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 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.

- SysML Implementation
  - SysML-Embeddable Behavior Ontology - Description
    - State variable
    - Parameter
    - ElementBehavior
      - Mathematical constraints
      - State machines
    - Interaction aspects
      - Interaction
      - Interaction using InteractionTer minal (optional)
    - Synthesis of constraints
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 represents 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](image)

![Figure 4. Battery and Lamp Properties](image)

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 \textbf{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 \textbf{ElementBehavior} makes a distinction between the constraints and their owner. For this purpose, two classes are introduced: the \textbf{ElementBehaviorConstraint} is a (Component) Constraint Block, while the \textbf{ElementBehaviorCharacterization} is an Analysis:Characterization that owns the potentially many \textbf{Constraint Blocks}. The relationship between the \textbf{ElementBehaviorCharacterization} and the \textbf{ElementBehaviorConstraint}, named \textbf{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 \textbf{A\_ownedAttribute\_structuredClassifier}, the appropriate UML association in this case). The class that represents these constraint parameter is named \textbf{StateVariableConstraintParticipant}. In the constraint, the participants "play the role" of the relevant state variables; this relationship is captured through the \textbf{playsRoleOf} relationship that is embedded as a binding connector between \textbf{StateVariableConstraintParticipant} and \textbf{StateVariable}. This choice allows for potential reuse of the Constraint Block and their addition to libraries.

---

\textbf{Figure 5. ElementBehavior and StateVariables}

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

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 ElementBehaviorCharacterization 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 analysis:characterizes dependency to the CLOSED state. The constraint on the voltage for the OPEN state is captured using an ElementBehaviorConstraint that points 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.
Figure 9. Switch ElementBehavior
**Interaction aspects**

**Interaction**

Another constraint type for StateVariable is in the context of an Interaction. The right part 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 InteractionBehaviorConstraint (embedded as Component Constraint Block). These constraint blocks in turn own (through the constrains relationship) StateVariableConstraint Participants that are related to StateVariables using binding connectors stereotyped by playsRoleOf. The InteractionBehaviorConstraint 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
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 cases for the hasConstraint relationship. This visual representation will be used in the remainder of this page.
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.
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.

The fact that a value property is typed by a value type is captured in the embedded ontology by the has ValueType relationship (embedded by the appropriate UML concept, AType_typedElement). The StateVariableValueType owns two value properties and this is captured through the hasValueType relationship. Once again to conform to the single range class expression pattern, the introduction of abstract superclasses is necessary as the StateVariable and the TimeDomain have several sources or targets. In the case of the StateVariable, all the value properties that are typed (StateVariable, TimeDomain and Codomain) are grouped into the AbstractValueType class, and all the value types (StateVariableValueType, TimeDomainValueType and AttributeCodomainValueType) are grouped into the AbstractValueTypedAttribute class. The has ValueType 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 has Feature 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 Codomain 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 Attribute eCodomainValueType is expanded in Figure 20 into two child classes, one of which is the AttributeAutomatonConstraint that captures the state machines.
To conclude on the topic of value type, the Parameter value properties can also be typed by their specific value type, namely ParameterValueType. 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 classes. 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.
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 an InteractionTerminal and a PropertyGroup
- **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 unambiguously defined by the AttributeAutomatonConstraint

- **constrains**: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_structuredClassifier. Relationship unambiguously defined from an AttributeConstraint to a StateVariableConstraintParticipant
- **uses**: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_structuredClassifier. Relationship unambiguously defined from an AttributeConstraint to a ParameterConstraintParticipant
- **hasAttribute**: stereotype cannot be applied due to the SysML specification of A_ownedAttribute_class. Relationship unambiguously defined from an 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

<table>
<thead>
<tr>
<th>Concept</th>
<th>SysML embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Value property</td>
</tr>
<tr>
<td>Codomain</td>
<td>Value property</td>
</tr>
<tr>
<td>ElementBehavior</td>
<td><em>no direct embedding</em> — see <code>ElementBehaviorCharacterization</code> and <code>ElementBehaviorConstraint</code></td>
</tr>
<tr>
<td>InteractionBehavior</td>
<td><em>embedded as <code>InteractionBehaviorConstraint</code></em> — see <code>InteractionBehaviorConstraint</code></td>
</tr>
<tr>
<td>Interaction</td>
<td>Component Block</td>
</tr>
<tr>
<td>State</td>
<td><em>no direct embedding</em> — see <code>AutomatonState</code></td>
</tr>
<tr>
<td>StateVariable</td>
<td>Value property</td>
</tr>
<tr>
<td>TimeDomain</td>
<td>Value property</td>
</tr>
<tr>
<td>*InteractionTerminal</td>
<td>Interface Block (optional use)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationship</th>
<th>SysML embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>constrains</td>
<td><code>A_ownedAttribute_structuredClassifier</code> — represents the UML association between a StructuredClassifier and its owned attributes</td>
</tr>
<tr>
<td>exposes</td>
<td>Binding connector</td>
</tr>
<tr>
<td>isDescribedBy</td>
<td>represented by <code>hasConstraint</code>, composite association (<em>black diamond</em>)</td>
</tr>
<tr>
<td>isElementOf</td>
<td><em>no direct embedding</em> — convention on the configuration of the State Machine typing the <code>StateVariable's Codomain</code></td>
</tr>
<tr>
<td>joins</td>
<td>Association Block</td>
</tr>
<tr>
<td>uses</td>
<td><code>A_ownedAttribute_structuredClassifier</code> — represents the UML association between a StructuredClassifier and its owned attributes</td>
</tr>
<tr>
<td>*presents</td>
<td>Full port (optional use)</td>
</tr>
</tbody>
</table>

`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

<table>
<thead>
<tr>
<th>Concept</th>
<th>SysML embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeAutomatonConstraint</td>
<td>State Machine</td>
</tr>
<tr>
<td>AutomatonRegion</td>
<td>Region</td>
</tr>
<tr>
<td>AutomatonState</td>
<td>State</td>
</tr>
</tbody>
</table>
**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

*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 between 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.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
<th>Multiplicity</th>
<th>Object</th>
<th>Reverse relation multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeConstraint</td>
<td>analysis:characterizes</td>
<td>[0..*]</td>
<td>AutomatonScope</td>
<td>[0..*]</td>
</tr>
<tr>
<td>ElementBehaviorCharacterization</td>
<td>analysis:characterizes</td>
<td>[0..1]</td>
<td>BehavingElement</td>
<td>[0..*]</td>
</tr>
<tr>
<td>PropertyGroup</td>
<td>analysis:characterizes</td>
<td>[0..1]</td>
<td>BehavingElement</td>
<td>[0..*]</td>
</tr>
<tr>
<td>AttributeConstraint</td>
<td>constrains</td>
<td>[0..*]</td>
<td>StateVariableConstraintParticipant</td>
<td>[0..1]</td>
</tr>
</tbody>
</table>
**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