



Reference  
Card

## SCADE Metamodels

*SCADE® 2025 Products Family*

---

# SCADE Metamodels Overview

This reference card describes the metamodels available for each SCADE product as UML notation diagrams. It is intended to advanced users who need access to SCADE models for integrating tools customized with these models. All API functions for navigating and exploiting these metamodels are illustrated in *SCADE Python API Guide* and in *SCADE Tcl/JAVA API Guide*. The Python API and the Tcl API available for some SCADE products are accessible in GUI (using model in memory) or in batch. The Java API of each SCADE product is compliant with the Eclipse Modeling Framework (EMF). Such EMF-based APIs can be used through generic model-manipulation tools working at EMF-level.

## SCADE ARCHITECT METAMODELS

Relying on the metamodels in this document, SCADE Architect API is a standalone read/write API which allows manipulating SCADE Architect model files. A set of metamodels is available for developing via SCADE Architect API using Python, Java, Tcl, or OCL scripting technologies. See Part 1: ["SCADE Architect Metamodels"](#)

## SCADE SUITE METAMODELS

Relying on the metamodels in this document, SCADE Suite API is a standalone read/write API which allows manipulating SCADE Suite model files. A set of metamodels is available for developing via SCADE Python API, SCADE Suite Tcl API, or SCADE Suite Java API.

See Part 2: ["SCADE Suite Metamodels for Python API"](#), Part 3: ["SCADE Suite Metamodels for Tcl API"](#), or Part 4: ["SCADE Suite Metamodels for Java API"](#)

## SCADE DISPLAY METAMODELS

Relying on the metamodels in this document, SCADE Display API is a standalone read/write API which allows manipulating SCADE Display model files. The SCADE Display API can be used through SCADE Python API, through direct programming in Java, or through generic model-manipulation tools working at EMF-level. See Part 5: ["SCADE Display Metamodels"](#)

## SCADE UA PAGE CREATOR METAMODELS

Relying on the metamodels in this document, SCADE UA Page Creator API is a standalone read/write API which allows manipulating the SCADE UA Page Creator model files and the common standard configuration used on all the A661 modules. The SCADE UA Page Creator API can be used through SCADE Python API, through direct programming in Java, or through generic model-manipulation tools working at EMF-level. See Part 6: ["SCADE UA Page Creator Metamodels"](#)

## SCADE TEST METAMODELS

Relying on the metamodels in this document, SCADE Test API is a standalone read/write API which allows manipulating SCADE Test project files. A set of metamodels is available for developing via SCADE Python API or SCADE Test Tcl API. A procedure metamodel is also available for developing via SCADE Test Java API.

See Part 7: ["SCADE Test Metamodels"](#)

## SCADE ALM GATEWAY METAMODELS

See Part 8: ["SCADE Traceability Metamodels for Python API"](#)

# Contents

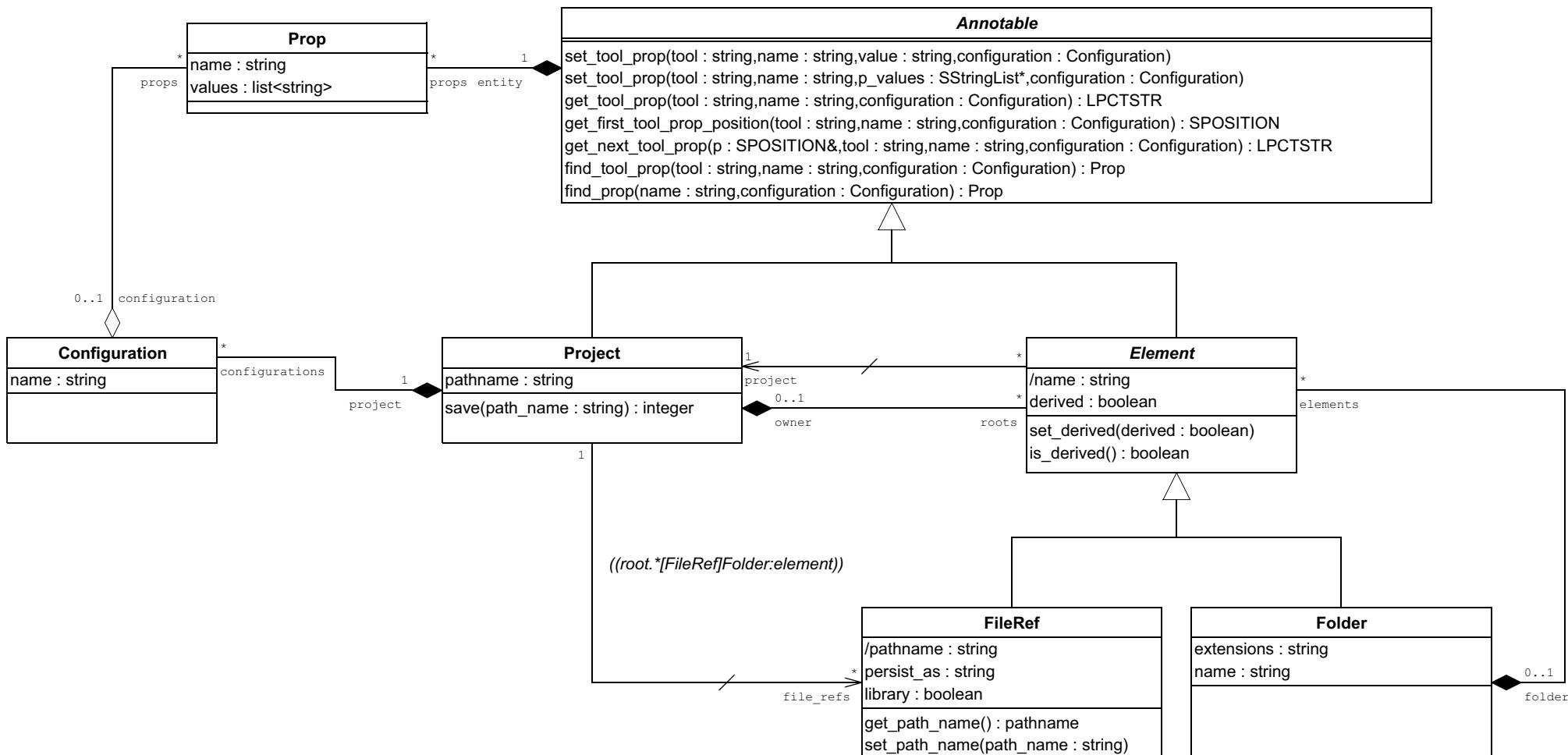
SCADE Metamodels Overview		
SCADE Projects Metamodel		
<b>Part 1: SCADE Architect Metamodels</b>		
General Construct Metamodels		
Data Types Metamodel		
Blocks Structure Metamodel		
Dataflow-Oriented Communications Metamodels		
Allocations Metamodel		
Diagrams, Tables, and Traceability Metamodels		
Behaviors Metamodels		
Synchronization Metamodel		
RichText Metamodel		
<b>Part 2: SCADE Suite Metamodels for Python API</b>		
Editor Metamodels (Python API)		
Annotation Metamodels (Python API)		
Timing and Stack Analysis Metamodel (Python API)		
Graphical Panel Coupling Metamodel (Python API)		
<b>i Part 3: SCADE Suite Metamodels for Tcl API</b>		
1 Editor Metamodels (Tcl API)		97
5 Annotation Metamodels (Tcl API)		118
18 Timing and Stack Analysis Metamodel (Tcl API)		122
<b>18 Part 4: SCADE Suite Metamodels for Java API</b>		
24 Scade Language Metamodels		124
34 Scade Graphics Metamodels		142
<b>38 Part 5: SCADE Display Metamodels</b>		
39 SCADE Display Metamodels		146
47 SCADE Display Mapping Files Metamodels		162
<b>62 Part 6: SCADE UA Page Creator Metamodels</b>		
63 SCADE UA Page Creator Metamodels		167
<b>65 Part 7: SCADE Test Metamodels</b>		
86 Test Environment for Host Metamodels (Tcl)		179
90 Test Environment for Host Metamodels (Python)		184
<b>91 Part 8: SCADE Traceability Metamodels for Python API</b>		
91 SCADE Traceability Metamodels		191

# 1 / SCADE Projects Metamodel

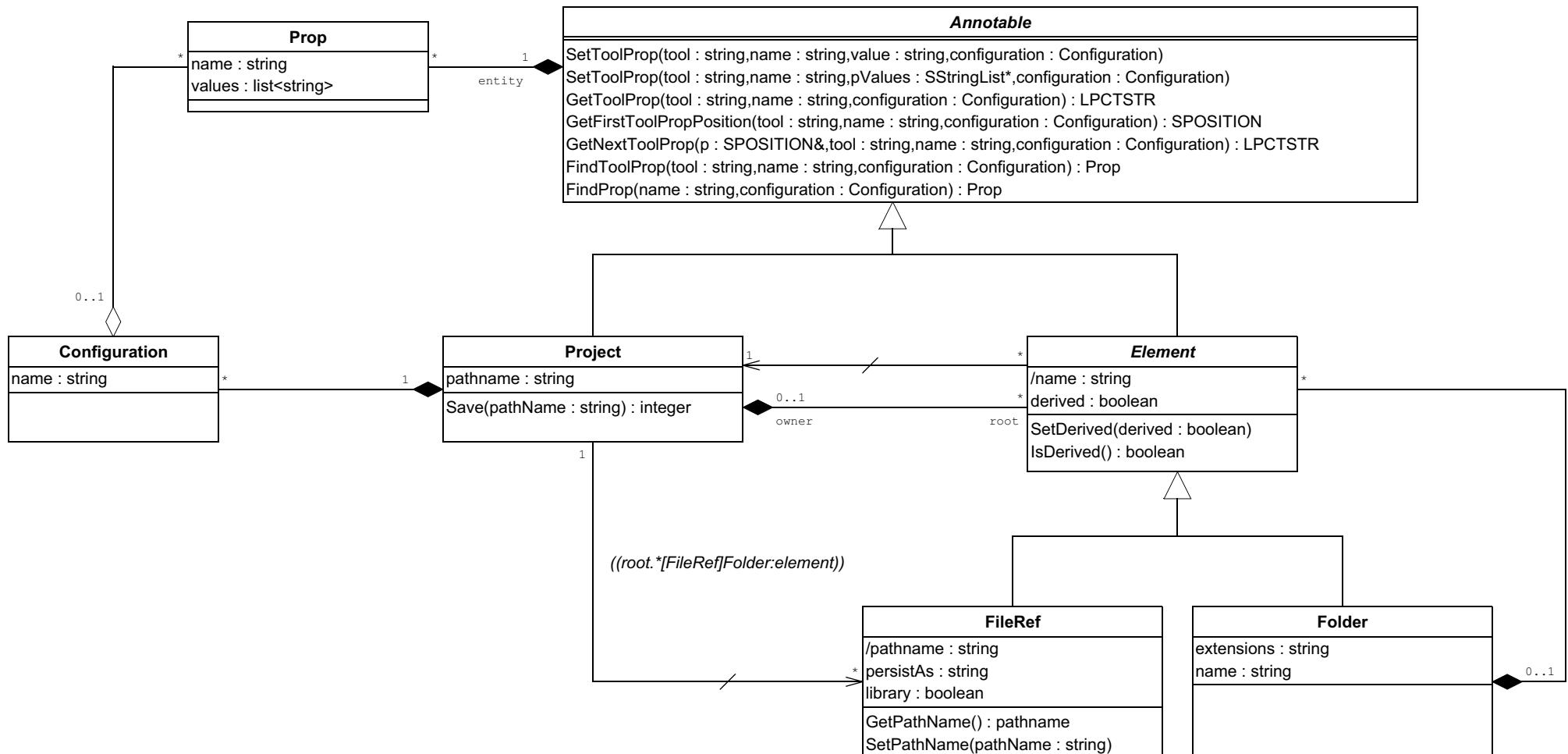
This metamodel presents the data structure that gives access to any SCADE project using SCADE Python API, Tcl API, or Java API:

- [“SCADE Projects \(Python\)”](#)
- [“SCADE Projects \(Tcl, Java\)”](#)

# SCADE Projects (Python)



# SCADE Projects (Tcl, Java)



## Part 1

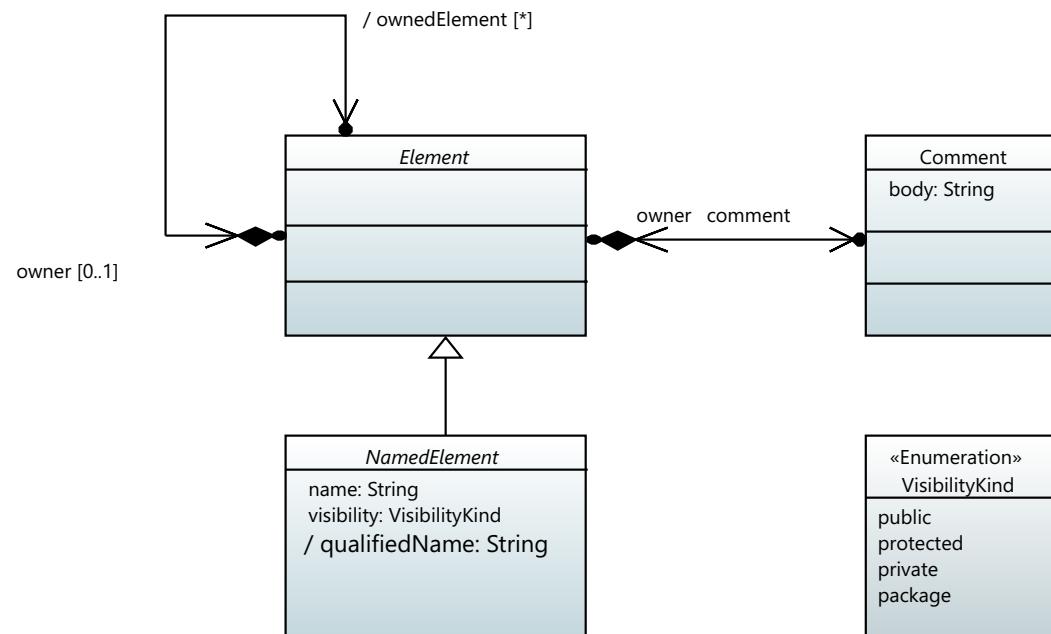
# SCADE Architect Metamodels

- 2/ ["General Construct Metamodels"](#)
- 3/ ["Data Types Metamodel"](#)
- 4/ ["Blocks Structure Metamodel"](#)
- 5/ ["Dataflow-Oriented Communications Metamodels"](#)
- 6/ ["Allocations Metamodel"](#)
- 7/ ["Diagrams, Tables, and Traceability Metamodels"](#)
- 8/ ["Behaviors Metamodels"](#)
- 9/ ["Synchronization Metamodel"](#)
- 10/ ["RichText Metamodel"](#)

