10 describes the new classes introduced to capture the constraints that occur during `BehavingElement` interactions.The relationship `joins` is embedded as a shared association ("white-diamond" association) from the `Interaction` block to the different `PropertyGroups` that have the `StateVariables` or `Parameters` that are involved in the interaction constraints.

The pattern to capture the interaction constraints is exactly symmetrical with the capture of `ElementBehavior` constraints: the `Interaction` (embedded as a Component Block) owns through the `hasConstraint` relationship `InteractionBehaviorConstraints` (embedded as Component Constraint Block). These constraint blocks in turn own (through the `constrains` relationship) `StateVariableConstraintParticipants` that are related to `StateVariables` using binding connectors stereotyped by `playsRoleOf`. The `InteractionBehavior` from the Conceptual Ontology is fulfilled by the `InteractionBehaviorConstraint` in the SysML-embeddable Ontology.

In a similar fashion, `InteractionBehaviorConstraint` can use `Parameters`, and this pattern is shown in Figure 11.

An example of the `InteractionBehaviorConstraint` is shown in Figure 12 using the flashlight example (mesh-analysis constraint version). Note once again that **the recommended specification of a constraint is part of an upcoming pattern**.

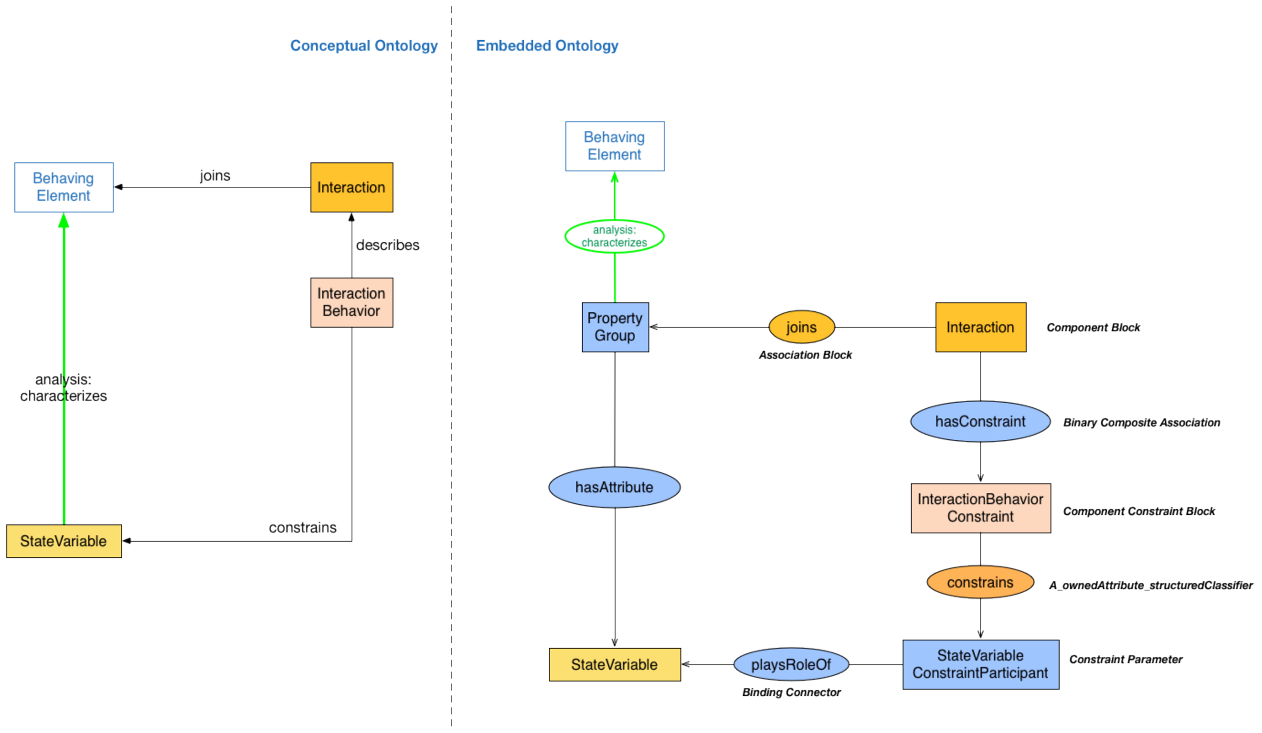


Figure 10. `Interaction` and `StateVariables`

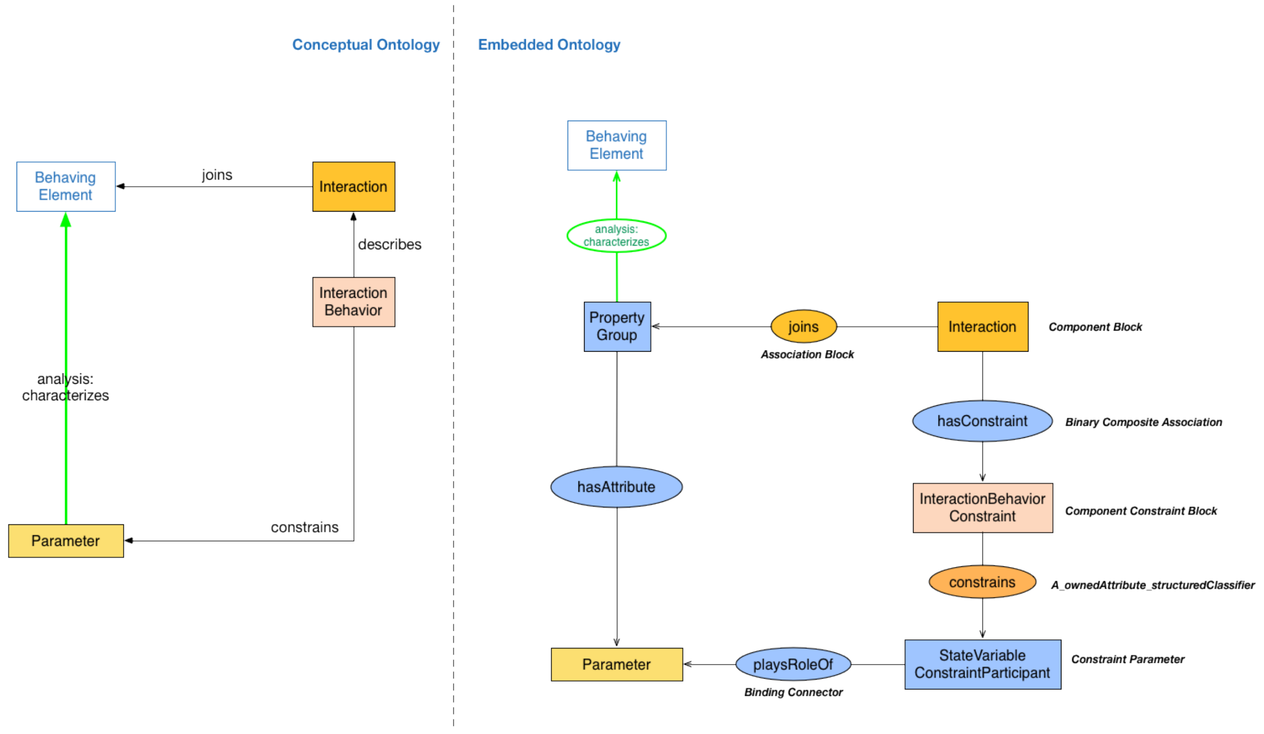


Figure 11. `Interaction` and `Parameters`

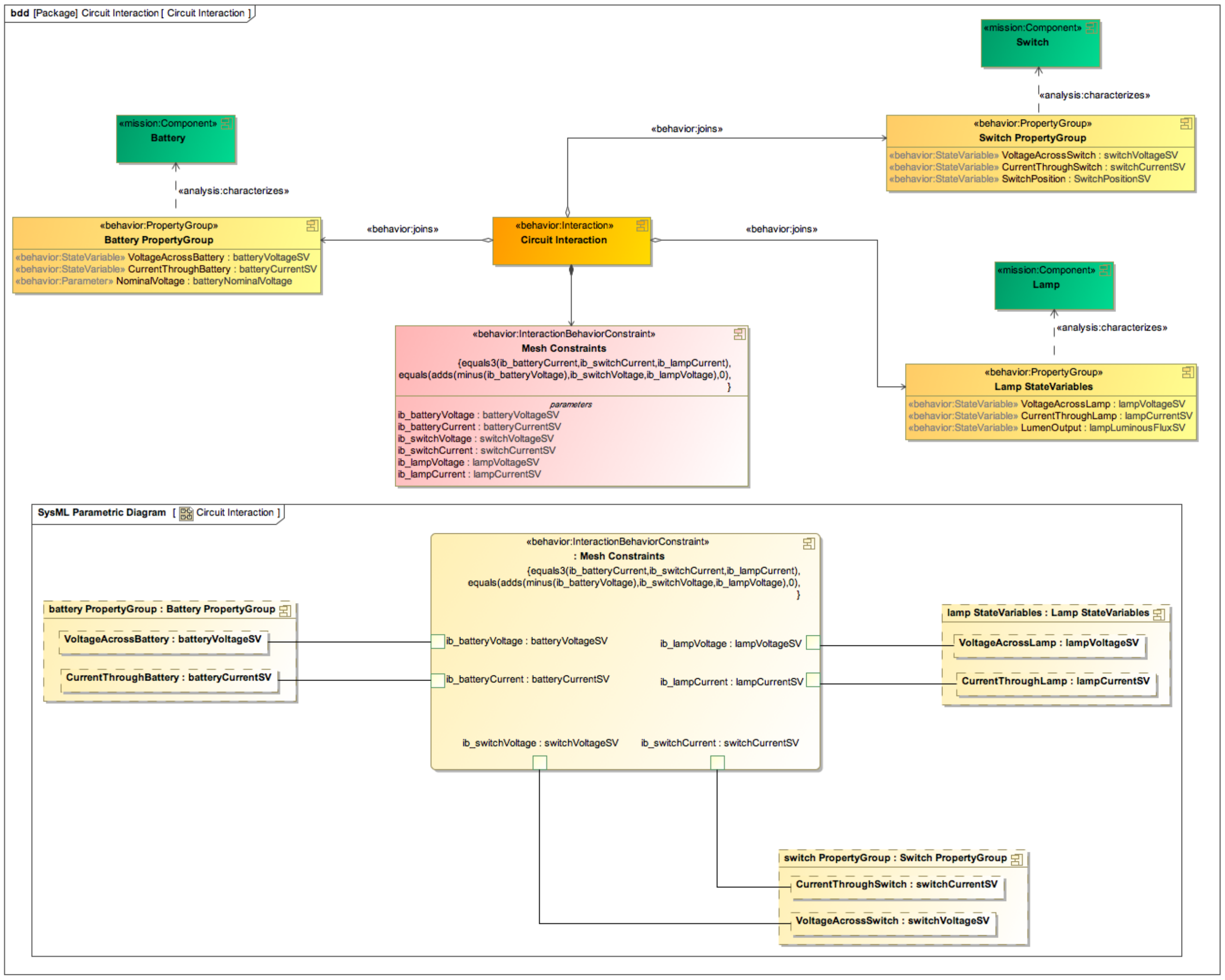

bdd [Package] Circuit Interaction [Circuit Interaction]
«mission:Component» Switch
«analysis:characterizes»
«behavior:PropertyGroup» Switch PropertyGroup
«behavior:StateVariable» VoltageAcrossSwitch : switchVoltageSV
«behavior:StateVariable» CurrentThroughSwitch : switchCurrentSV
«behavior:StateVariable» SwitchPosition : SwitchPositionSV
«mission:Component» Battery
«analysis:characterizes»
«behavior:joins»
«behavior:PropertyGroup» Battery PropertyGroup
«behavior:StateVariable» VoltageAcrossBattery : batteryVoltageSV
«behavior:StateVariable» CurrentThroughBattery : batteryCurrentSV
«behavior:Parameter» NominalVoltage : batteryNominalVoltage
«behavior:joins»
«behavior:Interaction» Circuit Interaction
«behavior:joins»
«mission:Component» Lamp
«analysis:characterizes»
«behavior:InteractionBehaviorConstraint» Mesh Constraints
{equals3(ib_batteryCurrent,ib_switchCurrent,ib_lampCurrent), equals(adds(minus(ib_batteryVoltage),ib_switchVoltage,ib_lampVoltage),0), }
parameters
ib_batteryVoltage : batteryVoltageSV