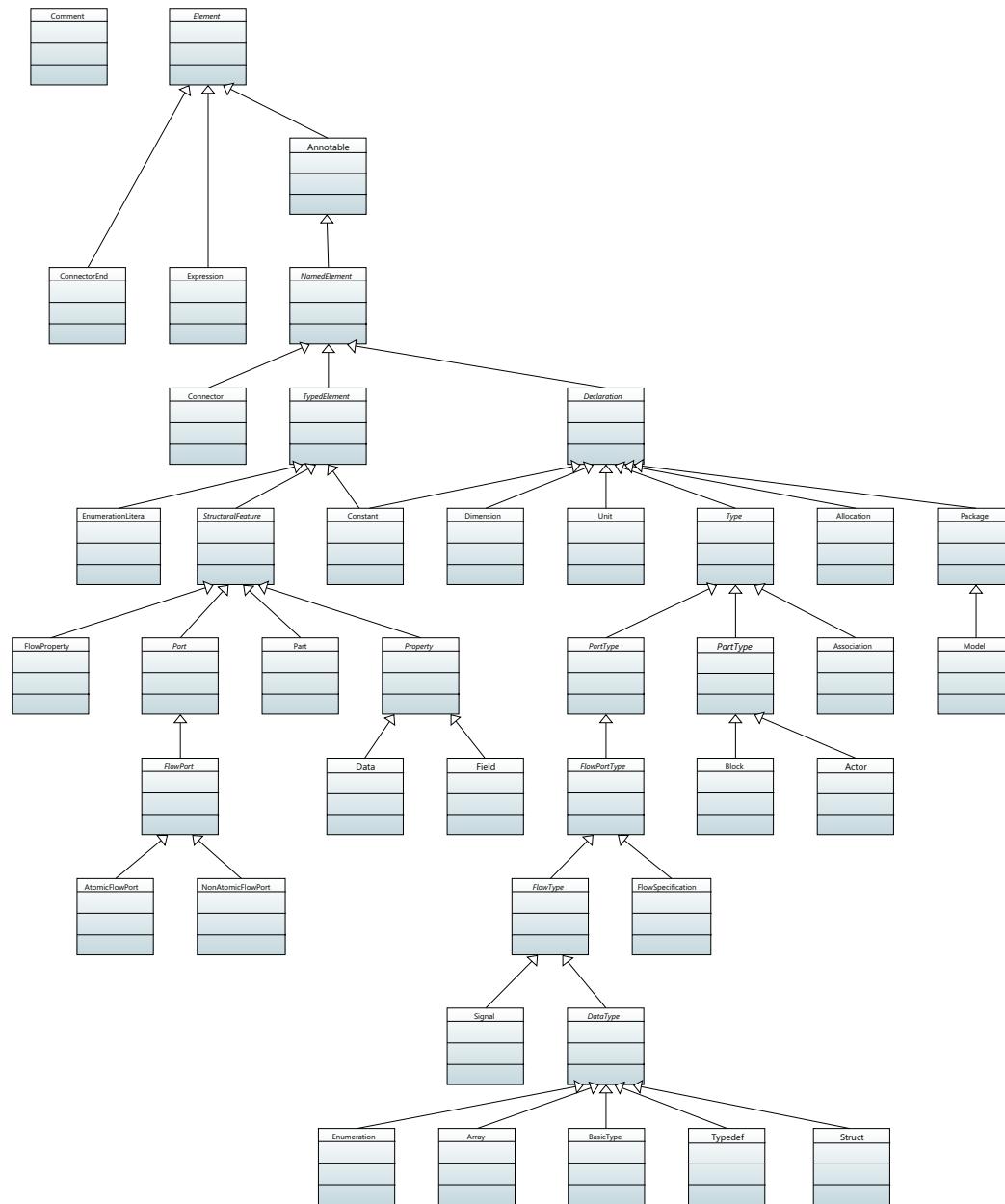
## 2 / General Construct Metamodels

These metamodels present the data structure and inheritance relationships that give access to SCADE Architect language constructs using Python API, Java API, or Tcl API:

- [“Element”](#)
- [“Comment”](#)
- [“NamedElement”](#)
- [“Declaration”](#)
- [“Annotable”](#)
- [“Package”](#)
- [“Model”](#)
- [“Type”](#)
- [“TypedElement”](#)
- [“StructuralFeature”](#)
- [“Property”](#)
- [“Association”](#)
- [“Constant”](#)
- [“Expression”](#)



The following metamodel shows the inheritance tree of all SCADE Architect language constructs:



## Element

<b>Definition</b>	The root abstract concept in the SCADE Architect Language. All other constructs except Comment inherit from Element.
<b>Inheritance</b>	n/a
<b>Attributes</b>	none  <code>comment: Comment</code> Comment applying to the element  <code>owner: Element</code> Parent of the element from a lexical point of view  <code>/ownedElement: Element {derived union}</code>
<b>Associations</b>	Set of elements owned by the element (derived union where subclasses introduce concrete compositions that subset the composition and share the owner role)  <code>table: Tablt</code> The set of tables owned by this element. Only a Package or a Block can contain table elements.  <code>diagram: Diagram</code> The set of diagrams owned by this element. Only a Package or a Block can contain diagram elements.

## Comment

<b>Definition</b>	A textual specification that can be associated to any element.
<b>Inheritance</b>	n/a
<b>Attributes</b>	<code>body: String</code> Text of the comment
<b>Associations</b>	<code>owner: Element</code> Owner of the element (which is also the commented element)

# NamedElement

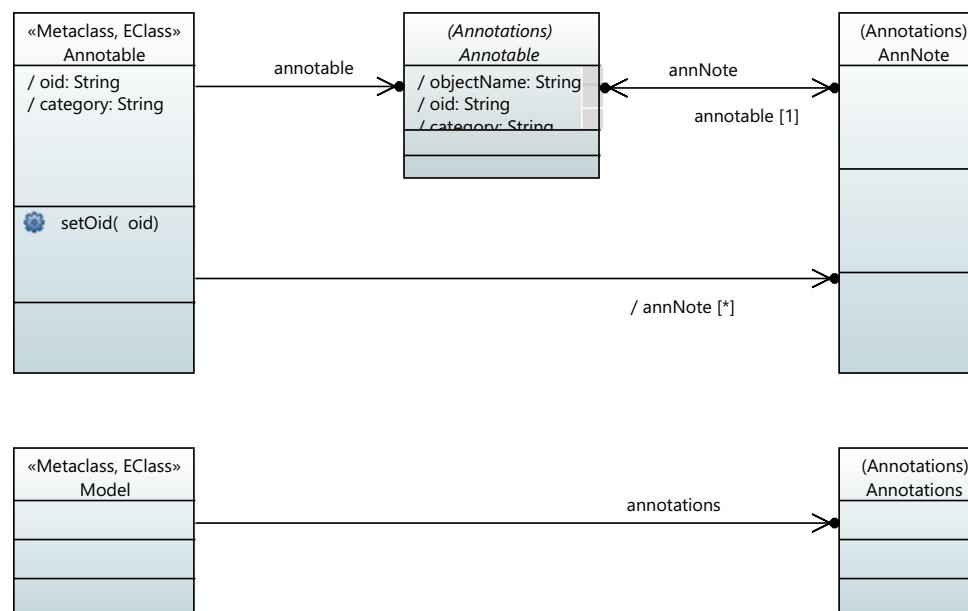
<b>Definition</b>	The abstract concept for all elements having a name.
<b>Inheritance</b>	NamedElement → Element
<b>Attributes</b>	<pre>name: String Name of the element  /qualifiedName: String Qualified name of the element formed by concatenation of all parent namespace of the element up to the root model separated by '::'  visibility: VisibilityKind (Enumeration public private protected package) Visibility of the element with respect to its parent</pre>
<b>Associations</b>	<pre>/incomingAllocation: Allocation [*] Allocations having this named element as target.  /outgoingAllocation: Allocation[*] Allocations having this named element as source.  /allocatedFrom: NamedElement [*] List of named elements that are sources of allocations having the current named element as target.  /allocatedTo: NamedElement[*] List of named elements that are targets of allocations having the current named element as source.</pre>

## Declaration

<b>Definition</b>	It represents any element that can be declared in a Package.
<b>Inheritance</b>	Declaration → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

# Annotable

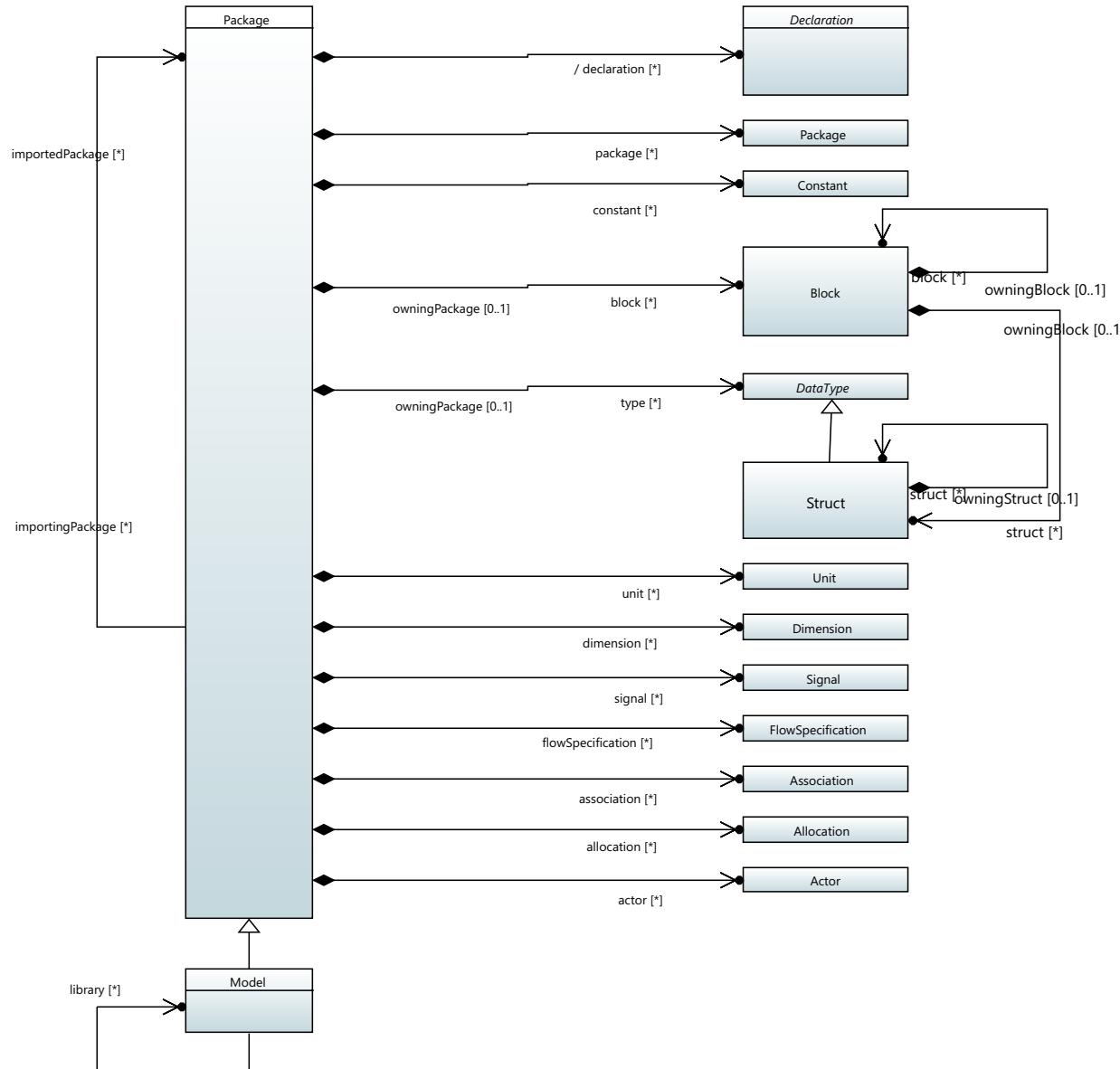
<b>Definition</b>	It represents any element that can carry annotation notes. It links together the SCADE Architect metamodel and the Annotation metamodel which is common to SCADE Suite and SCADE Architect.
<b>Inheritance</b>	Annotable → Element
<b>Attributes</b>	<p>/oid: String The oid of the annotable element</p> <p>/category: String The annotation category which the annotable element belongs to. A note can be created and attached to a given element if its note type matches that element's category, as specified in the annotation schema defining the note type.</p>
<b>Associations</b>	annNote: AnnNote The notes attached to Element



# Package

<b>Definition</b>	It is the main grouping construct at lexical level. A package can contain any number of declarations (including nested packages). A package defines a namespace for its contained declarations.
<b>Inheritance</b>	Package → Declaration
<b>Attributes</b>	n/a
<b>Associations</b>	<pre>declaration: Declaration[*] {derived union} Elements declared in the package (derived union of all more specific kinds of elements owned by the package)  package: Package[*] {subsets Package.declaration} Sub-packages declared within the package (a subset of its declarations)  constant: Constant[*] {subsets Package.declaration} Constants declared within the package (a subset of its declarations)  block: Block[*] {subsets Package.declaration} Reusable block definitions declared within the package (a subset of its declarations)  flowSpecification: FlowSpecification[*] {subsets Package.declaration} FlowSpecifications declared within the package (a subset of its declarations)  signal: Signal[*] {subsets Package.declaration} Signals declared within the package (a subset of its declarations)  type: DataType[*] {subsets Package.declaration} DataTypes declared within the package (a subset of its declarations)  unit: Unit[*] {subsets Package.declaration} Units declared within the package (a subset of its declarations)  dimension: Dimension[*] {subsets Package.declaration} Dimensions declared within the package (a subset of its declarations)  association: Association[*] {subsets Package.declaration} Associations declared within the package (a subset of its declarations)  allocation: Allocation[*] {subsets Package.declaration} Allocations declared within the package (a subset of its declarations)  actor: Actor[*] {subsets Package.declaration} Actors declared within the package (a subset of its declarations)</pre>

The following metamodel shows the data structure for packages and declarations:



## Model

**Definition** It is the root Package of a project or library.

**Inheritance** Model → Package

**Attributes** n/a

**Associations**  
library: Model[\*]  
List of library models used by the model

## Type

**Definition** The abstract concept for all elements that can be used as type for other elements.

**Inheritance** Type → Declaration

**Attributes** n/a

/structuralFeature: StructuralFeature[\*]  
**Associations** List of all structural features of the type (derived union where subclasses have associations that subset the association). Two structural features cannot have the same name

## TypedElement

**Definition** The abstract concept for all elements that can be typed by another element.

**Inheritance** TypedElement → NamedElement

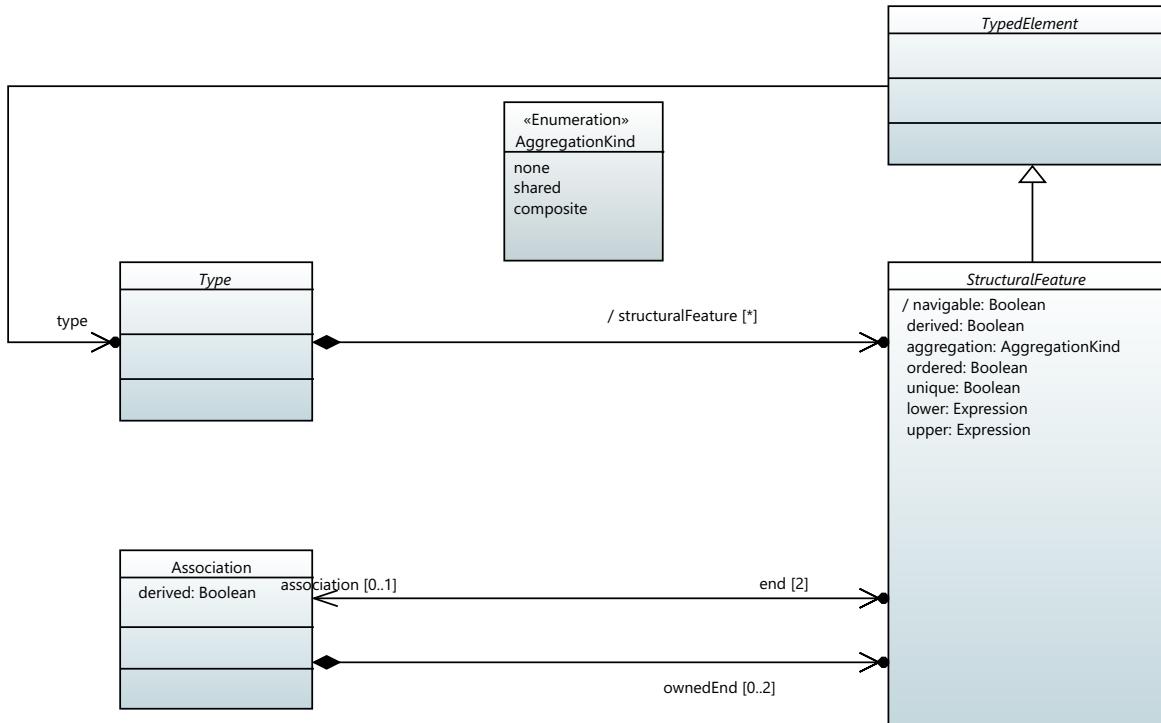
**Attributes** n/a

**Associations** type: Type

## StructuralFeature

<b>Definition</b>	The abstract concept that represents an element that provides sub-structure to another element. A Structural feature has a type (it is a kind of TypedElement) that provides the Type of the sub-elements. It also has a multiplicity, in terms of lower and upper bound, which specifies how many sub-elements this structure feature can provide to its owner at run-time.
<b>Inheritance</b>	StructuralFeature → TypedElement
	<code>lower: Expression</code> Lower bound on the number of sub-elements corresponding to the structural feature at runtime
	<code>upper: Expression</code> Upper bound on the number of sub-elements corresponding to the structural feature at runtime
	<code>ordered: Boolean</code> Whether the set of sub-elements at runtime is ordered (if there is more than one)
<b>Attributes</b>	<code>/navigable: Boolean</code> Whether it is possible to "navigate" through the structural feature at runtime (using the usual "dot expressions" to express instance paths). A derived attribute equal to true if the structural feature is not owned by an Association (which means that the structural feature is owned by a Type and acts as a navigable "field" for objects of this type)
	<code>aggregation: AggregationKind (none, shared or composite)</code> Whether the structural feature is a part of its owner ("composite") or not ("shared" or "none"). The "shared" case means the object(s) denoted by the structural feature belongs to its owner while being shared by another owner (weak ownership); the "shared" case is inherited from UML, but not used in SCADE Architect.
<b>Associations</b>	n/a

The following metamodel shows the data structure for structural features and associations:



## Property

<b>Definition</b>	It is an abstract specialization of <b>StructuralFeature</b> that can hold <b>DataType</b> value(s). Properties can be used to represent data of a <b>Block</b> , or fields of a <b>DataType</b> or <b>Signal</b> .
<b>Inheritance</b>	<b>Property</b> → <b>StructuralFeature</b>
<b>Attributes</b>	n/a
<b>Associations</b>	<p><b>type:</b> <code>DataType { redefines TypedElement.type }</code></p> <p>Type of the property (inherited from <b>TypedElement</b>) redefined to be restrained as mere <b>DataType</b></p>

# Association

It is a relationship between two types. An association has two ends, indicating which role the types play with respect to each other in the context of the relationship. Each end is a structural feature (with type and multiplicity).

There are two kinds of associations:

- Simple association
- Composition

A composition means that the relationship is a “whole-to-part” association: one of the ends of the association belongs to the other end.

In SCADE Architect, an association is navigable only from its source to its target (meaning that, in the case of blocks, the target plays the role of a part (if composition) or reference (if association) with respect to the source).

## Definition

## Inheritance

Association → NamedElement

## Attributes

n/a

`end: StructuralFeature[2] {ordered}`

Both ends of the association. A number of constraints limits what kind of types can be connected by an association:

- Neither the source nor the target can be a FlowSpecification
- The target cannot be a Signal
- If the target is a Block, so must be the source (the target is either a part or a reference with respect to the source, depending on aggregation)
- If the target is a DataType, then it must be composite (the target is a data if the source is a Block, or a field if the source is a FlowType)

`/source: StructuralFeature`

Source of the association, i.e. the first end

`/target: StructuralFeature`

Target of the association, i.e. the second end

# Constant

<b>Definition</b>	It represents a named expression that evaluates to a constant value. Constants in a model are typically used in expressions for the lower or upper bound multiplicities of StructuralFeatures, or for the size of Arrays.
<b>Inheritance</b>	Constant → Declaration, TypedElement
<b>Attributes</b>	<code>value: Expression</code> Value of the constant. Note that the type of the value expression must be compatible with the static type of the constant itself (which, in absence of type inheritance/aliasing, simply means that both types shall be the same).
<b>Associations</b>	n/a

# Expression

**Definition** It represents a way to compute a value of a certain type.

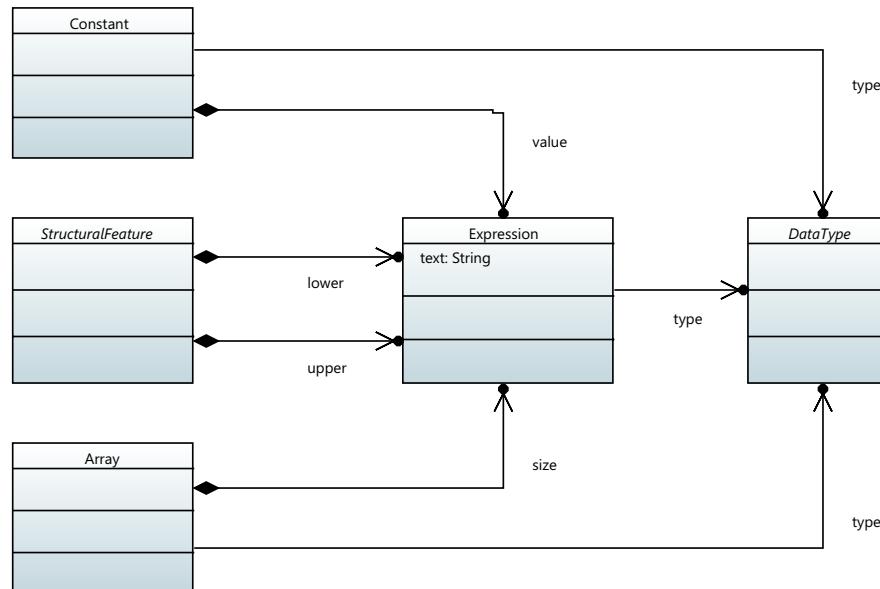
**Inheritance** Expression → Element

**Attributes**  
body: String  
Non-interpreted text for the expression

**Associations**  
type: DataType[0..1]  
Type of the expression when it can be computed from the expression's body

## Note

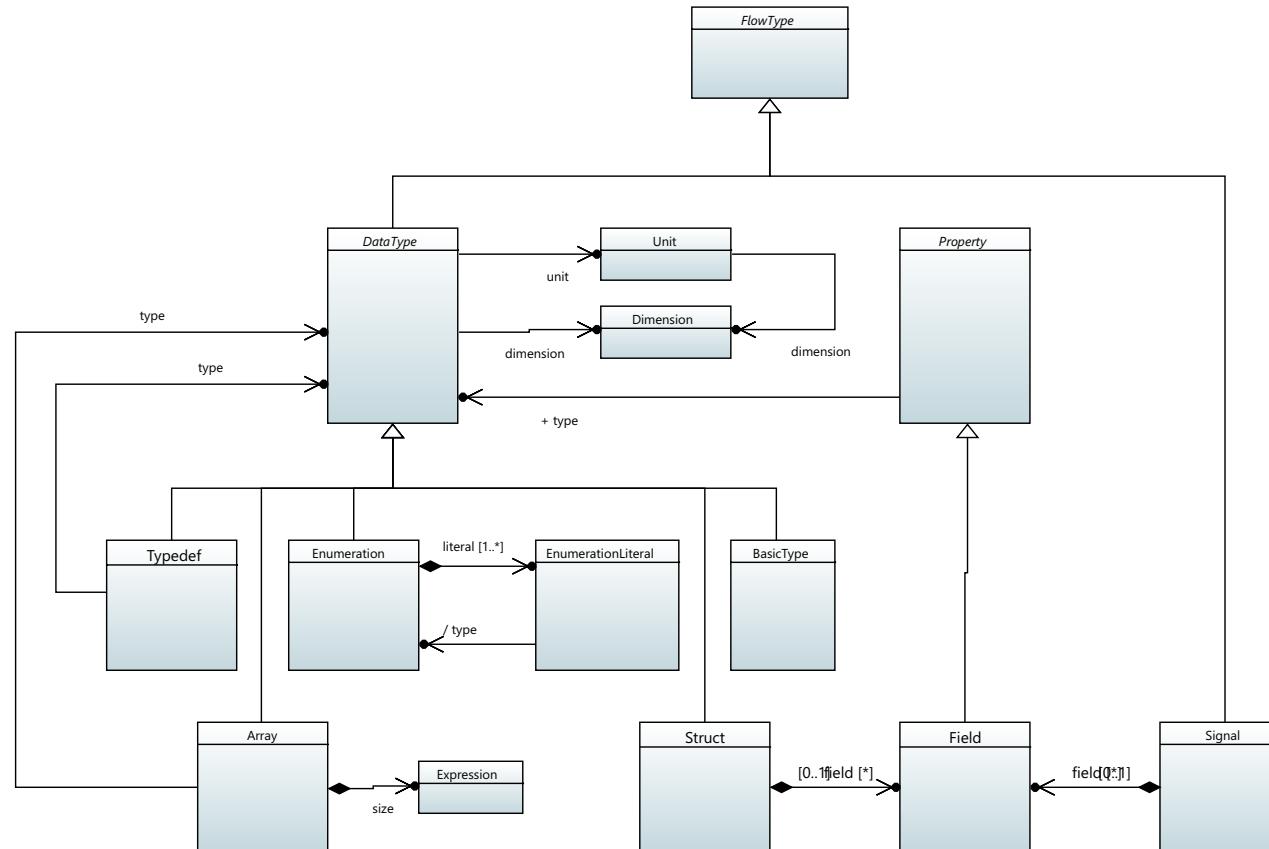
SCADE Architect only supports "opaque" expressions given in the form of non-interpreted text.



# 3 / Data Types Metamodel

This metamodel presents the data structures that give access to SCADE Architect data types using Python API, Java API, or Tcl API:

- [“DataType”](#)
- [“Enumeration”](#)
- [“EnumerationLiteral”](#)
- [“Array”](#)
- [“Signal”](#)
- [“Unit”](#)
- [“Dimension”](#)
- [“Typedef”](#)
- [“BasicType”](#)
- [“Struct”](#)
- [“Field”](#)



# DataType

<b>Definition</b>	It is the (abstract) concept representing a data-type in the usual meaning in language design. DataTypes can carry Unit and Dimension as defined in the SI system. Unit and Dimension have an impact on typing. DataTypes are compatible if they have the same dimension (but assignment may require scaling if units are different).
<b>Inheritance</b>	DataType → FlowType
<b>Attributes</b>	n/a  unit: Unit[0..1] Optional unit of a DataType
<b>Associations</b>	dimension: Dimension[0..1] Optional dimension of a DataType  Unit and dimension, when specified, shall be consistent: The dimension of the unit shall be the same as the dimension of the type.

# Enumeration

<b>Definition</b>	It represents a data-type whose set of possible values is discrete and finite. Those values are enumerated as EnumerationLiterals.
<b>Inheritance</b>	Enumeration → DataType
<b>Attributes</b>	n/a
<b>Associations</b>	literals: EnumerationLiteral[*] {ordered} Enumeration literals for the Enumeration type

## EnumerationLiteral

<b>Definition</b>	One of the discrete values of an Enumeration type.
<b>Inheritance</b>	EnumerationLiteral → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	<pre>/type: Enumeration {redefines TypedElement.type}</pre> <p>Type of the EnumerationLiteral redefined to be restrained to Enumeration (also derived to always point to the owning Enumeration, <i>i.e.</i>, /type=owner)</p>

## Array

<b>Definition</b>	<p>It represents an ordered and indexed set of values. An array has a type (the types of its values) and a fixed size (the static number of values).</p>
<b>Inheritance</b>	Array → DataType
<b>Attributes</b>	n/a
<b>Associations</b>	<pre>type: DataType</pre> <p>Type of elements present in the array</p> <pre>size: Expression</pre> <p>Size of the Array (size expression must be statically computable)</p>

## Signal

<b>Definition</b>	<p>It represents a sporadic event that can occur in the system. A Signal is a Type (occurrences being the instances of the Type). A Signal can be "pure" (without Fields) or valued (with Fields).</p>
<b>Inheritance</b>	Signal → FlowType
<b>Attributes</b>	n/a
<b>Associations</b>	<pre>fields: Field[*] {ordered}</pre> <p>The fields of the signal.</p>

## Unit

<b>Definition</b>	It represents the Unit concept of the SI system. Examples of Units are "kilometers" or "miles" (which dimension is "distance"), or "kilograms" or "pounds" (which dimension is "weight").
<b>Inheritance</b>	Unit → Declaration
<b>Attributes</b>	n/a
<b>Associations</b>	dimension: Dimension Dimension of the Unit

## Dimension

<b>Definition</b>	It represents the Dimension concept of the SI system. Examples of Dimension are "distance" or "weight".
<b>Inheritance</b>	Dimension → Declaration
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Typedef

<b>Definition</b>	It represents a type alias as can be found in C-based programming languages. A typedef can also be used to specify types whose representation is based on predefined types such as "real" or "int", but that have context-specific units and/or dimensions in addition.
<b>Inheritance</b>	Typedef → DataType
<b>Attributes</b>	n/a
<b>Associations</b>	type: DataType Data type of which this data type is an alias

## BasicType

<b>Definition</b>	It represents a type with no further structural decomposition, such as "int" or "bool".
<b>Inheritance</b>	BasicType → DataType
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

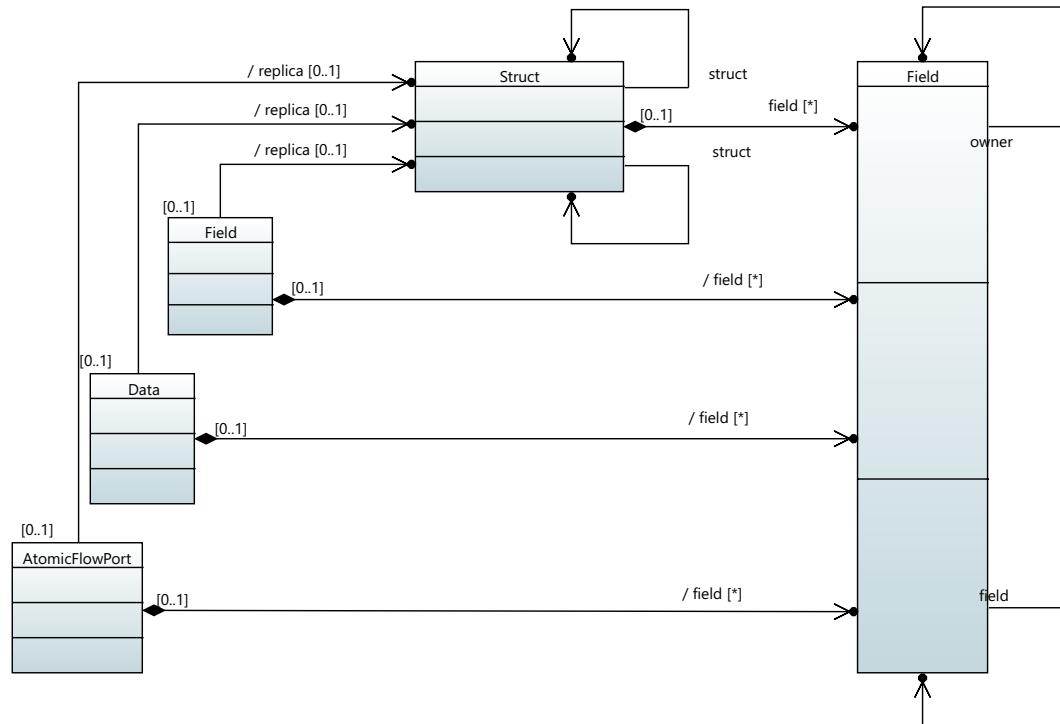
## Struct

<b>Definition</b>	<p>It represents a data type that can be further structured with fields.</p> <p>When used to type atomic flow ports, data or fields of other structs, the struct is replicated (using the same mechanism as for block replicas), which means it becomes possible to denote a field within a specific port, data or field (that is, a field within the struct replica that types the owning port, data or field).</p>
<b>Inheritance</b>	Struct → DataType
<b>Attributes</b>	n/a
<b>Associations</b>	<p>n/a fields: Field[*] {ordered}</p> <p>The fields of the data type.</p> <p>prototype: Struct[1]</p> <p>The prototype for a struct replica.</p> <p>declaration: Struct[0..1]</p> <p>The block declaration for a struct replica.</p>

### Note

Struct is a special kind of SysML Block so as to benefit from the replication mechanism.

The following metamodel shows the data structure for Struct replication within ports/data/fields:



## Field

**Definition** It represents a sub-value within of a Struct or Signal type.

**Inheritance** Field → Property

**Attributes** n/a

`replica: Struct[0..1]`

The field-specific struct replica if this field is typed by a struct.

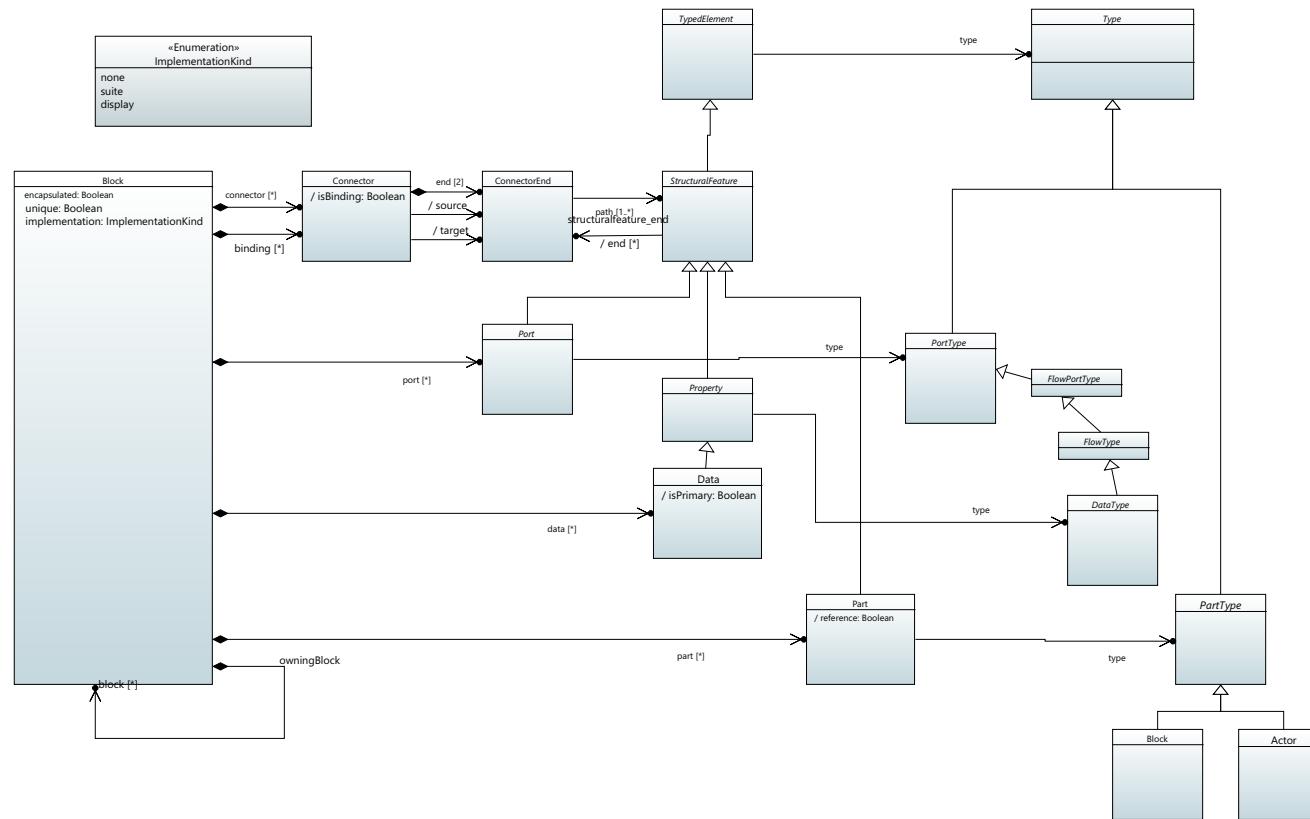
`fields: Field[*] {ordered}`

The fields of this field, if the field is typed by a struct (that is, the fields of its replica).