ib_batteryCurrent : batteryCurrentSV
ib_switchVoltage : switchVoltageSV
ib_switchCurrent : switchCurrentSV
ib_lampVoltage : lampVoltageSV
ib_lampCurrent : lampCurrentSV
«behavior:PropertyGroup» Lamp StateVariables
«behavior:StateVariable» VoltageAcrossLamp : lampVoltageSV
«behavior:StateVariable» CurrentThroughLamp : lampCurrentSV
«behavior:StateVariable» LumenOutput : lampLuminousFluxSV
SysML Parametric Diagram [Circuit Interaction]
«behavior:InteractionBehaviorConstraint» : Mesh Constraints
{equals3(ib_batteryCurrent,ib_switchCurrent,ib_lampCurrent), equals(adds(minus(ib_batteryVoltage),ib_switchVoltage,ib_lampVoltage),0), }
battery PropertyGroup : Battery PropertyGroup
VoltageAcrossBattery : batteryVoltageSV
CurrentThroughBattery : batteryCurrentSV
ib_batteryVoltage : batteryVoltageSV
ib_batteryCurrent : batteryCurrentSV
ib_lampVoltage : lampVoltageSV
ib_lampCurrent : lampCurrentSV
lamp StateVariables : Lamp StateVariables
VoltageAcrossLamp : lampVoltageSV
CurrentThroughLamp : lampCurrentSV
ib_switchVoltage : switchVoltageSV
ib_switchCurrent : switchCurrentSV
switch PropertyGroup : Switch PropertyGroup
CurrentThroughSwitch : switchCurrentSV
VoltageAcrossSwitch : switchVoltageSV

Figure 12. Flashlight circuit `Interaction`

Interaction using InteractionTerminal (optional)

As discussed in the conceptual ontology, this ontology gives to the modeler the choice to use InteractionTerminal for experimentation. This section explains how the related concepts and relationships are embedded in SysML. These concepts will be present and shown greyed in the rest of this page.
Expand this section if you want to read more.

Synthesis of constraints

As described in the previous two sections, some relationships appears several times with different sources or targets. To conform to our simple range class expression pattern, some synthesis is necessary and presented below.

- Let's address first the case of the `hasConstraint` relationship: it is shown in the upper panel of Figure 16 that it has two sources (`ElementBehaviorCharacterization` and `Interaction`) and two targets (`ElementBehaviorConstraint` and `InteractionBehaviorConstraint`). As a consequence, two abstract superclasses are created, namely the `AttributeConstraintOwner` and the `AttributeConstraint` as shown in the lower part of Figure 16. Range restrictions are defined to prevent incorrect mapping between subclasses. **Range restrictions are defined and shown using red dashed lines** in Figure 16 for the allowed use cased for the hasConstraint relationship. This visual representation will be used in the remainder of this page.
]