# 4 / Blocks Structure Metamodel

This metamodel presents the data structures that give access to SCADE Architect structural modeling of blocks using Python API, Java API, or Tcl API:

- ["Actor"](#)
- ["Block"](#)
- ["Behavior"](#)
- ["Constraint Blocks"](#)
- ["Part & Reference"](#)
- ["PartType"](#)
- ["Data"](#)
- ["Port"](#)
- ["PortType"](#)
- ["Connector"](#)
- ["ConnectorEnd"](#)



## Actor

<b>Definition</b>	An actor represents an external stakeholder - usually a user - of the system being modeled. They can be represented in BDDs and IBDs to show how the user(s) interacts with the system. Actors are usually part of a top-level block representing the system and its environment (a.k.a. "universe"). An IBD for such a top-level block is called a "Context Diagram".
<b>Inheritance</b>	Actor → PartType
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

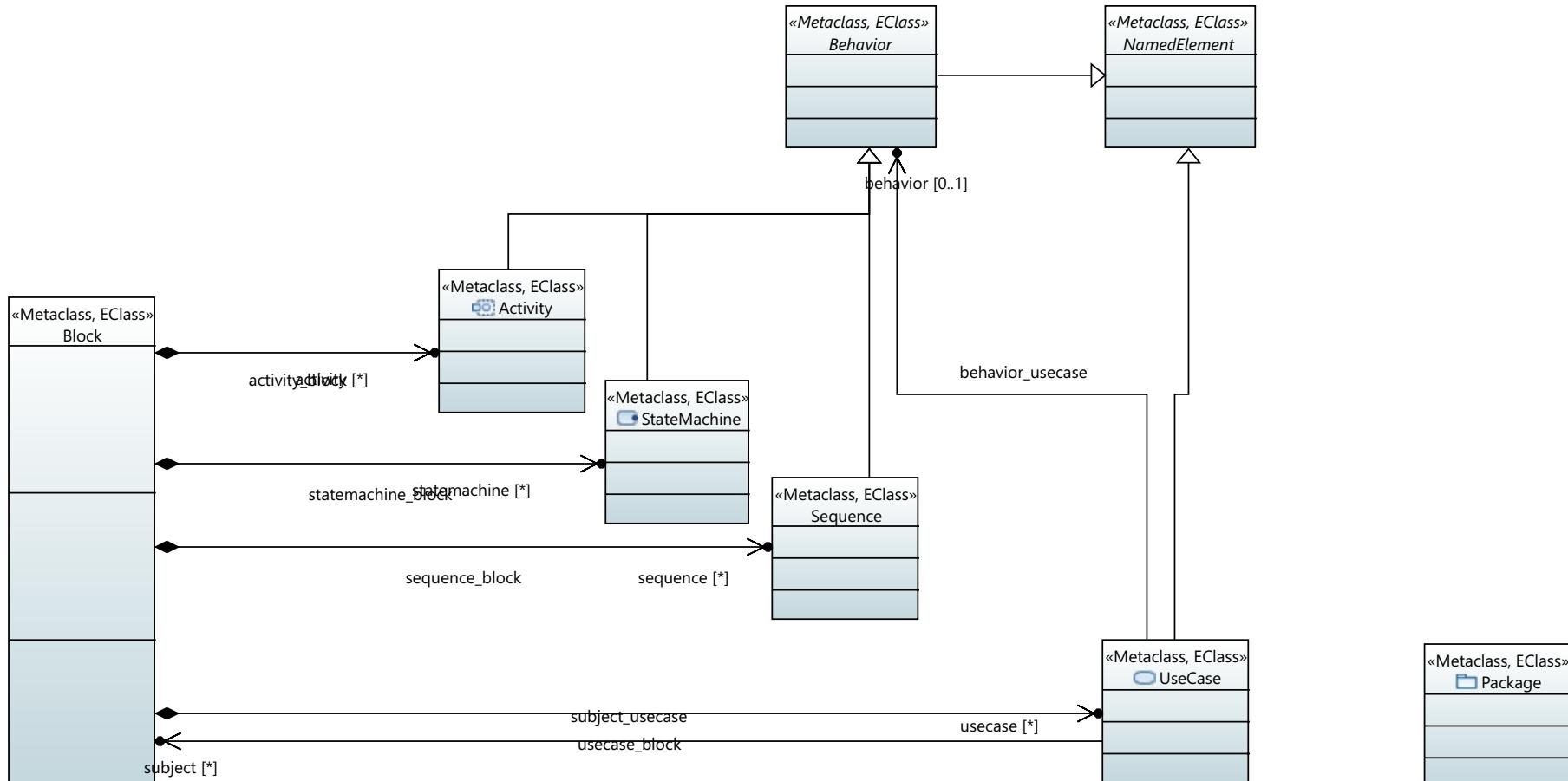
# Block

<b>Definition</b>	It is the primary structuring construct in the language. Blocks are the modular units of system description. Blocks can be used to model either the logical/functional or physical decomposition of a system. Blocks are types: there can be several replicas of a Block definition (also called instances). However, a special kind of block, called unique block, can be used only once, in the context of the parent block in which its defined.
<b>Inheritance</b>	$\text{Block} \rightarrow \text{PartType}$
<b>Attributes</b>	<p><code>encapsulated: Boolean</code> Whether there can be inbound connections to inner parts/references of the block or whether the block is “opaque” with respect to the outside</p> <p><code>unique: Boolean</code> Whether the block can be replicated several times for several parts, or whether the block is a unique block (meant to type only a single part).</p>
<b>Associations</b>	<p><code>port: Port[*] {ordered, subsets structuralFeature}</code> Ports of the block</p> <p><code>part: Part[*] {subsets structuralFeature}</code> Parts and references making up the block, that is, replicas of other Blocks (including “single” replicas of unique blocks) or Actors owned or referenced by this Block.</p> <p><code>data: Data[*] {ordered, subsets structuralFeature}</code> Data of the block</p> <p><code>connector: Connector[*]</code> Connectors of the block. Binding connectors are excluded.</p> <p><code>binding: Connector[*]</code> Data-propagation binding connectors of the block</p> <p><code>block: Block[*]</code> Unique blocks defined and used as parts of this block.</p> <p><code>prototype: Block[1]</code> Prototype for a block replica</p> <p><code>declaration: Block[0..1]</code> Block declaration for a block replica</p>

The concepts of block prototypes and part replica support the possibility to distinguish an element (part, port, connector, data) within a replica of a block (*i.e.*, within a part typed by the block) from the “same” element within a different replica of the same block (*i.e.*, within another part typed by the same block).

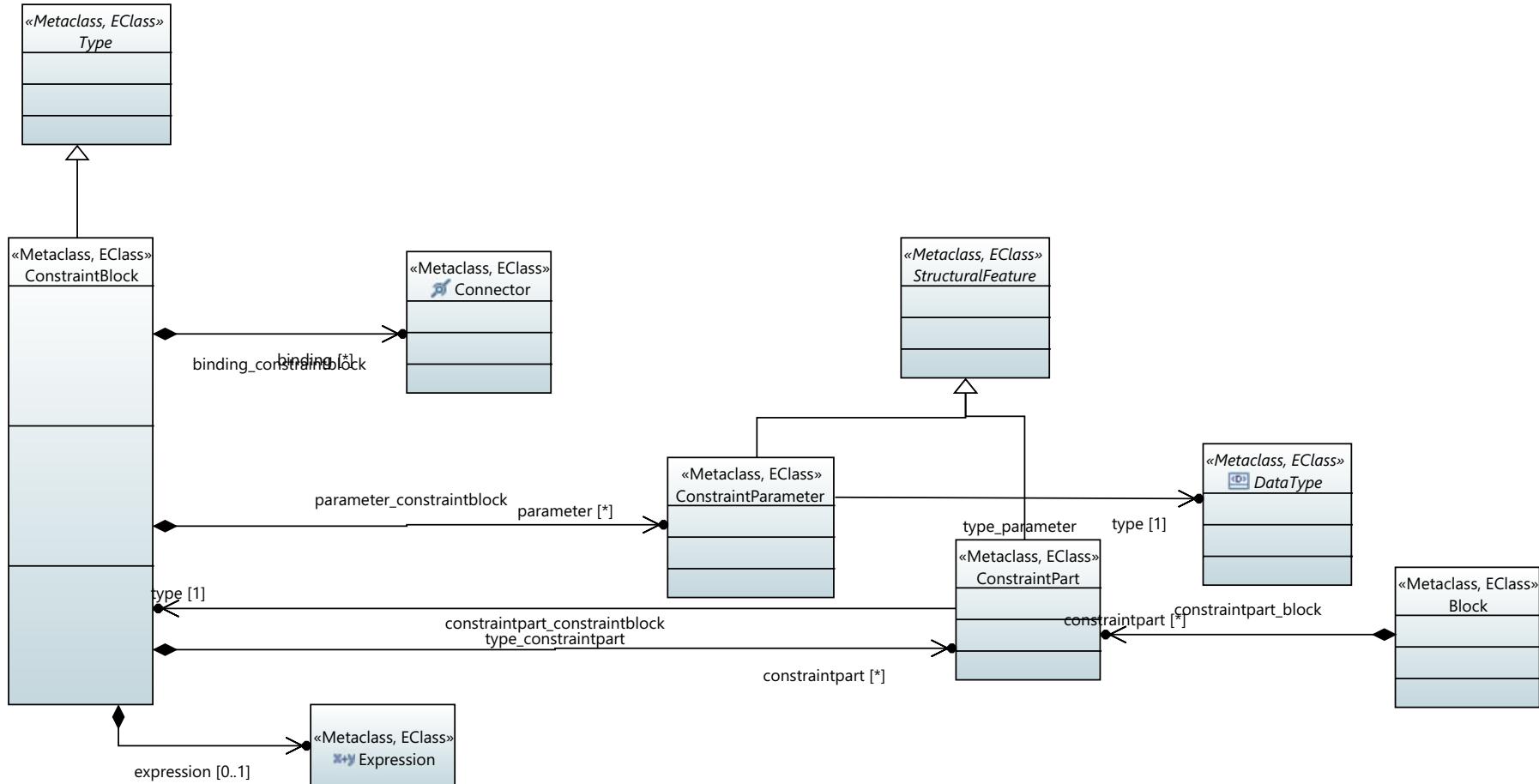
# Behavior

The following metamodel shows the data structure for behavior definitions:



# Constraint Blocks

The following metamodels shows the data structure for constraint blocks:

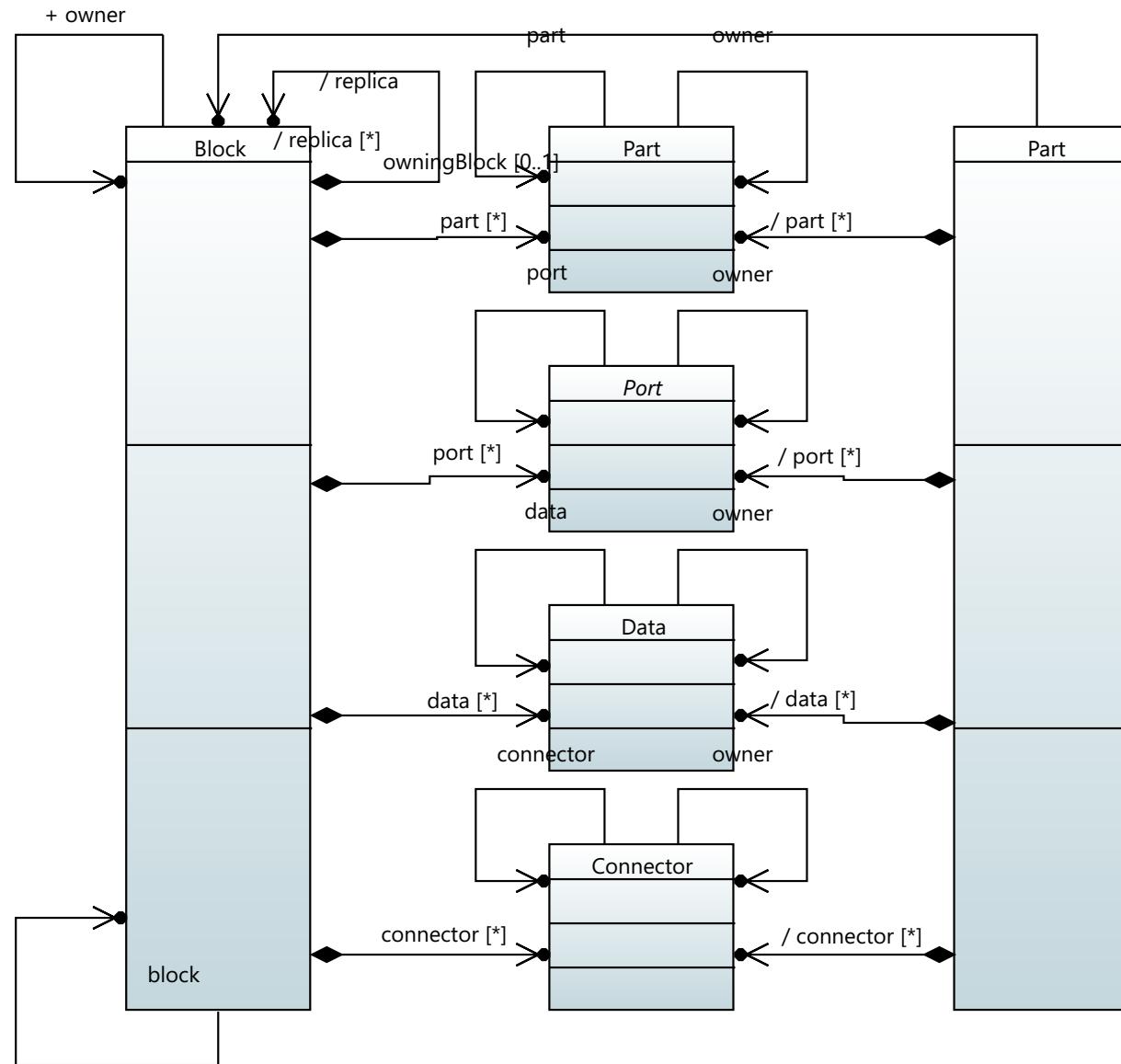


## Part & Reference

Definition	<p>A <i>part of a block</i> is made of nested replica(s) of other blocks reused as building block for the block. A part with multiplicity greater than one represents a set of identical replicas (by opposition to a single replica) that all play the same role with respect to the owner block (they are addressed as an indistinguishable set). Yet a block can have several parts that are replicas of the same block, if those parts play different roles. A <i>part by reference within a block</i> represents a pseudo-part, that is, a mere “pointer” typed by a given block definition, with the intention of pointing to a set of replicas of the corresponding block definition that are located “somewhere else” in the block hierarchy. References are a means of “projecting” within the current block context other replicas that do not actually belong to the current block, to manipulate them as if they were parts of the block. References can be seen as shortcuts throughout the instance tree. A reference cannot specify what replica it refers to.</p>
Inheritance	Part → StructuralFeature  <pre>/reference: Boolean</pre>
Attributes	<p>Whether the part is actually only a reference (not a full-fledged part) derived from <code>StructuralFeature.aggregation</code> (<code>reference == true</code> if <code>aggregation != composite</code>)</p> <pre>type: PartType {redefines TypedElement.type}</pre> <p>Type of the Part (inherited from <code>TypedElement</code>) redefined to be restrained to Block or Actor</p> <pre>association: Association</pre> <p>Association of which this part is an end</p> <pre>/replica: Block[0..1]</pre> <p>Block replica corresponding to this part if the type of the part is a block (there is no replica for actor-part as actors have no sub-elements to be distinguished). Multiplicity is not taken into account, so there is at most one replica per part.</p> <pre>port: Port[*] {ordered, subsets structuralFeature}</pre> <p>Ports of the part, derived from the ports of the part's associated replica</p> <pre>part: Part[*] {subsets structuralFeature}</pre> <p>Parts and references of the part, that is, replicas of other Blocks or Actors owned or referenced by this part, derived from the parts of the part's associated replica</p> <pre>data: Property[*] {ordered, subsets structuralFeature}</pre> <p>Data of the part, derived from the data of the part's associated replica</p> <pre>connector: Connector[*]</pre> <p>Connectors of the part, derived from the connectors of the part's associated replica</p>

Any reference should be initialized with an expression representing a path to a (set of) replica(s) within the instance tree relative to the block context.