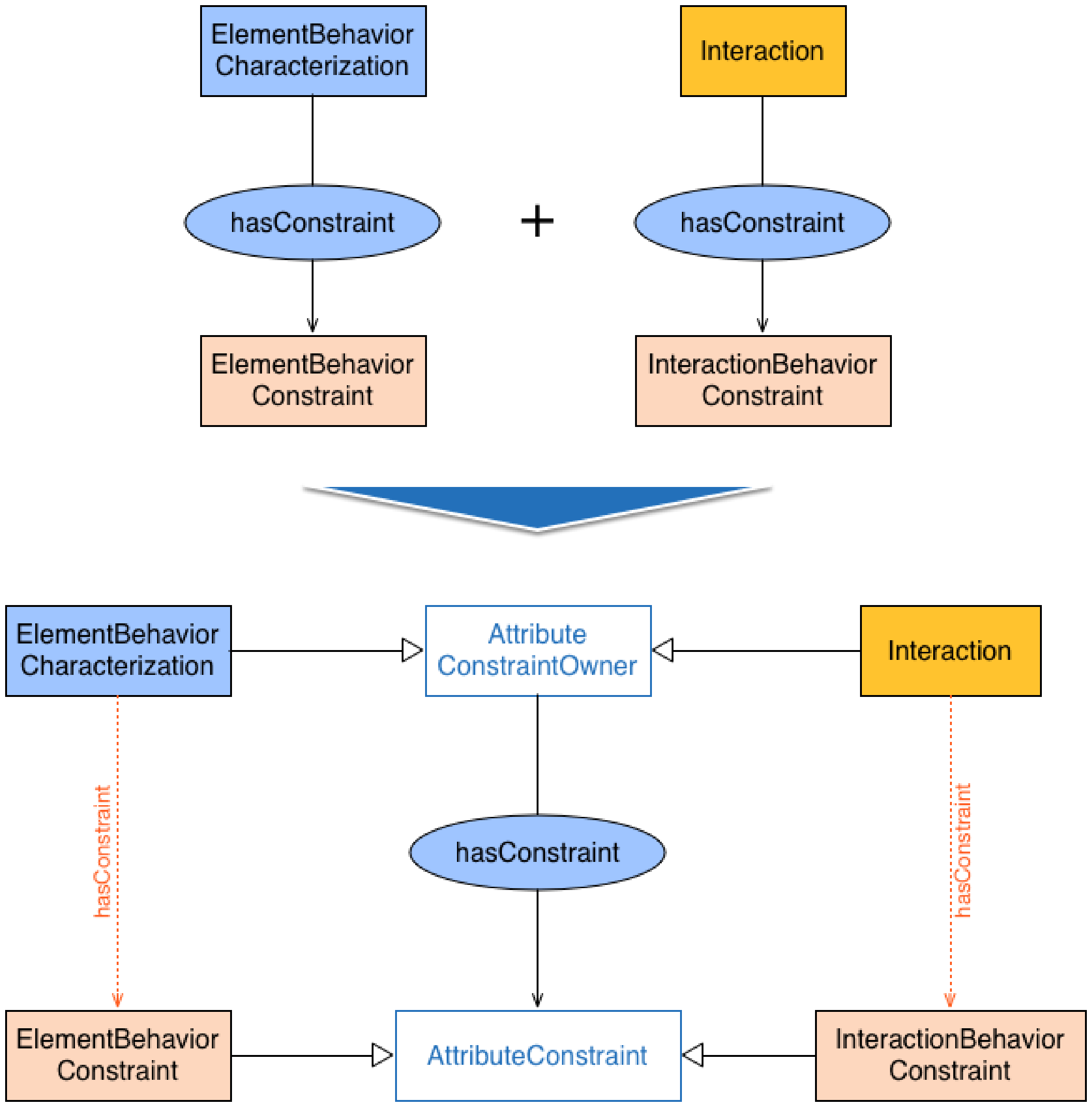

ElementBehavior Characterization
Interaction
hasConstraint
+
hasConstraint
ElementBehavior Constraint
InteractionBehavior Constraint
ElementBehavior Characterization
Attribute ConstraintOwner
Interaction
hasConstraint
hasConstraint
hasConstraint
ElementBehavior Constraint
AttributeConstraint
InteractionBehavior Constraint

Figure 16. `hasConstraint` synthesis

- Next, let's address the cases of the `constrains`, `uses` and `playsRoleOf` relationships. As shown in the upper panel of Figure 17, the `constrains` relationship has two source classes (`ElementBehaviorConstraint` and `InteractionBehaviorConstraint`) and one target. These two source classes were already grouped into the `AttributeConstraint` abstract class just above, so this class is reused and is defined as the source of the `constrains` relationship. The same observation can be made for the `uses` relationship, whose source is defined as the `AttributeConstraint` class and the target is the `ParameterConstraintParticipant`.
- The `playsRoleOf` relationship case is more complex: it has two source classes (`StateVariableConstraintParticipant` and `ParameterConstraintParticipant`) and four target classes (`StateVariable`, `StateVariableSurrogate` (optional), `Parameter` and `ParameterSurrogate` (optional)). In the case of the source classes, a new abstract superclass is introduced, named `AttributeConstraintParticipant`. In the case of the four target classes, a previous grouping can be reused (featuring `Property`, `PropertySurrogate` (optional) and `Attribute` shown in Figure 12). Range restrictions were not defined in this case due to the complexity of the

restriction expression: it will instead be handled by validation rules that will check that the model only has relationships that fall only in the four cases presented in the upper panel of Figure 17. The **optional classes related to InteractionTerminal** (see discussion in the previous section) **are grayed out**, and this visual representation will be used in the remainder of this page.

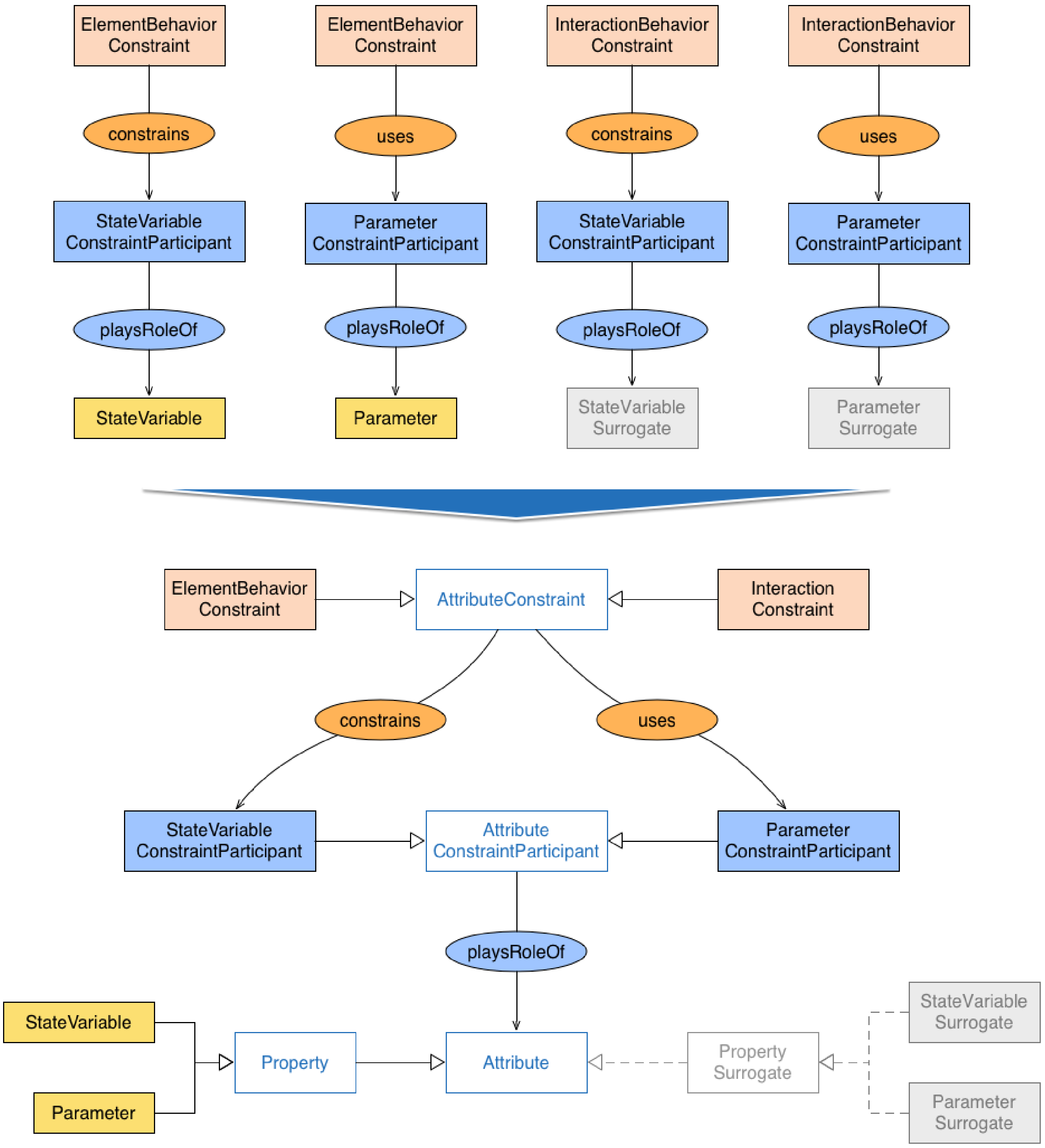
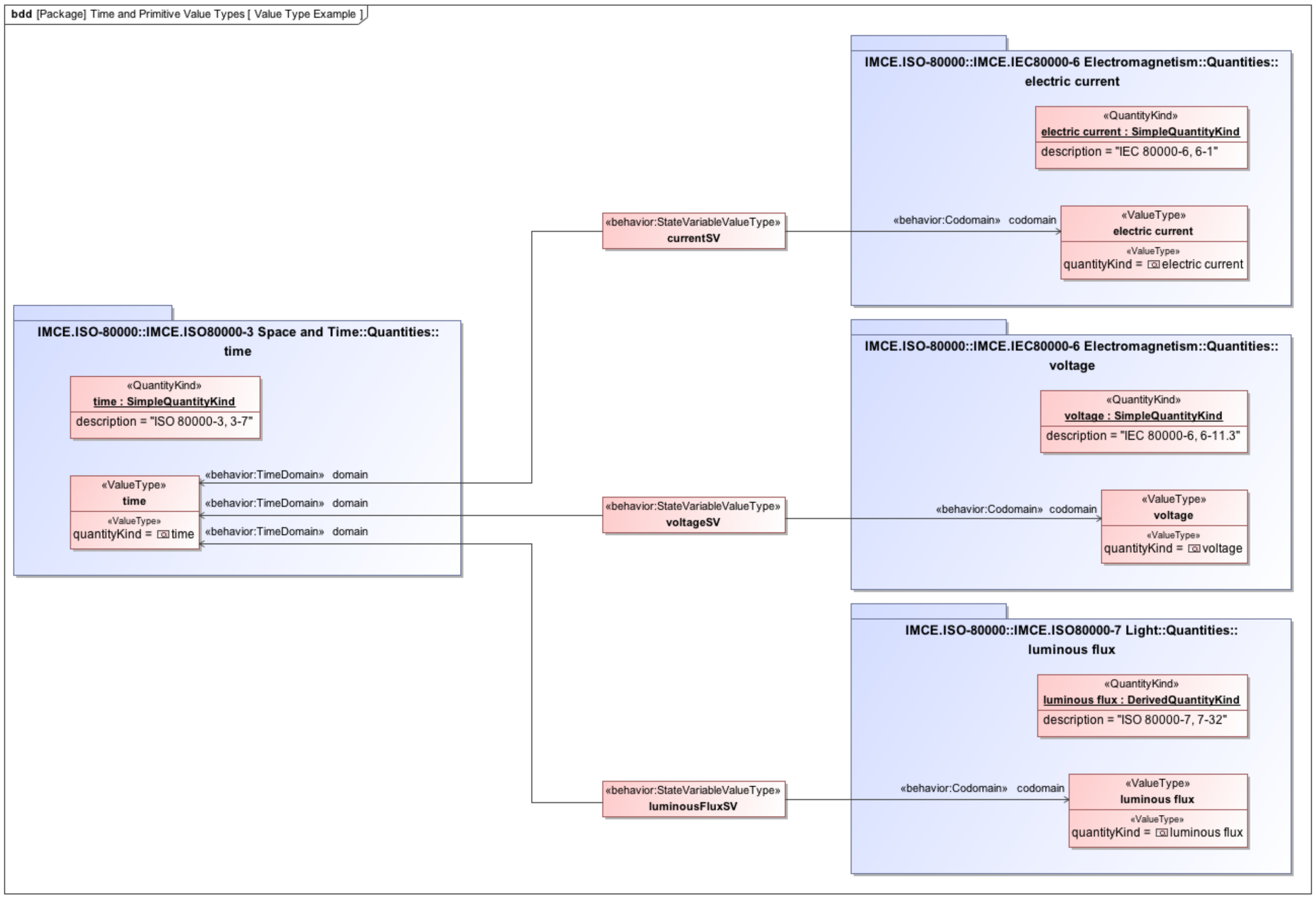

ElementBehavior Constraint
constrains
StateVariable ConstraintParticipant
playsRoleOf
StateVariable
ElementBehavior Constraint
uses
Parameter ConstraintParticipant
playsRoleOf
Parameter
InteractionBehavior Constraint
constrains
StateVariable ConstraintParticipant
playsRoleOf
StateVariable Surrogate
InteractionBehavior Constraint
uses
Parameter ConstraintParticipant
playsRoleOf
Parameter Surrogate
ElementBehavior Constraint
AttributeConstraint
Interaction Constraint
constrains
uses
StateVariable ConstraintParticipant
Attribute ConstraintParticipant
Parameter ConstraintParticipant
playsRoleOf
StateVariable
Parameter
Property
Attribute
Property Surrogate
StateVariable Surrogate
Parameter Surrogate

Figure 17. `playsRoleOf` embedding

StateVariable, Parameter and Value Types

As explained in the conceptual ontology, the `StateVariable` is defined as having a `TimeDomain` and a `Codomain`, as shown in Figure 19. This is captured in the SysML embedding by using Value Types. The `StateVariable` value property is typed by a specific value type named `StateVariableValueType`. This value type in turn owns two value properties: a `TimeDomain` one and a `Codomain` one. Each of these value properties are typed by distinct value types respectively named `TimeDomainValueType` and `AttributeCodomainValueType` (note that the latter is greyed in Figure 18 as it will be expanded upon later in this page). As a consequence, the `StateVariableValueType` is a structured value type.

Figure 18 shows a simple example for generic `StateVariables` such as Voltage, Current or LuminousFlux. Note that the Value Types of the domain and codomain properties are not `TimeDomainValueType` and `AttributeCodomainValueType`, as they are extracted from the ISO-80000 library. One could specialize these library Value Types for their components, and type the specialized `StateVariable`'s `TimeDomain` and `Codomain` with these. **Guidance on how to handle Value Types is under evaluation and Value Type modeling may be part of a separate pattern. This section is a first attempt at capturing Value Types from a behavior perspective and might evolve in future iterations**.



bdd [Package] Time and Primitive Value Types [Value Type Example]
IMCE.ISO-80000::IMCE.ISO80000-3 Space and Time::Quantities::time
«QuantityKind»
time : SimpleQuantityKind
description = "ISO 80000-3, 3-7"
«ValueType»
time
«ValueType»
quantityKind = time
«behavior:TimeDomain» domain
«behavior:TimeDomain» domain
«behavior:TimeDomain» domain
«behavior:StateVariableValueType»
currentSV
«behavior:StateVariableValueType»
voltageSV
«behavior:StateVariableValueType»
luminousFluxSV
IMCE.ISO-80000::IMCE.IEC80000-6 Electromagnetism::Quantities::electric current
«QuantityKind»
electric current : SimpleQuantityKind
description = "IEC 80000-6, 6-1"
«behavior:Codomain» codomain
«ValueType»
electric current
«ValueType»
quantityKind = electric current
IMCE.ISO-80000::IMCE.IEC80000-6 Electromagnetism::Quantities::voltage
«QuantityKind»
voltage : SimpleQuantityKind
description = "IEC 80000-6, 6-11.3"
«behavior:Codomain» codomain
«ValueType»
voltage
«ValueType»
quantityKind = voltage
IMCE.ISO-80000::IMCE.ISO80000-7 Light::Quantities::luminous flux
«QuantityKind»
luminous flux : DerivedQuantityKind
description = "ISO 80000-7, 7-32"
«behavior:Codomain» codomain
«ValueType»
luminous flux
«ValueType»
quantityKind = luminous flux

Figure 18. Value Type Example

The fact that a value property is typed by a value type is captured in the embedded ontology by the `hasValueType` relationship (embedded by the appropriate UML concept, A_type_typedElement). The `StateVariableValueType` owns two value properties and this is captured through the `hasFeature` relationship. Once again to conform to the single range class expression pattern, the introduction of abstract superclasses is necessary as the `hasValueType` and the `hasFeature` have several sources or targets. In the case of `hasValueType`, all the value properties that are typed (`StateVariable`, `TimeDomain` and `Codomain`) are grouped into the `AbstractValueTypedAttribute` class, and all the value types (`StateVariableValueType`, `TimeDomainValueType` and `AttributeCodomainValueType`) are grouped into the `AbstractValueType` class. The `hasValueType` relationship is now defined as having `AbstractValueTypedAttribute` as source and `AbstractValueType` as target. Range restrictions are defined as appropriate and are shown in Figure 18. In the case of the `hasFeature` relationship, the abstract superclass `AbstractValueTypedFeature` is created and groups the `TimeDomain` and `Codomain` value properties.