The following metamodel shows the data structure for block replicas:



## PartType

<b>Definition</b>	An abstract concept that encompasses both Actor and Block. It is used to type parts (or references).
<b>Inheritance</b>	PartType → Type
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Data

<b>Definition</b>	It represents some information carried by a block.
<b>Inheritance</b>	Data → Property
<b>Attributes</b>	<code>/isPrimary: Boolean</code> Whether the data is a primary data or a data proxy resulting from data propagation.
<b>Associations</b>	<code>replica: Struct[0..1]</code> Data-specific struct replica if this data is typed by a struct <code>fields: Field[*] {ordered}</code> Fields of this data, if the data is typed by a struct (that is, the fields of its replica)

# Port

<b>Definition</b>	The port of a block represents a gate between the block and its environment. All communications between blocks go through ports. The set of ports are the “interface” of the block. The port concept is abstract. Depending on the kind of intended communication, there are different kinds of concrete ports for dataflow-oriented communication: <ul style="list-style-type: none"><li>• Atomic flow ports: simple input/output flow ports</li><li>• Non-atomic flow ports: grouped flow ports</li></ul>
<b>Inheritance</b>	Port → StructuralFeature
<b>Attributes</b>	n/a
<b>Associations</b>	<code>type: PortType {redefines TypedElement.type}</code> Type of the port (inherited from TypedElement) redefined to be restrained to PortType

# PortType

<b>Definition</b>	Any type that can be used to type a Port. It is used to type ports (flow ports and standard ports, respectively).
<b>Inheritance</b>	PortType → Type
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

# Connector

It represents a communication link. There are two kinds of connectors:

- A *delegation connector* connects a port of a block to one of its part (directly or through a port of the part). Information is delegated by the block to the corresponding part.
- An *assembly connector* connects two parts (directly or through ports of the parts). Information is exchanged between both parts.

Direct connection to a part is less precise than connection through a port of the part. The second kind of connector should be used when the model is precise enough. Connectors are owned by the block that owns the connected part(s).

## Definition

## Inheritance

Connector → NamedElement

## Attributes

/isBinding: Boolean

Whether the connector is a data-propagation binding connector or a plain connector

## Associations

end: ConnectorEnd[2] {ordered}

Two connected ends

# ConnectorEnd

## Definition

It represents one side of the connector and carries all required information to determine the target of the connection precisely. As seen above, a connector may connect to a part directly or through a port of that part. This means that a connector-end might involve not just one model element, but two: the part and a port of that part (more precisely, a port of the block that types the part).

## Inheritance

ConnectorEnd → Element

## Attributes

n/a

path: StructuralFeature[1..\*] {ordered}

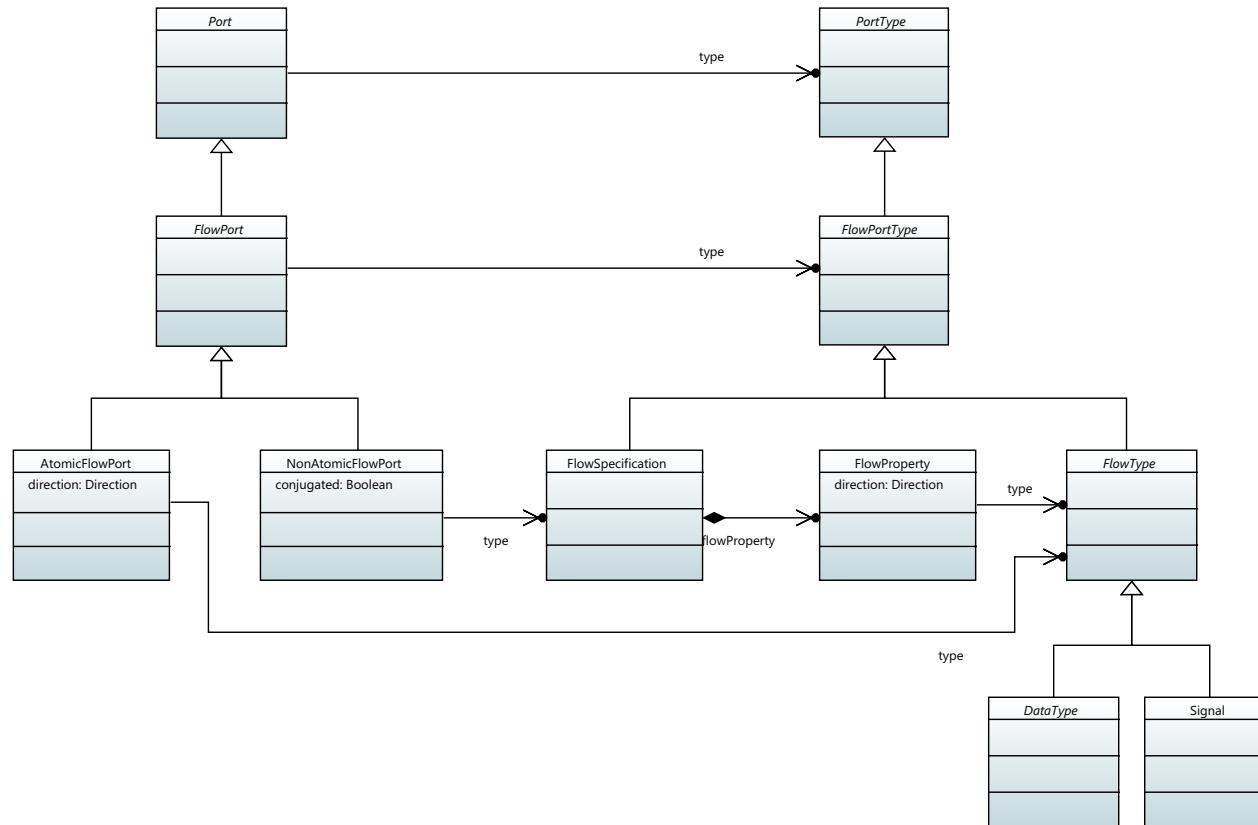
Path to the connected element:

- If connector-end connects to a port directly, the path is made of the port alone
- If connector-end connects to a part directly, the path is made of the part alone
- If connector-end connects to a port of a part, the path is made of the part and port in that order
- If owning connector passes through non-encapsulated parts before reaching its target, the connector-end is “nested”, and crossed parts are added at the beginning of the path

# 5 / Dataflow-Oriented Communications Metamodels

This metamodel presents the data structures that give access to SCADE Architect dataflow-oriented communications using Python API, Java API, or Tcl API:

- [“AtomicFlowPort”](#)
- [“FlowPort”](#)
- [“NonAtomicFlowPort”](#)
- [“FlowType”](#)
- [“FlowSpecification”](#)
- [“FlowProperty”](#)



## AtomicFlowPort

<b>Definition</b>	A FlowPort that represents a single dataflow (input, output, or bidirectional flow).
<b>Inheritance</b>	AtomicFlowPort → FlowPort
<b>Attributes</b>	<code>direction: DirectionKind {enumeration in, out, inout} (=SysML:::FlowPort.direction)</code> Direction of the dataflow
<b>Associations</b>	<code>type: FlowType {redefines FlowPort.type}</code> Type of the AtomicFlowPort (inherited from FlowPort) covariantly redefined to be restrained to FlowType ( <i>i.e.</i> , DataType or Signal)

## FlowPort

<b>Definition</b>	An abstract concept for dataflow-oriented communication ports. It can take the following forms: <ul style="list-style-type: none"><li>• <i>Atomic flow ports</i> representing a single dataflow</li><li>• <i>Non-atomic flow ports</i> representing a group of dataflows (group type is given through a so-called FlowSpecification that itself contains the individual dataflows)</li></ul>
<b>Inheritance</b>	FlowPort → Port
<b>Attributes</b>	<code>/atomic: Boolean</code> Whether the port is atomic (representing a single dataflow) or not
<b>Associations</b>	<code>type: FlowPortType {redefines Port.type}</code> Type of the FlowPort (inherited from Port) covariantly redefined to be restrained to FlowPortType ( <i>i.e.</i> , FlowSpecification, DataType, or Signal) <code>replica: Struct[0..1]</code> Port-specific struct replica if this port is typed by a struct <code>fields: Field[*] {ordered}</code> Fields of this port, if the port is typed by a struct (that is, the fields of its replica)

## NonAtomicFlowPort

<b>Definition</b>	It represents a group of dataflows (the group description is given through a so-called FlowSpecification that itself contains the individual dataflows).
<b>Inheritance</b>	NonAtomicFlowPort → FlowPort
<b>Attributes</b>	<code>conjugated: Boolean</code> : Indicates that the direction of all individual dataflows specified by the FlowSpecification must be reversed
<b>Associations</b>	<code>type: FlowSpecification {redefines FlowPort.type}</code> Type of the NonAtomicFlowPort (inherited from Port) covariantly redefined to be restrained to FlowSpecification

## FlowType

<b>Definition</b>	An abstract concept that encompasses both DataType and Signal. It is used to type atomic flow ports and flow properties (both accept either DataTypes or Signals as type).
<b>Inheritance</b>	FlowType → FlowPortType
<b>Attributes</b>	n/a
<b>Associations</b>	<code>field: Field[0..*] {ordered, subsets Type.structuralFeature}</code> Optional fields of the DataType or (valued) Signal

## FlowSpecification

<b>Definition</b>	It describes a group of dataflows (each called a flow property).
<b>Inheritance</b>	FlowSpecification → FlowPortType
<b>Attributes</b>	n/a
<b>Associations</b>	<code>flowProperty: FlowProperty[*] {ordered, subsets Type.structuralFeature}</code> FlowProperties that compose the FlowSpecification

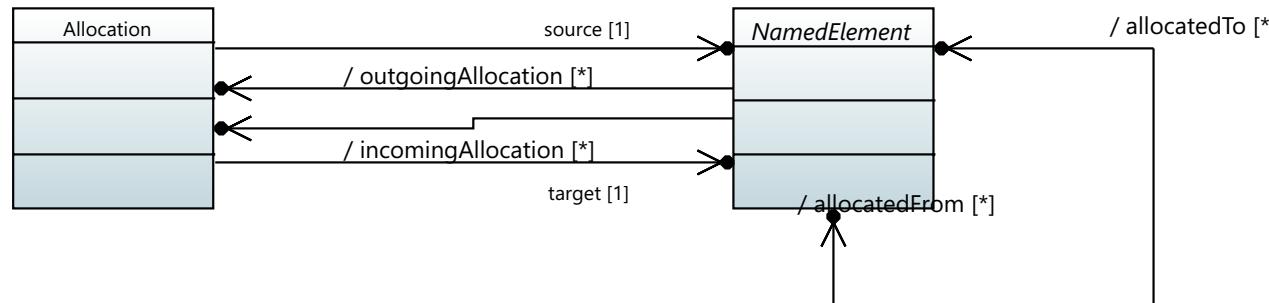
# FlowProperty

<b>Definition</b>	It represents an individual dataflow within a FlowSpecification. It is similar to an AtomicFlowPort, but owned by a FlowSpecification instead of a Block. From an end-user perspective, it is possible to assimilate atomic flow ports and flow properties. They are just inputs or outputs whether directly at block level or within a non-atomic port specification.
<b>Inheritance</b>	FlowProperty → StructuralFeature
<b>Attributes</b>	<code>direction: DirectionKind (enumeration in, out, inout)</code> Direction of the dataflow
<b>Associations</b>	<code>type: FlowType {redefines TypedElement.type}</code> Type of the flow property redefined covariantly to be restrained to FlowType

# 6 / Allocations Metamodel

This metamodel presents the data structures that give access to SCADE Architect allocations using Python API, Java API, or Tcl API:

- [“Allocation”](#)



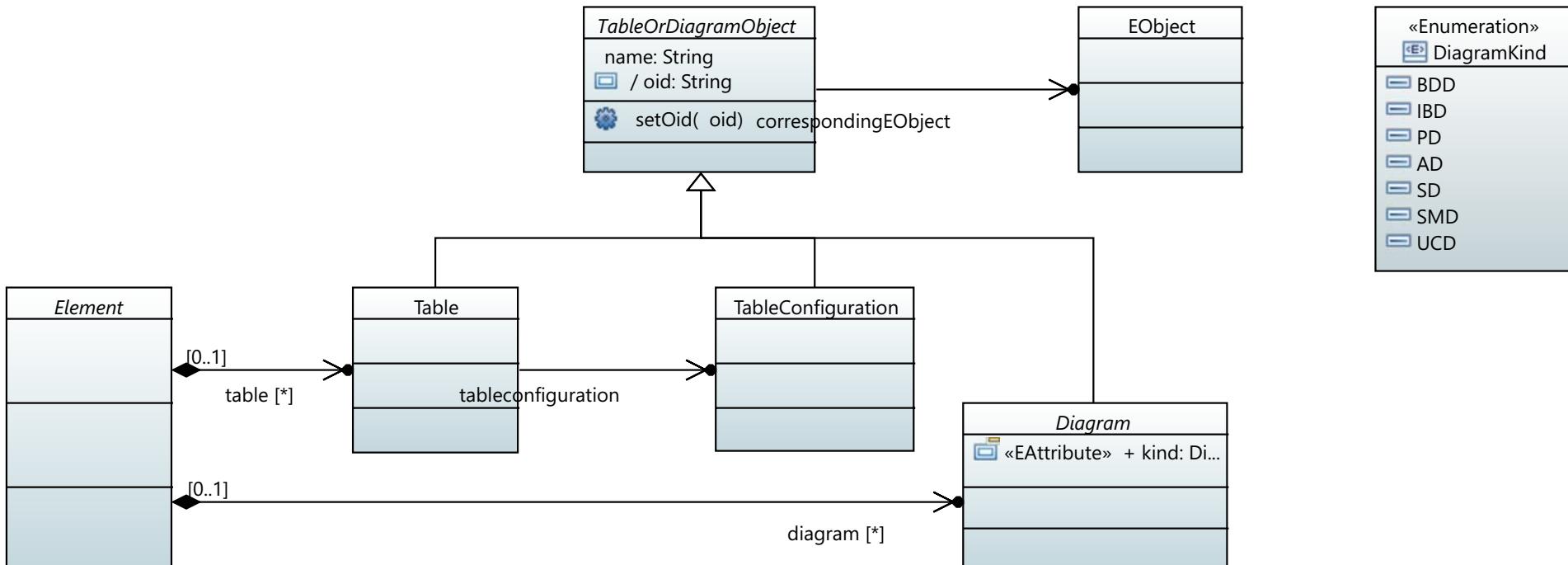
## Allocation

<b>Definition</b>	It is a directed relationship between two arbitrary elements that specifies that one element must be allocated onto another element. Usually, the source element represents a logical element (typically a logical function in system engineering) while the target element represents a physical element (typically an “organ” or a piece of hardware resource that supports the realization of the function).
<b>Inheritance</b>	Allocation → Declaration
<b>Attributes</b>	n/a
<b>Associations</b>	<p>source: NamedElement Source element of the allocation (the element being allocated)</p> <p>target: NamedElement Target element of the allocation (the element onto which something is allocated)</p>

# 7 / Diagrams, Tables, and Traceability Metamodels

These metamodels present the data structures that give access to SCADE Architect diagrams, tables, and traceability information using Python API, Java API, or Tcl API:

- [“TableorDiagramObject”](#)
- [“Table”](#)
- [“TableConfiguration”](#)
- [“Diagram”](#)
- [“All Diagrams”](#)
- [“Requirements Traceability”](#)



## TableorDiagramObject

<b>Definition</b>	TableOrDiagramObject is the root (abstract) concept in the SCADE Architect language for diagram or table elements. All the diagram and table constructs inherit from TableOrDiagramObject.
<b>Inheritance</b>	n/a
<b>Attributes</b>	<code>name: String</code> The name of the element.
<b>Associations</b>	<code>correspondingEObject: EObject</code> The corresponding EObject from the notation model (for diagram concepts) or nattable model (for table concepts).

## Table

<b>Definition</b>	Table is the construct in SCADE Architect language that corresponds to the representation as a hierarchical tabular view of the elements starting from a context Element (table owner that can be a Package or a Block).
<b>Inheritance</b>	Table → TableOrDiagramObject
<b>Attributes</b>	n/a
<b>Associations</b>	<code>tableconfiguration: TableConfiguration</code> The tableconfiguration of a table defines the table configuration (or style) applied to the table.

## TableConfiguration

<b>Definition</b>	TableConfiguration is the construct in SCADE Architect language that corresponds to a table configuration (or style) defining the configuration of a hierarchical tabular view (rows, columns, ...).
<b>Inheritance</b>	Table → TableOrDiagramObject
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

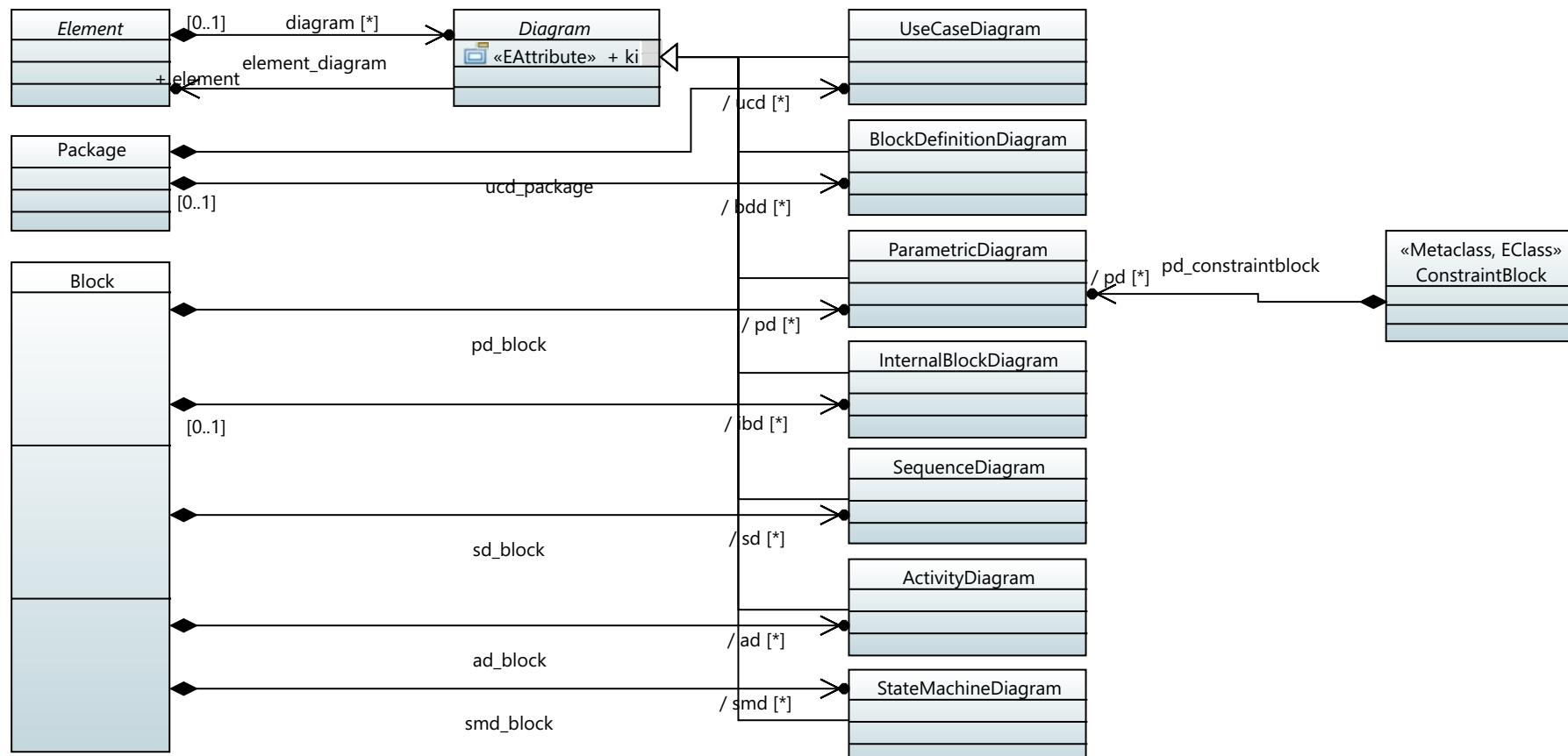
## Diagram

<b>Definition</b>	Diagram is the abstract construct in SCADE Architect language that corresponds to the representation as a diagram view of the elements starting from a source Element (diagram owner that can be a Package or a Block).
<b>Inheritance</b>	Diagram → TableOrDiagramObject
<b>Attributes</b>	<code>kind: DiagramKind (enumeration BDD, IBD, PD, AD, SD, SMD, UCD)</code> The kind of the diagram: Block Definition, Internal Block, Parametric, Activity, Sequence, State Machine, or Use Case.
<b>Associations</b>	n/a

# All Diagrams

The Diagram concept can have various implementations as described below.

- ["Block Definition Diagram \(BDD\)"](#)
- ["Internal Block Diagram \(IBD\)"](#)
- ["Parametric Diagram \(PD\)"](#)
- ["Activity Diagram \(AD\)"](#)
- ["Sequence Diagram \(SD\)"](#)
- ["State Machine Diagram \(SMD\)"](#)
- ["Use Case Diagram \(UCD\)"](#)



## Block Definition Diagram (BDD)

<b>Definition</b>	BlockDefinitionDiagram is the concept representing the SysML Block Definition Diagram supported by SCADE Architect. When this class is instantiated from the API, the 'kind' enum takes the value 'BDD'.
<b>Inheritance</b>	BlockDefinitionDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows a declaration-oriented view of the model. Typically, it presents the Blocks and the associations between Blocks. It can also present the DataTypes and the FlowSpecification used to type attributes or flow ports of the Blocks, and various Constants if necessary.

## Internal Block Diagram (IBD)

<b>Definition</b>	InternalBlockDiagram is the concept representing the SysML Internal Block Diagram supported by SCADE Architect. When this class is instantiated from the API, the 'kind' enum takes the value 'IBD'.
<b>Inheritance</b>	InternalBlockDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows an instance-oriented view of the model. An IBD has a frame representing the border of the Block described by that diagram. The ports of this block can be represented on that frame as gates towards the environment. The inside of IBDs

shows the various parts (or references) of the block, providing a white-box view on the block in terms of nested replicas. The ports of the parts/references can also be displayed. Finally, connectors are added to connect the different parts representing the internal structure of the block. An IBD for a top-level "universe" pseudo-block describing the system together with the actors constituting its environment is also called a "Context Diagram".

## Parametric Diagram (PD)

<b>Definition</b>	ParametricDiagram is the concept representing the SysML Parametric Diagram supported by SCADE Architect. When this class is instantiated from the API, the 'kind' enum takes the value 'PD'.
<b>Inheritance</b>	ParametricDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows Constraint expressions for a Block or a Constraint definition. A PD has a frame representing the border of the Block or of the Constraint definition described by the diagram. Owned Data or block parts can be displayed for a "Block" Parametric Diagram using the same formalism as in IBDs. It is possible to display the hierarchy of Blocks with their Data. Constraint can be shown for Block or Constraint definition Parametric Diagram. A constraint is always typed by a Constraint definition. So, any Constraint definition can be composed by several child Constraints. Each Constraint definition can have several Parameters displayed on the border of the figure for a constraint and on the root Parametric Diagram figure for a

Constraint definition. The center of the Constraint figure is reserved to display the Constraint expression that could be an equation that is using Constraint Parameters. Constraint parameters can be linked to any Data using a Binding connector. In this case Data value can be “injected” within the constraint expression.

## Activity Diagram (AD)

<b>Definition</b>	ActivityDiagram is the concept representing the UML Activity Diagram supported by SCADE Architect. When this class is instantiated from the API, the ‘kind’ enum takes the value ‘AD’.
<b>Inheritance</b>	ActivityDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows a logical view of actions related to a Block. These actions can have different kinds and can reference other Behaviors (Activity, Interaction, or State Machine). When a behavior is referenced by a Behavior Action, a diagram icon of the related behavioral diagram is displayed in the Action next to the behavior name. A double click on the Behavior Action opens the behavioral diagram. If no behavior is attached, a double click opens a dialog allowing to create a new Behavior or reference an existing one; after creation the behavioral diagram opens. Any behavior referenced by an Action owned by an Activity must be a sibling of the Activity. Hierarchical behaviors are not supported to simplify the model explorer.

## Sequence Diagram (SD)

<b>Definition</b>	SequenceDiagram is the concept representing the UML Sequence Diagram supported by SCADE Architect. When this class is instantiated from the API, the ‘kind’ enum takes the value ‘SD’.
<b>Inheritance</b>	SequenceDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows a temporal view of execution specifications performed by a Block. A Lifeline represents a Block. These execution specifications can reference other Behaviors (Activity, Interaction, State Machine) or Actions. A Message can be sent between lifelines. Lifelines can be created by palette tools to represent either an Actor or a Block. Actors are stored into the package and can be attached as replica (Property into Block). Moreover, an actor definition can be dragged on the diagram to create a property into the block and a lifeline that represents it. It is also possible to do the same thing with a Block replica. Like in IBDs, it is possible to create a Lifeline that represents a unique Block into the SD.

## State Machine Diagram (SMD)

<b>Definition</b>	StateMachineDiagram is the concept representing the UML State Machine Diagram supported by SCADE Architect. When this class is instantiated from the API, the 'kind' enum takes the value 'SMD'.
<b>Inheritance</b>	StateMachineDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows possible States and Transitions of a Block. Complex State Machine could be managed using Sub-Machine or using State into State. Transitions allow users to type free text (with multi-line support), persistence is performed by a Constraint object that wraps a LiteralString. A State can also be referenced by several StateInvariants even if it is not possible to navigate to all SMDs. All these states can reference a submachine to refine a complex state. They can also reference a doable Activity. Thus, two diagram icons may display in a state if it references an SMD and an AD: a double click on the state opens the related diagram. If both diagrams are defined, a popup chooser opens. If no behavior is attached, a double click opens a dialog allowing to create a new Behavior or reference an existing one; after creation, the behavioral diagram opens. Behavior kinds are limited by default to State Machine and Activity because it does not make sense to reference an Interaction even if the norm allows it. Any behavior referenced by a State (owned by the State

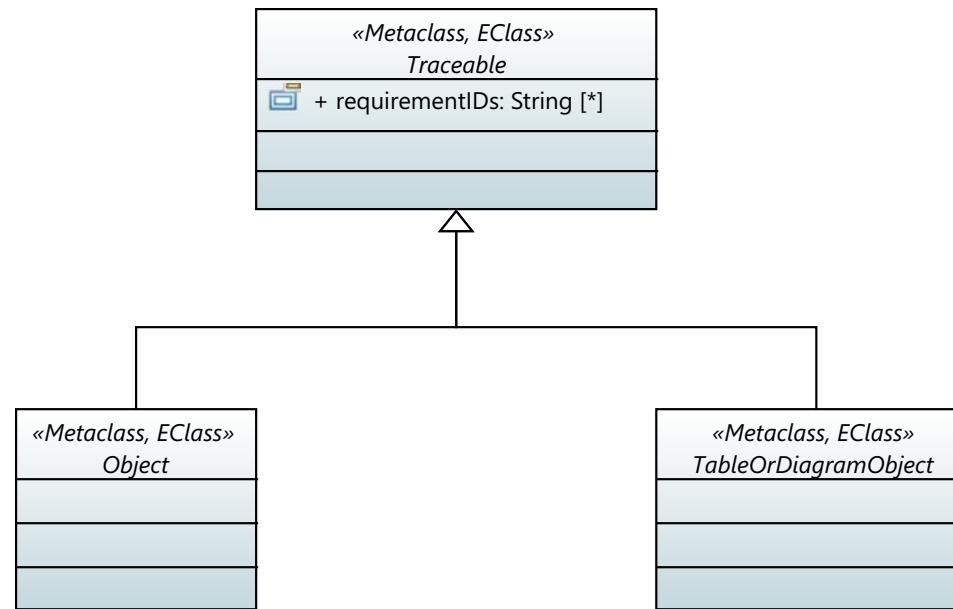
Machine) must be a sibling of the State Machine. Hierarchical behaviors are not supported to simplify the model explorer except for sub-machines.

## Use Case Diagram (UCD)

<b>Definition</b>	UseCaseDiagram is the concept representing the UML Use Case Diagram supported by SCADE Architect. When this class is instantiated from the API, the 'kind' enum takes the value 'UCD'.
<b>Inheritance</b>	UseCaseDiagram → Diagram
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

Such a diagram shows possible Use Cases of a Block (designated as subject), their relations between each other and with other stakeholders (Actors and Blocks). UCDs are used to show all possible usage of the System seen as a black box. It is not currently supported to create UCD for internal operation (white box). Each Use Case can be linked to a behavior used to describe its implementation. Use Cases are mainly linked to Interaction showing the System as a part of the Context Block. Typically, it presents a subject (representing the System Block) that contains several Use Cases. Stakeholders (Actors and Blocks) can be associated to Use Cases. A Use Case can have several subjects if it concerns several Blocks, it should be dropped into several diagrams since it is not possible to clone a Use Case into the same diagram. Alternatively, the attached behavior could be dropped into another subject to create another view of the source Use Case.

# Requirements Traceability



# 8 / Behaviors Metamodels

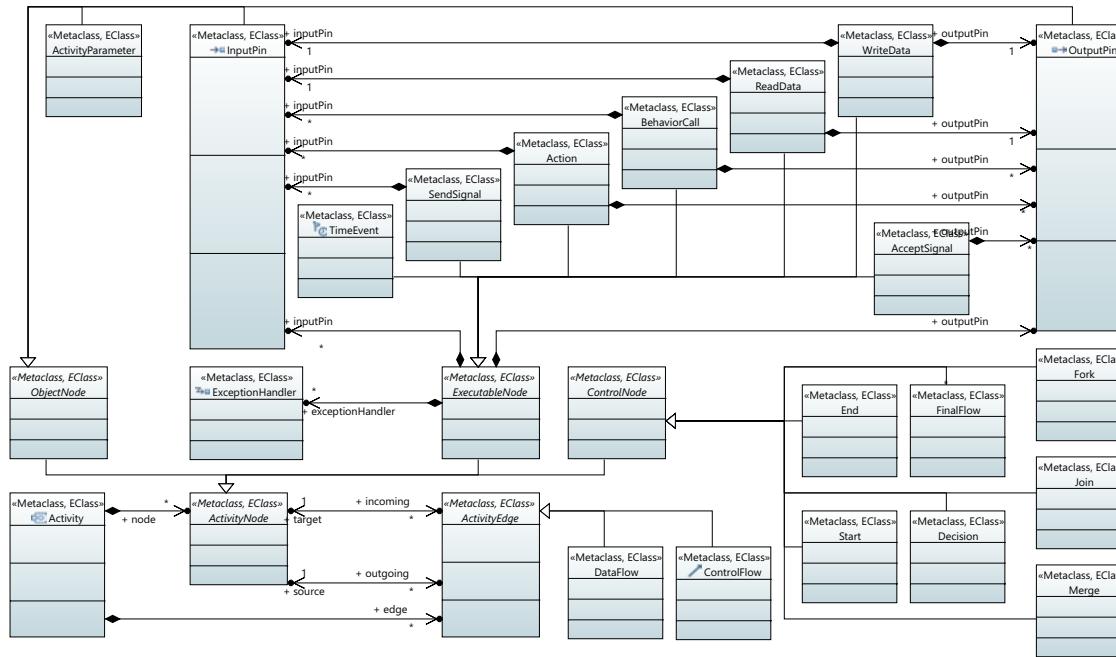
These metamodels present the data structures that give access to SCADE Architect behavior modeling using Python API, Java API, or Tcl API:

- [“Activity”](#)
- [“State Machine”](#)

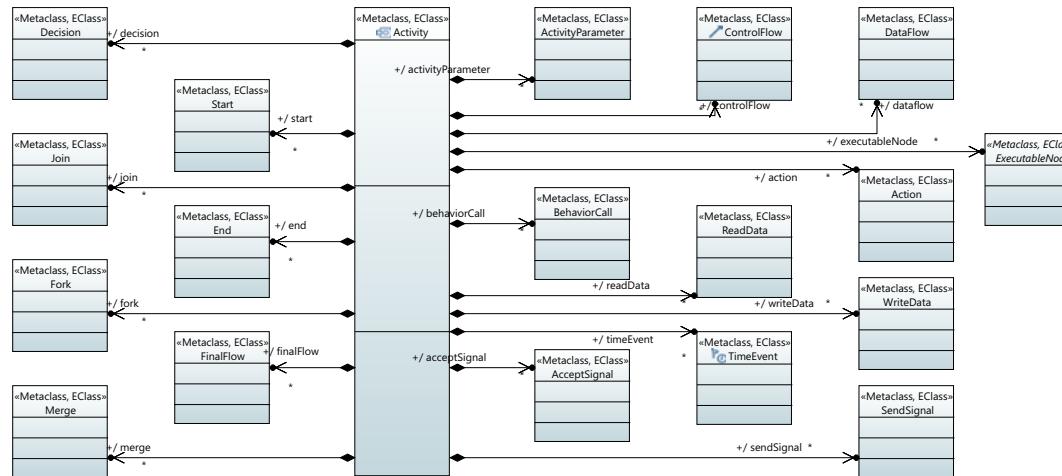
## Activity

The Activity concept relies on the constructs described below.