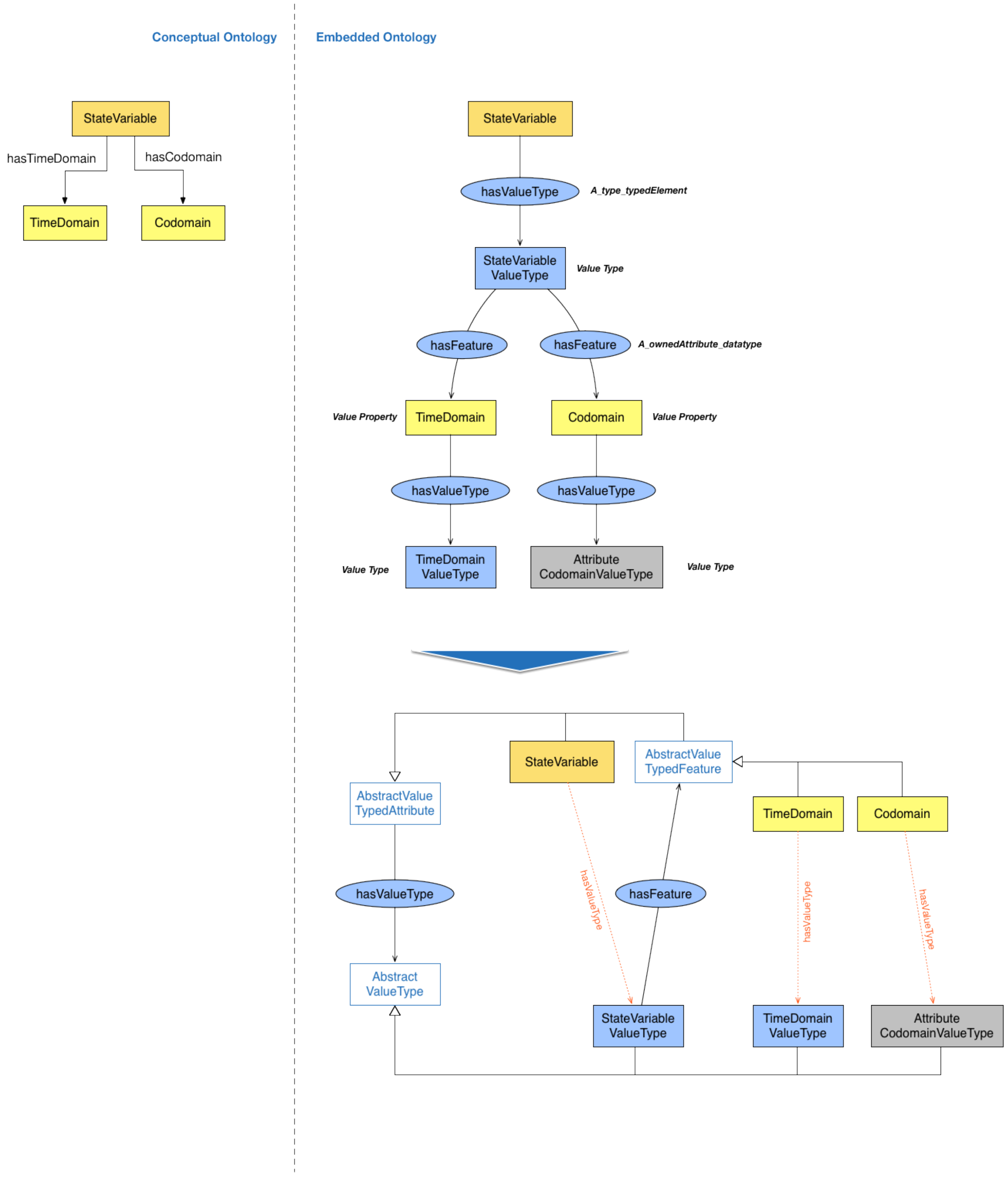


Figure 19. `StateVariable`'s `TimeDomain` and `Codomain`

There are however cases where the nature of the `StateVariable` calls for a different way of capturing its type. For example, the case of a discrete `StateVariable` with a small number of `States`, such as the switch position `StateVariable` in the flashlight example that can either be in the open or closed `States`. It is more natural then to capture these `States` using a State Machine (that has region and states). The `Codom ain` of the `StateVariable` is defined by the configuration of the State Machine (i.e., the combination of the active states in different regions). In the case of a single region, the configuration of the state machine and the currently active state are confounded. The state machine is also used to capture the behavior of the `BehavingElement` associated with the `StateVariable`, and as such the state machine is owned by the `Attrib` uteConstraintOwner class, as shown in Figure 20. Figure 22 shows specifically the value type of the switch position state variable. This case was already touched upon in Figure 9.

Different constraints might be applicable when in different states (such as in the case of the switch position: an open switch constrains the current to be zero, while a closed switch constrains the voltage to be zero), and this is captured by linking the appropriate constraint blocks to the states using the analysis:characterizes relationship.The analysis:characterizes relationship points at the AutomatonScope concept, specialized by AutomatonRegion and AutomatonState. This gives the possibility to the model to constraint several States at once by pointing at a state machine region.

To go back to the original point regarding the fact that the state machine could type a state variable, or more exactly its Codomain, the AttributeCodomainValueType is expanded in Figure 20 into two child classes, one of which is the AttributeAutomatonConstraint that captures the state machines.