- [“Activity”](#)
- [“ActivityNode”](#)
- [“ActivityEdge”](#)
- [“ObjectNode”](#)
- [“ExecutableNode”](#)
- [“ControlNode”](#)
- [“AcceptSignal”](#)
- [“Action”](#)
- [“ActivityParameter”](#)
- [“BehaviorCall”](#)
- [“ControlFlow”](#)
- [“DataFlow”](#)
- [“Decision”](#)
- [“End”](#)
- [“ExceptionHandler”](#)
- [“FinalFlow”](#)
- [“Fork”](#)
- [“InputPin”](#)
- [“Join”](#)
- [“Merge”](#)
- [“OutputPin”](#)
- [“ReadData”](#)
- [“SendSignal”](#)
- [“Start”](#)
- [“TimeEvent”](#)
- [“WriteData”](#)



The above metamodel diagram gives the list of all activity nodes and the second provides subsets.



## Activity

### Definition

An Activity is a kind of Behavior that is specified as a graph of nodes interconnected by edges. A subset of the nodes are executable nodes that embody lower-level steps in the overall Activity. Object nodes hold data that is input to and output from executable nodes, and moves across object flow edges. Control nodes specify sequencing of executable nodes via control flow edges.

### Inheritance

Behavior → NamedElement

### Attributes

n/a

node: ActivityNode[\*]

All the nodes of the activity

edge: ActivityEdge[\*]

All the edges of the activity

/acceptSignal: AcceptSignal[\*] {derived, subsets executableNode}

The Accept Signal actions of the activity

/action: Action[\*] {derived, subsets executableNode}

The actions of the activity

/activityParameter: ActivityParameter[\*] {derived, subsets node}

The parameters of the activity

/behaviorCall: BehaviorCall[\*] {derived, subsets executableNode}

The behavior call actions of the activity

/controlFlow: ControlFlow[\*] {derived, subsets edge}

The control flows of the activity

### Associations

/dataFlow: DataFlow[\*] {derived, subsets edge}

The data flows of the activity

/decision: Decision[\*] {derived, subsets node}

The decision flows of the activity

/executableNode: ExecutableNode[\*] {derived, subsets node}

The executables nodes of the activity

/end: End[\*] {derived, subsets node}

The end flows of the activity

/finalFlow: FinalFlow[\*] {derived, subsets node}

The final flows of the activity

/fork: Fork[\*] {derived, subsets node}

The fork flows of the activity

/join: Join[\*] {derived, subsets node}

The join flows of the activity

/merge: Merge[\*] {derived, subsets node}

The merge flows of the activity

/readData: ReadData[\*] {derived, subsets executableNode}

The Read of Data actions of the activity

/sendSignal: SendSignal[\*] {derived, subsets executableNode}

The Send Signal actions of the activity

/start: start[\*] {derived, subsets node}

The start flows of the activity

/timeEvent: TimeEvent[\*] {derived, subsets executableNode}

The time events of the activity

/writeData: WriteData[\*] {derived, subsets executableNode}

The Write Data actions of the activity

### Associations

## ActivityNode

<b>Definition</b>	ActivityNodes are used to model the individual steps in the behavior specified by an Activity.
<b>Inheritance</b>	NamedElement → Annotable
<b>Attributes</b>	n/a
<b>Associations</b>	<code>outgoing: ActivityEdge[ * ]</code> Outgoing edges of the node <code>incoming: ActivityEdge[ * ]</code> IncomingEdges of the node

## ActivityEdge

<b>Definition</b>	An ActivityEdge is a directed connection between two ActivityNodes along which tokens may flow, from the source ActivityNode to the target ActivityNode.
<b>Inheritance</b>	NamedElement → Annotable
<b>Attributes</b>	n/a
<b>Associations</b>	<code>source: ActivityNode</code> Source of the edge <code>target: ActivityNode</code> Target of the edge

## ObjectNode

<b>Definition</b>	An ObjectNode is a kind of ActivityNode used to hold value-containing object tokens during the course of the execution of an Activity.
<b>Inheritance</b>	ActivityNode → NamedElement TypedElement → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## ExecutableNode

<b>Definition</b>	An ExecutableNode is a kind of ActivityNode that may be executed as a step in the overall desired behavior of the containing Activity.
<b>Inheritance</b>	ActivityNode → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	<code>exceptionHandler: ExceptionHandler[ * ]</code> ExceptionHandler of the node <code>inputPin: InputPin[ * ]</code> Input pins of the node <code>outputPin: OutputPin[ * ]</code> Output pins of the node

## ControlNode

<b>Definition</b>	A ControlNode is a kind of ActivityNode used to manage the flow of tokens between other nodes in an Activity.
<b>Inheritance</b>	ActivityNode → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## AcceptSignal

<b>Definition</b>	An AcceptSignal is an action that waits for the occurrence of one Signal event.
<b>Inheritance</b>	ExecutableNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	<p>signal: Signal The signal the action is waiting for</p> <p>outputPin: OutputPin[*] Output pins of the action</p>

## Action

<b>Definition</b>	An Action is a fundamental unit of executable functionality whose specification may be given in a textual concrete syntax.
<b>Inheritance</b>	ExecutableNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	<p>inputPin: InputPin[*] Input pins of the action</p> <p>outputPin: OutputPin[*] Output pins of the action</p>

## ActivityParameter

<b>Definition</b>	An ActivityParameter is an ObjectNode for accepting values from the input Parameters or providing values to the output Parameters of an Activity.
<b>Inheritance</b>	ObjectNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## BehaviorCall

<b>Definition</b>	A BehaviorCall is an Action that results in the invocation of a Behavior.
<b>Inheritance</b>	ExecutableNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	<p>behavior: Behavior The behavior invoked by the action</p> <p>inputPin: InputPin[*] Input pins of the action</p> <p>outputPin: OutputPin[*] Output pins of the action</p>

## ControlFlow

<b>Definition</b>	A ControlFlow is an ActivityEdge traversed by control tokens or object tokens of control type, which are used to control the execution of ExecutableNodes.
<b>Inheritance</b>	ActivityEdge → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## DataFlow

<b>Definition</b>	A DataFlow is an ActivityEdge that is traversed by object tokens that may hold values.
<b>Inheritance</b>	ActivityEdge → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Decision

<b>Definition</b>	A Decision is a ControlNode that chooses between outgoing ActivityEdges for the routing of tokens.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## End

<b>Definition</b>	An End is an abstract ControlNode that terminates the execution of its owning Activity.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## ExceptionHandler

<b>Definition</b>	An ExceptionHandler is an Element that specifies a handlerBody ExecutableNode to execute in case the specified exception occurs during the execution of the protected ExecutableNode.
<b>Inheritance</b>	Annotable → Element
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## FinalFlow

<b>Definition</b>	A FinalFlow is a ControlNode that terminates a flow by consuming the tokens offered to it.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Fork

<b>Definition</b>	A Fork is a ControlNode that splits a flow into multiple concurrent flows.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## InputPin

<b>Definition</b>	An InputPin is an ObjectNode that holds input values to be consumed by an Action.
<b>Inheritance</b>	ObjectNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Join

<b>Definition</b>	A Join is a ControlNode that synchronizes multiple flows.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Merge

<b>Definition</b>	A Merge is a ControlNode that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## OutputPin

<b>Definition</b>	An OutputPin is a Pin that holds output values produced by an Action.
<b>Inheritance</b>	ObjectNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## ReadData

<b>Definition</b>	A ReadData is an Action that retrieves the values of a StructuralFeature.
<b>Inheritance</b>	ExecutableNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	<p><code>data: Data</code> The data to be read</p> <p><code>inputPin: InputPin</code> Input pin of the action</p> <p><code>outputPin: OutputPin</code> Output pin of the action</p>

## SendSignal

<b>Definition</b>	A SendSignal is an Action that creates a Signal instance and transmits it to the target object. Values from the argument InputPins are used to provide values for the attributes of the Signal. The requestor continues execution immediately after the Signal instance is sent out and cannot receive reply values.
<b>Inheritance</b>	ExecutableNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	<p><code>signal: Signal</code> The signal to be sent by the action</p> <p><code>inputPin: InputPin[ * ]</code> Input pins of the action</p>

## Start

<b>Definition</b>	An Start is a ControlNode that offers a single control token when initially enabled.
<b>Inheritance</b>	ControlNode → ActivityNode
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## TimeEvent

<b>Definition</b>	A TimeEvent is an action that waits for the occurrence of a time event.
<b>Inheritance</b>	ExecutableNode → ActivityNode
<b>Attributes</b>	time: String
<b>Associations</b>	n/a

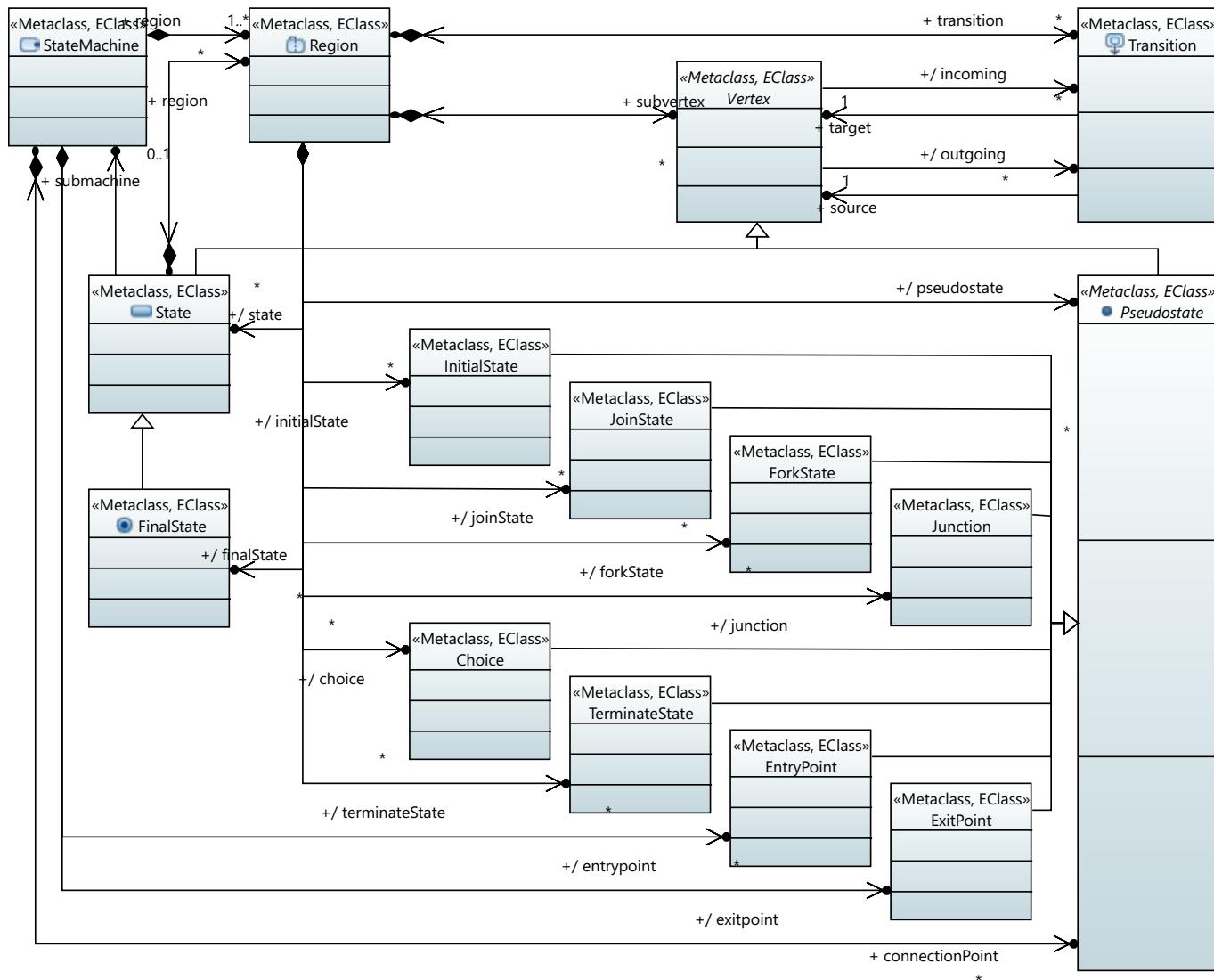
## WriteData

<b>Definition</b>	A WriteData is an Action for adding values to a StructuralFeature.
<b>Inheritance</b>	ActivityNode → NamedElement
<b>Attributes</b>	n/a
	<p>data: Data The data to be added</p> <p>inputPin: InputPin Input pin of the action</p> <p>outputPin: OutputPin Output pin of the action</p>

## State Machine

The StateMachine concept relies on the constructs described below.

- ["StateMachine"](#)
- ["Choice"](#)
- ["EntryPoint"](#)
- ["ExitPoint"](#)
- ["FinalState"](#)
- ["ForkState"](#)
- ["InitialState"](#)
- ["JoinState"](#)
- ["Junction"](#)
- ["Pseudostate"](#)
- ["Region"](#)
- ["State"](#)
- ["TerminateState"](#)
- ["Transition"](#)
- ["Vertex"](#)



## StateMachine

<b>Definition</b>	StateMachines can be used to express event-driven behaviors of parts of a system. Behavior is modeled as a traversal of a graph of Vertices interconnected by one or more joined Transition arcs that are triggered by the dispatching of successive Event occurrences. During this traversal, the StateMachine may execute a sequence of Behaviors associated with various elements of the StateMachine.
<b>Inheritance</b>	Behavior → NamedElement
<b>Attributes</b>	n/a
	<code>connectionPoint: Pseudostate[*]</code> Connection points defined for this StateMachine
<b>Associations</b>	<code>region: Region[1..*]</code> Regions owned by the StateMachine
<b>Choice</b>	
<b>Definition</b>	Choice is used to realize a dynamic conditional branch.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## EntryPoint

<b>Definition</b>	An EntryPoint represents an entry point for a StateMachine or a composite State.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## ExitPoint

<b>Definition</b>	An ExitPoint represents an exit point for a StateMachine or a composite State.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## FinalState

<b>Definition</b>	A special kind of State, which, when entered, signifies that the enclosing Region has completed.
<b>Inheritance</b>	State → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## ForkState

<b>Definition</b>	ForkStates split an incoming Transition into two or more Transitions.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## InitialState

<b>Definition</b>	An initialState represents a starting point for a Region.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## JoinState

<b>Definition</b>	JoinState is a common target Vertex for two or more Transitions.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Junction

<b>Definition</b>	A Junction connects multiple Transitions.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Pseudostate

<b>Definition</b>	A Pseudostate is an abstraction that encompasses different types of transient Vertices in the StateMachine graph.
<b>Inheritance</b>	Vertex → NamedElement
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Region

<b>Definition</b>	A Region is a top-level part of a StateMachine, that serves as a container for the Vertices and Transitions of the StateMachine. A StateMachine may contain multiple Regions representing behaviors that may occur in parallel.
<b>Inheritance</b>	NamedElement → Annotable
<b>Attributes</b>	n/a  subvertex: Vertex[*] Set of Vertices owned by the Region  transition: Transition[*] Set of Transitions owned by the Region  /state: State[*] {derived, subsets subvertex} Set of States owned by the Region  /finalState: FinalState[*] {derived, subsets /state} Set of FinalStates that are owned by this Region.  /pseudostate: Pseudostate[*] {derived, subsets subvertex} Set of Pseudostates that are owned by this Region.  /initialState: InitialState[*] {derived, subsets pseudostate } Set of InitialStates that are owned by this Region.  /joinState: JoinState[*] {derived, subsets pseudostate } Set of JoinStates that are owned by this Region.  /forkState: ForkState[*] {derived, subsets pseudostate } Set of ForkStates that are owned by this Region.  /junction: Junction[*] {derived, subsets pseudostate } Set of Junctions that are owned by this Region.  /choice: Choice[*] {derived, subsets pseudostate } Set of Choices that are owned by this Region.  /terminateState: TerminateState[*] {derived, subsets pseudostate } Set of TerminateStates that are owned by this Region.
<b>Associations</b>	