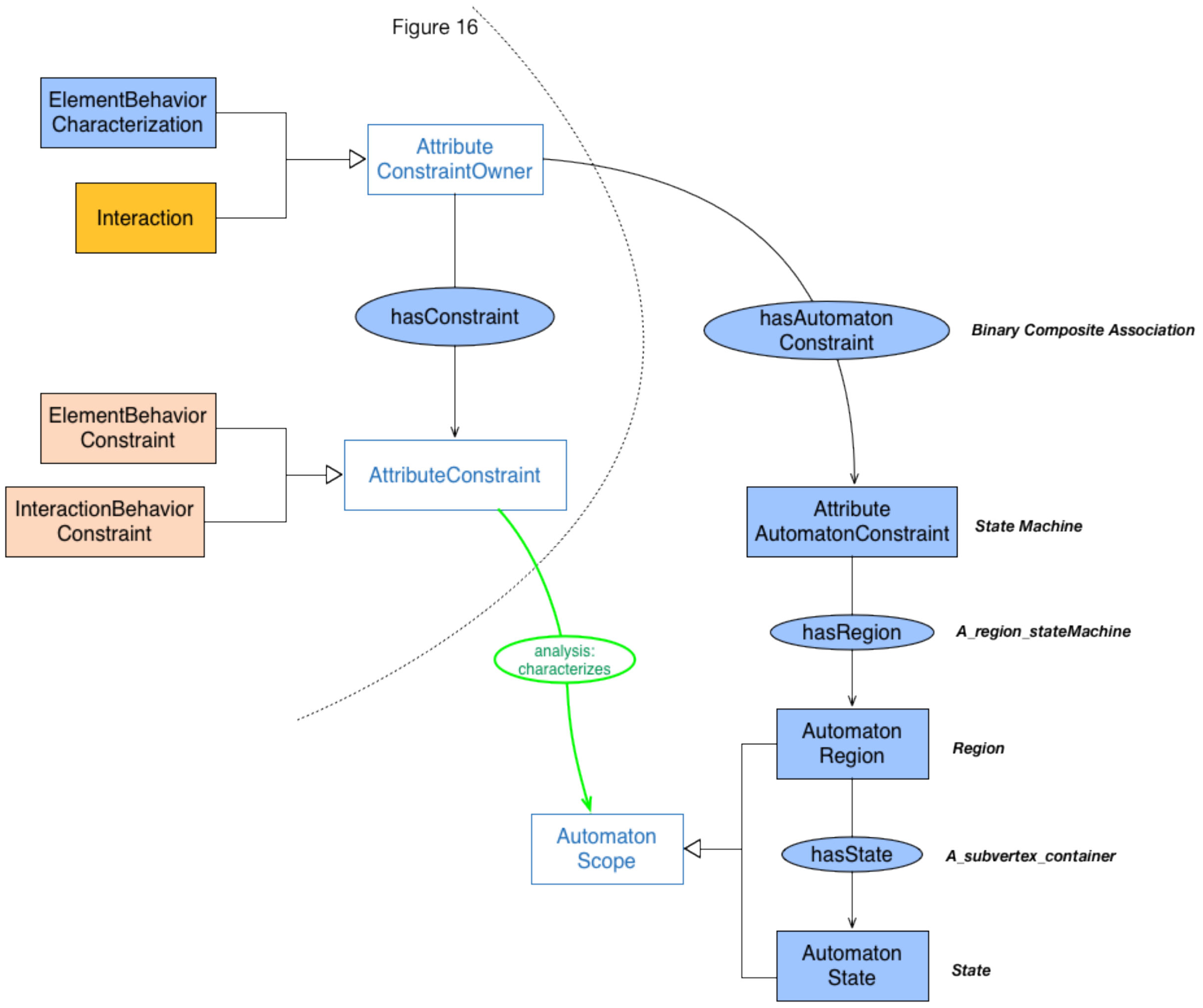


Figure 20. State machines

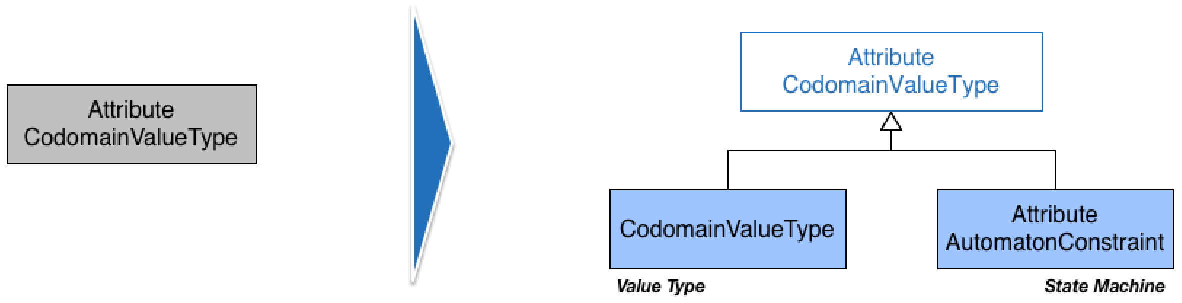

Attribute CodomainValueType
Attribute CodomainValueType
CodomainValueType
Attribute AutomatonConstraint
Value Type
State Machine

Figure 21. `Codomain` value types

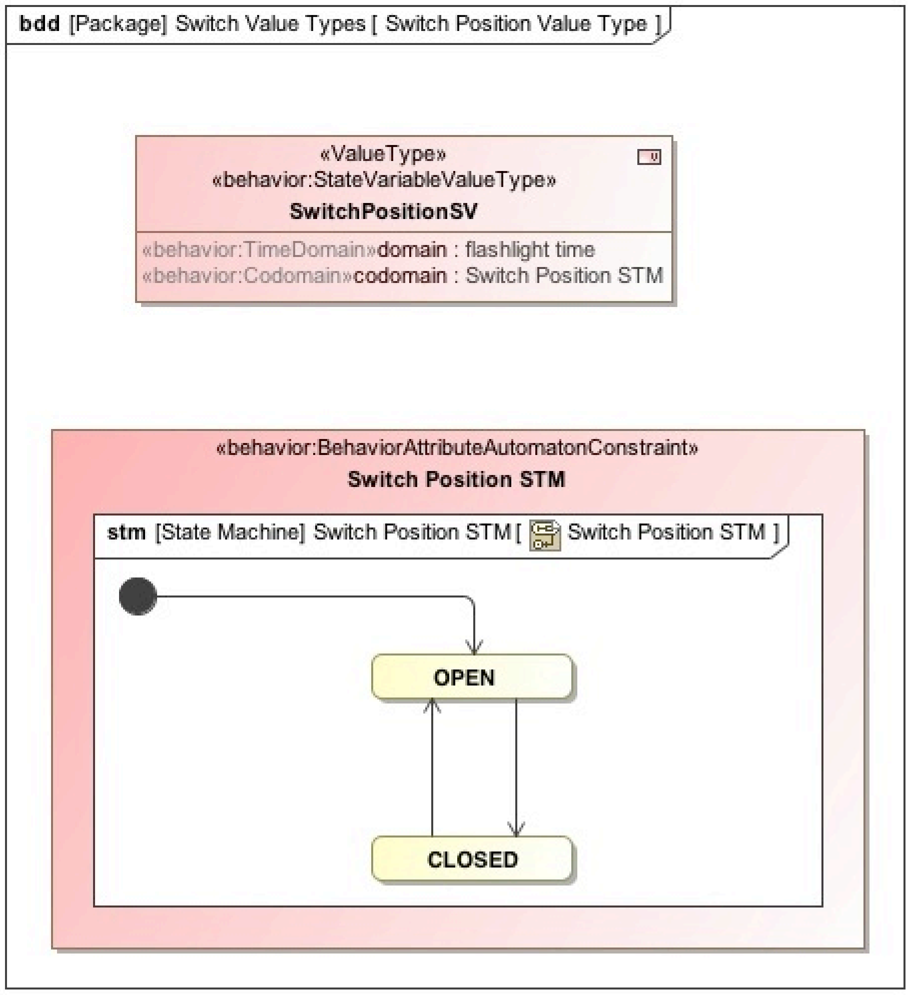

bdd [Package] Switch Value Types [Switch Position Value Type]
«ValueType»
«behavior:StateVariableValueType»
SwitchPositionSV
«behavior:TimeDomain»domain : flashlight time
«behavior:Codomain»codomain : Switch Position STM
«behavior:BehaviorAttributeAutomatonConstraint»
Switch Position STM
stm [State Machine] Switch Position STM [Switch Position STM]
OPEN
CLOSED

Figure 22. Switch position value type

To conclude on the topic of value type, the `Parameter` value properties can also be typed by their specific value type, namely `ParameterValue Type`. This is shown in Figure 23. Figure 24 a synthesis of all the classes and relationships discussed in this section, with the addition of another class, named `QuantityValueType` (extreme right of the figure) that specializes the `CodomainValueType` and the `ParameterValueType` cla sses. This value type class is introduced to allow for flexibility in capturing value types in model and particularly to allow the reuse of the same value type for typing `Parameters` and `Codomains` of `StateVariables`, when appropriate (there are cases when this reuse is not possible, such as when the quantity kind associated with the value type is specific to some behavior properties). The modeler will have the choice to reuse value types from libraries such as the ISO 80000, or define their own. Examples of defining specific value types are given in the flashlight SysML model and the interested reader is referred to the MagicDraw file. <<<insert link/reference>>>

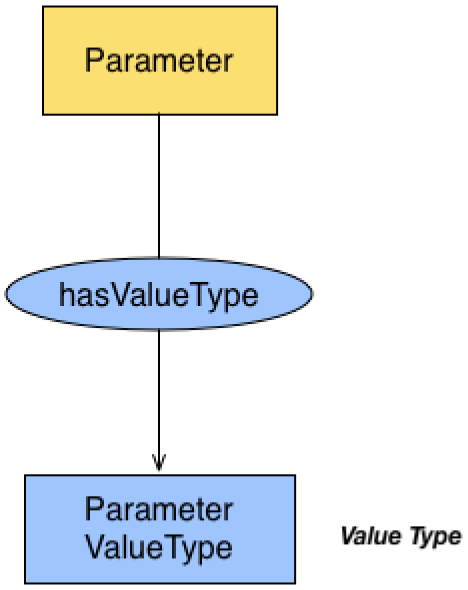


Figure 23. Behavior attribute `Codomain`

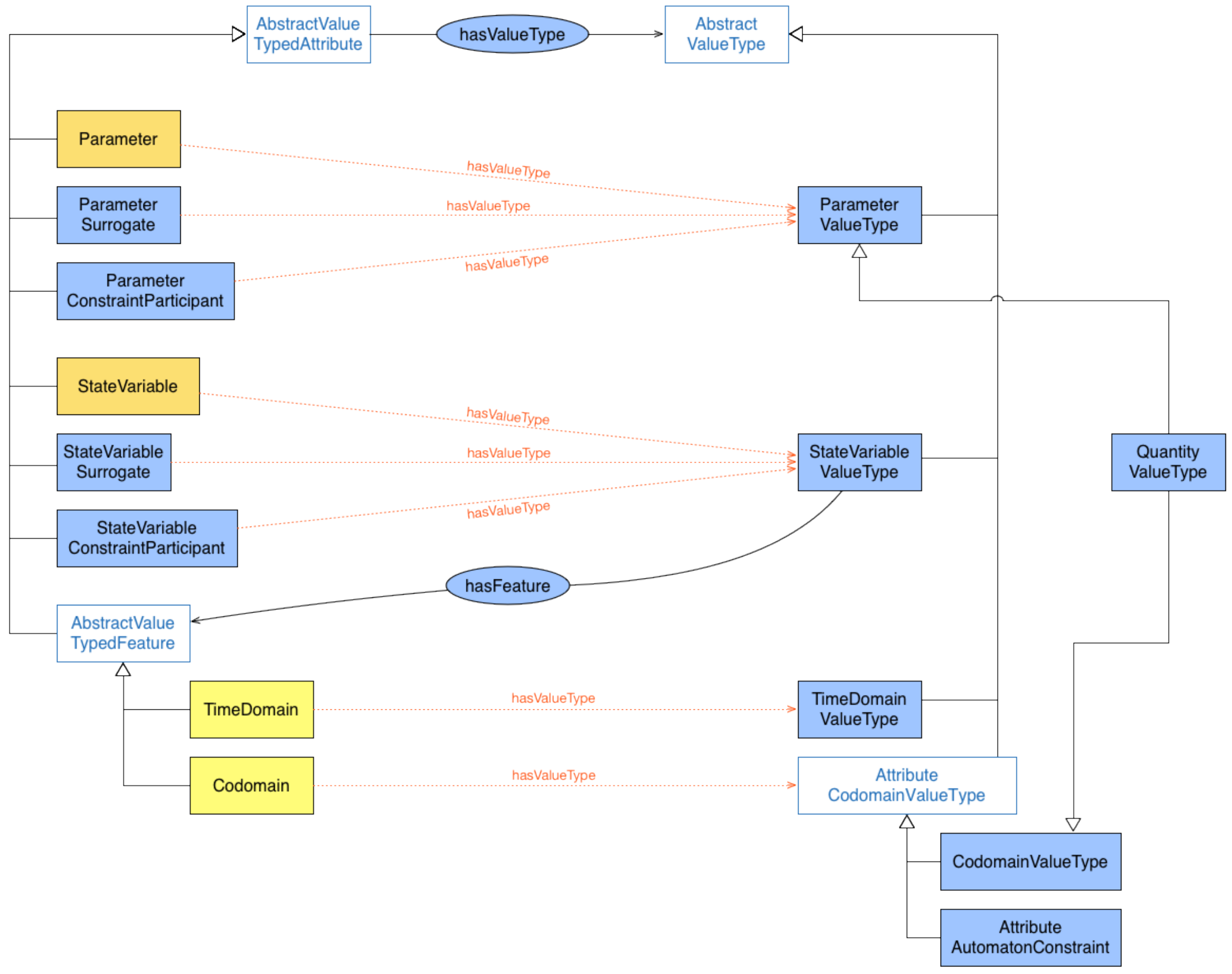


Figure 24. Value type synthesis

Complete embedded ontology

Ontology map

Figure 25 shows the complete embedded ontology map that was step-by-step described in this page. As explained before, range restrictions are displayed using red dotted lines, and the optional concepts related to InteractionTerminal are grayed out.

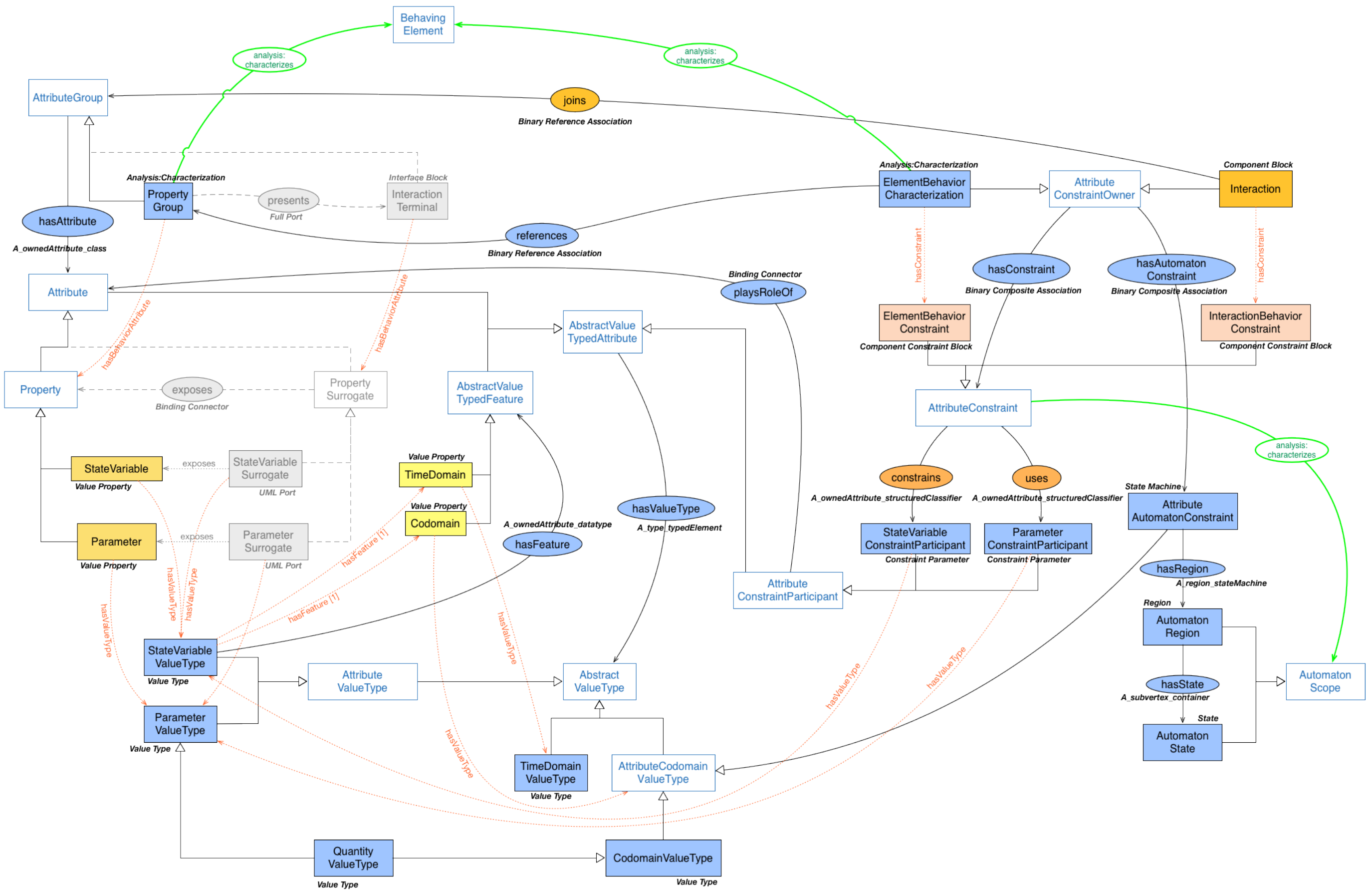


Figure 25. Complete embedded ontology

Optional usage of stereotypes in the SysML model

Several of the concepts introduced here are necessary for the definition of the ontology, but their usage can be optional in the model, as the information they would provide is already unambiguously provided by the combination of other stereotypes, or as the elements that should be stereotyped do not have that capability. This in turn alleviate the number of stereotypes the modeler needs to use for capturing behavior.

- `presents`: optional use, as the port's meaning is unambiguously defined between a `PropertyGroup` and an `InteractionTerminal`
- `exposes`: optional use, unambiguous context provided by `Properties` and `InteractionTerminal`
- `playsRoleOf`: optional use, as the binding connector's meaning is unambiguously defined between a `AttributeConstraintParticipant` and a `Attribute`
- `hasConstraint`: optional use, as the association's meaning is unambiguously defined between a `AttributeConstraintOwner` and a `AttributeConstraint`
- `hasAutomatonConstraint`: optional use, as the association's meaning is unambiguously defined between a `AttributeConstraintOwner` and a `AttributeAutomatonConstraint`
- `AutomatonRegion` and `AutomatonState`: optional use, as the context is unambigously defined by the `AttributeAutomatonConstraint`

- `constrains`: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_structuredClassifier. Relationship unambiguously defined from a `AttributeConstraint` to a `StateVariableConstraintParticipant`
- `uses`: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_structuredClassifier. Relationship unambiguously defined from a `AttributeConstraint` to a `ParameterConstraintParticipant`
- `hasAttribute`: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_class. Relationship unambiguously defined from a `AttributeGroup` to a `Attribute`
- `hasFeature`: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_datatype. Relationship unambiguously defined from a `StateVariableValueType` to an `AbstractValueTypedFeature`
- `hasRegion` and `hasState`: stereotype cannot be applied due to the SysML specification of A_region_stateMachine and A_subvertex_container.
- `hasValueType`: stereotype cannot be applied due to the SysML specification of A_type_typedElement. Relationship unambiguously

defined from a `AbstractValueTypedFeature` to an `AbstractValueType`

If using `InteractionTerminal`:

- `StateVariableSurrogate` and `ParameterSurrogate`: optional use, as meaning is provided by the `InteractionTerminal` and binding connector to `Properties`

Summary tables for all non-abstract concepts and relationships

- from the original Conceptual ontology

Concept	SysML embedding
`Parameter`	Value property
`Codomain`	Value property
`ElementBehavior`	*no direct embedding – see `ElementBehaviorCharacterization` and `ElementBehaviorConstraint`*
`InteractionBehavior`	*embedded as `InteractionBehaviorConstraint` – see `InteractionBehaviorConstraint`*
`Interaction`	Component Block
`State`	*no direct embedding – see `AutomatonState`*
`StateVariable`	Value property
`TimeDomain`	Value property
`*InteractionTerminal`	Interface Block (optional use)
Relationship	**SysML embedding**
`constrains`	A_ownedAttribute_structuredClassifier – *represents the UML association between a StructuredClassifier and its owned attributes*
`exposes`	Binding connector
`isDescribedBy`	represented by `hasConstraint`, composite association ("black diamond")
`isElementOf`	*no direct embedding – convention on the configuration of the State Machine typing the `StateVariable`'s `Codomain`*
`joins`	Association Block
`uses`	A_ownedAttribute_structuredClassifier – *represents the UML association between a StructuredClassifier and its owned attributes*
`*presents`	Full port (optional use)

`Scenario`, `FamilyOfTrajectories`, `Trajectory`, `belongsTo` and `prescribes` are not embedded in this pattern, as their full specification is part of an upcoming pattern.

- introduced in the SysML-embeddable ontology

Concept	SysML embedding
`AttributeAutomatonConstraint`	State Machine
`AutomatonRegion`	Region
`AutomatonState`	State

CodomainValueType	Value type
ElementBehaviorCharacterization	analysis:Characterization (Component Block)
ElementBehaviorConstraint	Component Constraint Block
InteractionBehaviorConstraint	Component Constraint Block
InteractionModel	analysis:Analysis (Component Block)
ParameterConstraintParticipant	Constraint parameter
ParameterValueType	Value type
PropertyGroup	analysis:Characterization (Component Block)
QuantityValueType	Value type
StateVariableConstraintParticipant	Constraint parameter
StateVariableValueType	Value type
TimeDomainValueType	Value type
*ParameterSurrogate	UML port (optional use)
*StateVariableSurrogate	UML port (optional use)
Relationship	**SysML embedding**
hasAttribute	A_ownedAttribute_class – *represents the UML association between a Class and the attributes owned by that Class*
hasAutomatonConstraint	Composite association ("black diamond")
hasConstraint	Composite association ("black diamond")
hasFeature	A_ownedAttribute_datatype – *represents the UML association betwe en a DataType and its owned attributes*
hasRegion	A_region_stateMachine – *represents the UML association between a State Machine and its region*
hasState	A_subvertex_container – *represents the UML association between a region and its state*
hasValueType	A_type_typedElement – *represents the UML association between a TypedElement and its Type*
playsRoleOf	Binding connector
references	Shared association ("white diamond")

Relation multiplicity table

See Concept ontology page for rationales driving the multiplicities.

Subject	Verb	Multiplicity	Object	Reverse relation multiplicity
AttributeConstraint	analysis:characterizes	[0..*]	AutomatonScope	[0..*]
ElementBehaviorCharacterization	analysis:characterizes	[0..1]	BehavingElement	[0..*]
PropertyGroup	analysis:characterizes	[0..1]	BehavingElement	[0..*]
AttributeConstraint	constrains	[0..*]	StateVariableConstraintParticipant	[0..1]

AttributeGroup	hasAttribute	[0..*]	Attribute	[0..1]
AttributeConstraintOwner	hasAutomatonConstraint	[0..*]	AttributeAutomatonConstraint	[0..1]
AttributeConstraintOwner	hasConstraint	[0..*]	AttributeConstraint	[0..1]
StateVariableValueType	hasFeature	[0..1]	TimeDomain	[0..1]
StateVariableValueType	hasFeature	[0..1]	Codomain	[0..1]
AttributeAutomatonConstraint	hasRegion	[0..*]	AutomatonRegion	[0..1]
AutomatonRegion	hasState	[0..*]	AutomatonState	[0..1]
AbstractValueTypedAttribute	hasValueType	[0..1]	AbstractValueType	[0..*]
Interaction	joins	[0..*]	AttributeGroup	[0..*]
AttributeConstraintParticipant	playsRoleOf	[0..1]	Attribute	[0..*]
ElementBehaviorCharacterization	references	[0..*]	PropertyGroup	[0..*]
AttributeConstraint	uses	[0..*]	ParameterConstraintParticipant	[0..1]
*PropertySurrogate	*exposes	[0..1]	Property	[0..*]
PropertyGroup	*presents	[0..*]	*InteractionTerminal	[0..1]

SysML Example

The complete flashlight example presented piece-by-piece above is shown here.

Model Implementation Concerns

None for now.

Supporting Scripts/Tooling

None for now, but supporting tooling appears necessary to assist the modeler in creating behavior models.

Tooling Tricks

None for now.

Behavior Page Navigation - continue reading:

(0) Community Page

(1) Main Behavior Pattern Page

(2) Conceptual Behavior Ontology v1.1 (3) Behavior Pattern: Conceptual Examples v1.1

(4) SysML-Embeddable Ontology & Implementation v1.1 (5) Behavior Pattern: SysML Example v1.1