## State

<b>Definition</b>	A State models a situation during which some (usually implicit) invariant condition holds.
<b>Inheritance</b>	Vertex → NamedElement
<b>Attributes</b>	n/a  region: Region[*] Regions owned directly by the State
<b>Associations</b>	submachine: StateMachine[0..1] StateMachine to be inserted in place of the (submachine) State

## TerminateState

<b>Definition</b>	Entering a TerminateState implies that the execution of the StateMachine is terminated immediately.
<b>Inheritance</b>	Pseudostate → Vertex
<b>Attributes</b>	n/a
<b>Associations</b>	n/a

## Transition

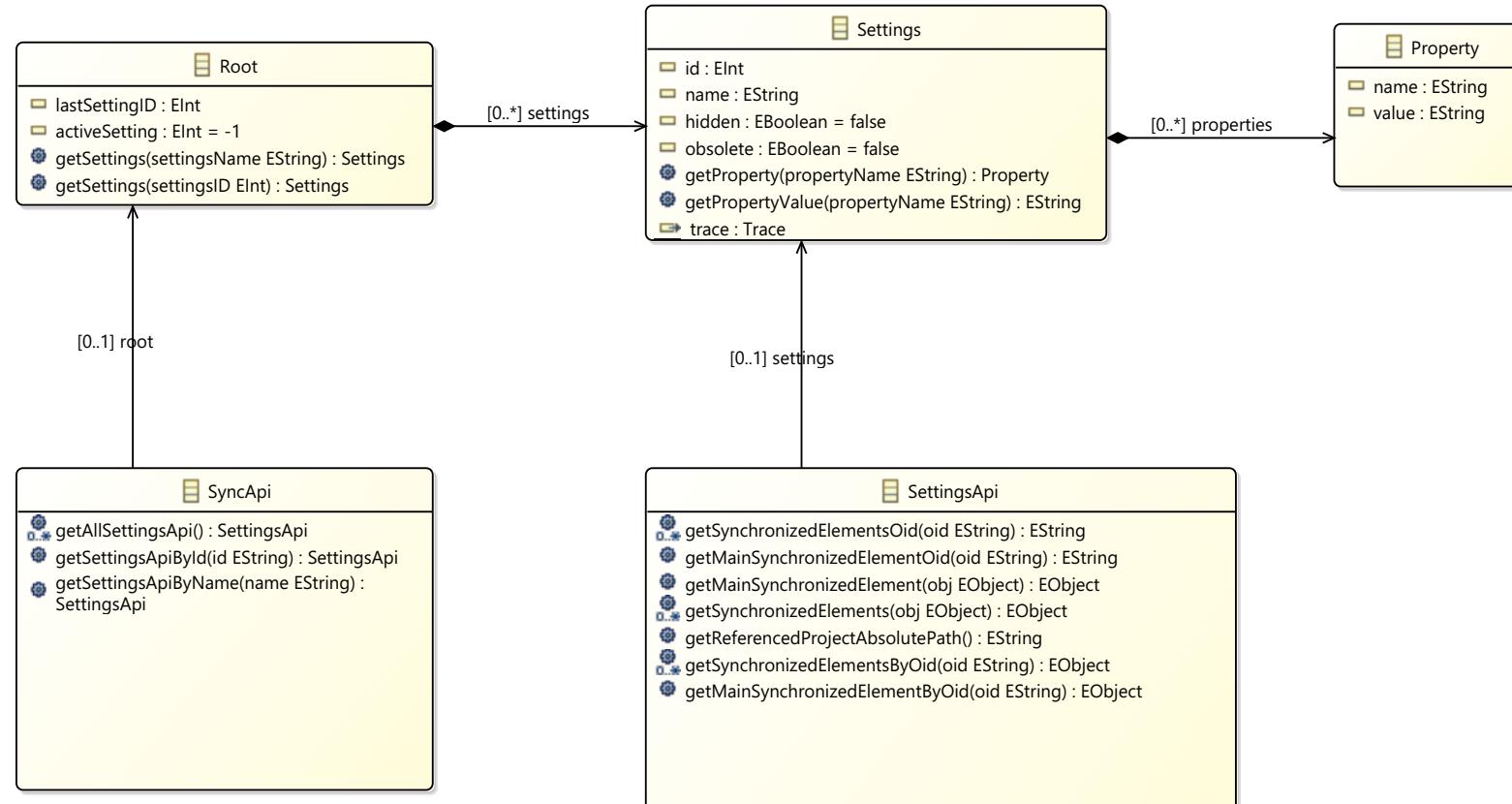
<b>Definition</b>	A Transition represents an arc between exactly one source Vertex and exactly one Target vertex (the source and targets may be the same Vertex).
<b>Inheritance</b>	NamedElement → Annotable
<b>Attributes</b>	<p><code>condition: String</code> Constraint specification providing a fine-grained control over the firing of the Transition</p> <p><code>effect: String</code> Specifies an optional behavior to be performed when the Transition fires</p> <p><code>trigger: String</code> Specifies the Trigger that may fire the transition</p>
<b>Associations</b>	<p><code>source: Vertex</code> Designates the originating Vertex (State or Pseudostate) of the Transition</p> <p><code>target: Vertex</code> Designates the target Vertex that is reached when the Transition is taken</p> <p><code>parent: Region</code> Designates the Region that owns this Transition</p>

## Vertex

<b>Definition</b>	A Vertex is an abstraction of a node in a StateMachine graph. It can be the source or destination of any number of Transitions.
<b>Inheritance</b>	NamedElement → Annotable
<b>Attributes</b>	n/a
<b>Associations</b>	<p><code>/incoming: Transition[*] {derived}</code> Specifies the Transitions entering this Vertex</p> <p><code>/outgoing: Transition[*] {derived}</code> Specifies the Transitions departing from this Vertex</p> <p><code>parent: Region</code> The Region that contains this Vertex</p>

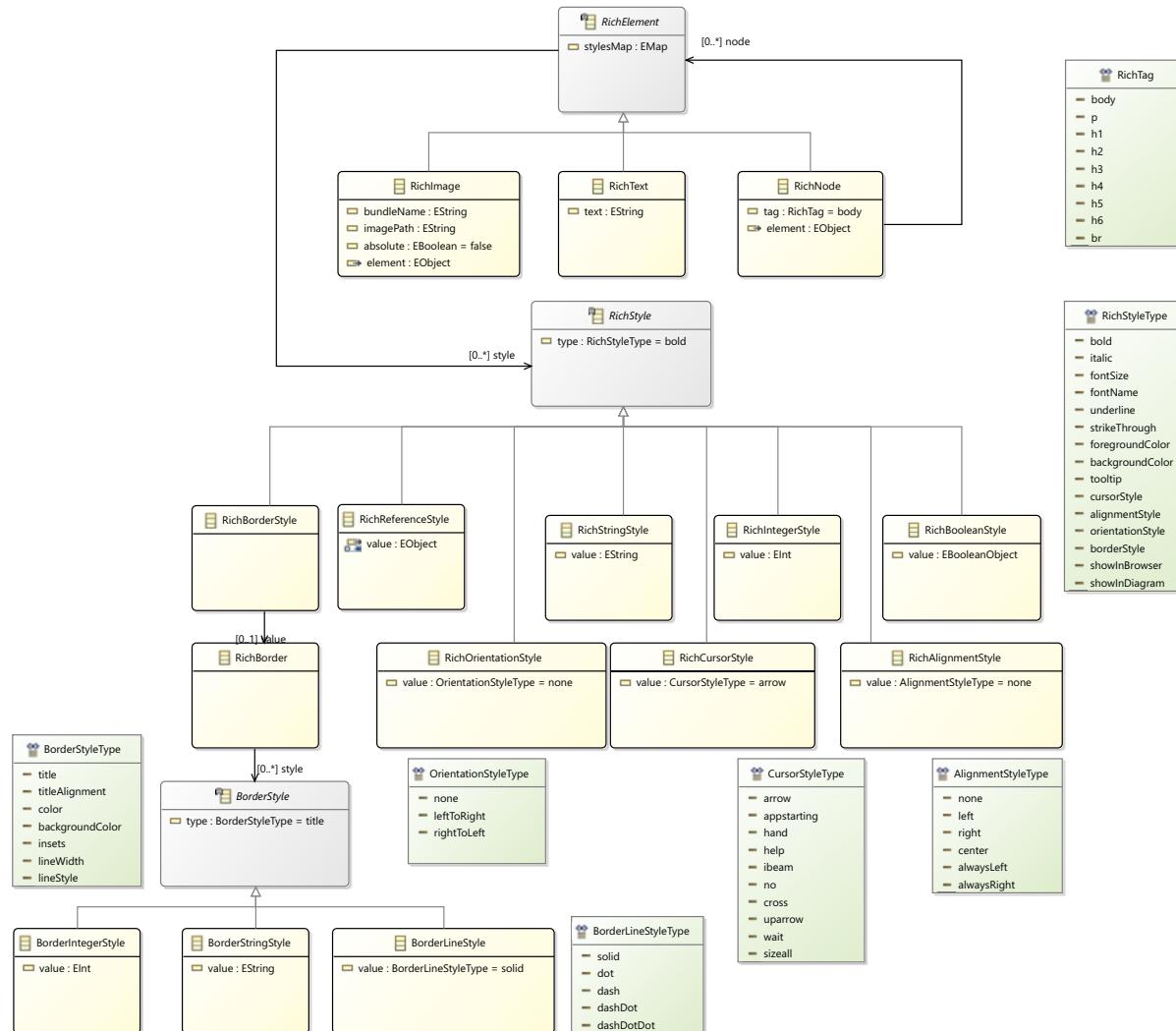
# 9 / Synchronization Metamodel

This metamodel presents the data structures that give access to SCADE Architect synchronization information using Python API:



# 10 /RichText Metamodel

This metamodel presents the data structures that give access to SCADE Architect rich text information available in diagrams using Python API:



## Part 2

# SCADE Suite Metamodels for Python API

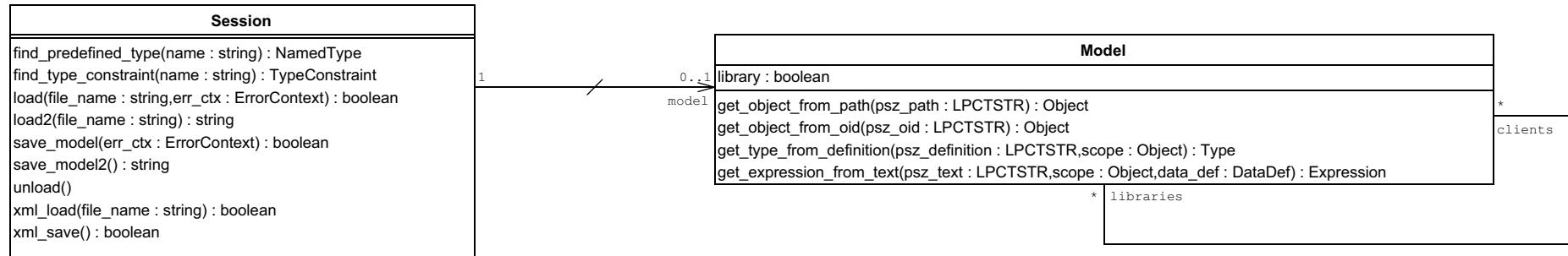
- 11/ ["Editor Metamodels \(Python API\)"](#)
- 12/ ["Annotation Metamodels \(Python API\)"](#)
- 13/ ["Timing and Stack Analysis Metamodel \(Python API\)"](#)
- 14/ ["Graphical Panel Coupling Metamodel \(Python API\)"](#)

# 11 /Editor Metamodels (Python API)

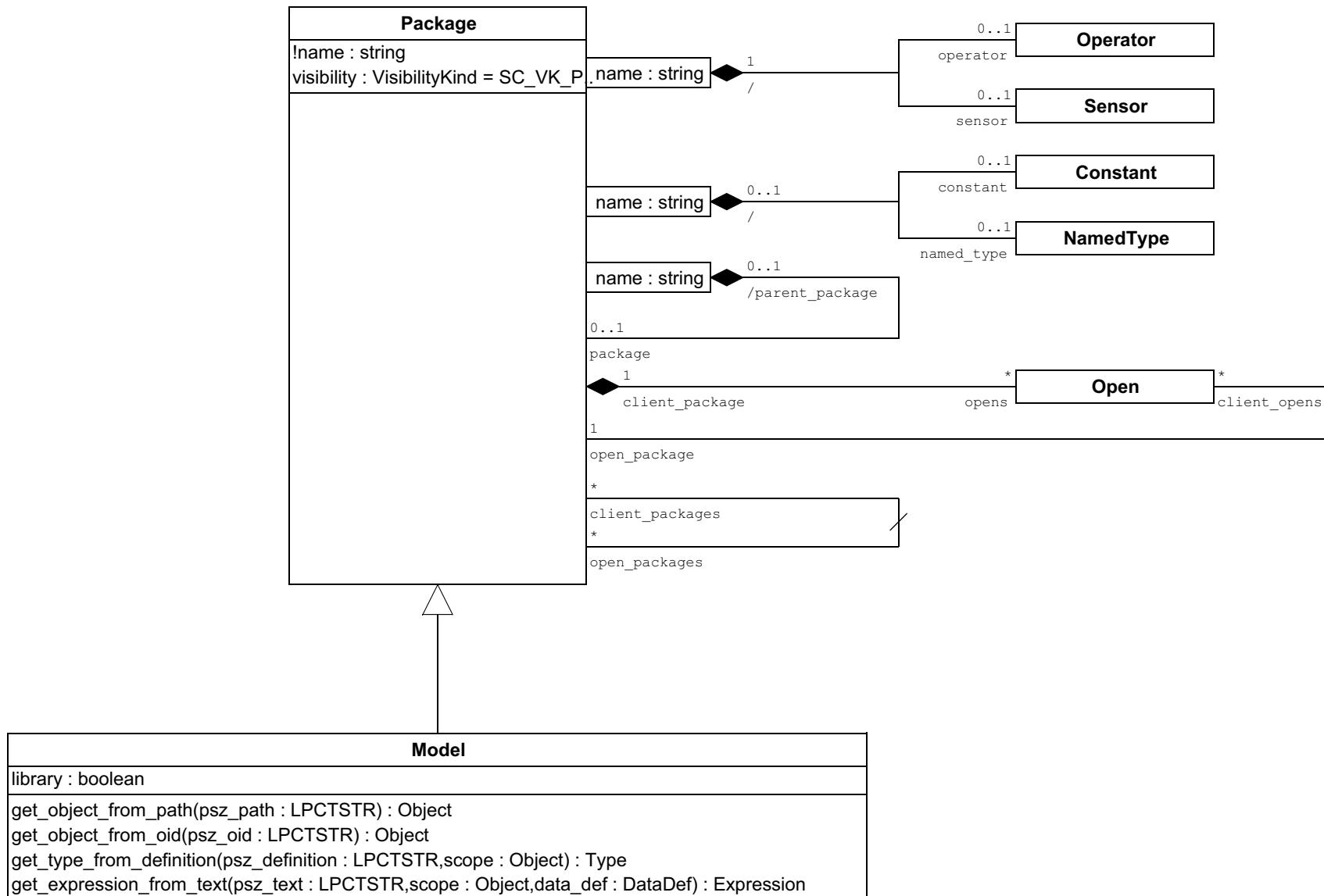
These metamodels present the data structures that give access to SCADE Suite Editor environment using Python API:

- ["Session \(Python\)"](#)
- ["Model \(Python\)"](#)
- ["Variable \(Python\)"](#)
- ["Operator \(Python\)"](#)
- ["Expression \(Python\)"](#)
- ["Type \(Python\)"](#)
- ["Data Definition \(Python\)"](#)
- ["Data Flow \(Python\)"](#)
- ["Model Storage - IO \(Python\)"](#)
- ["Composition \(Python\)"](#)
- ["Inheritance \(Python\)"](#)
- ["State Machine \(Python\)"](#)
- ["ActivateBlock \(Python\)"](#)
- ["Specialization \(Python\)"](#)
- ["Expression Map \(Python\)"](#)
- ["Model and Package Maps \(Python\)"](#)
- ["Annotable \(Python\)"](#)
- ["Pragmas \(Python\)"](#)
- ["Scopes \(Python\)"](#)
- ["Graphical \(Python\)"](#)

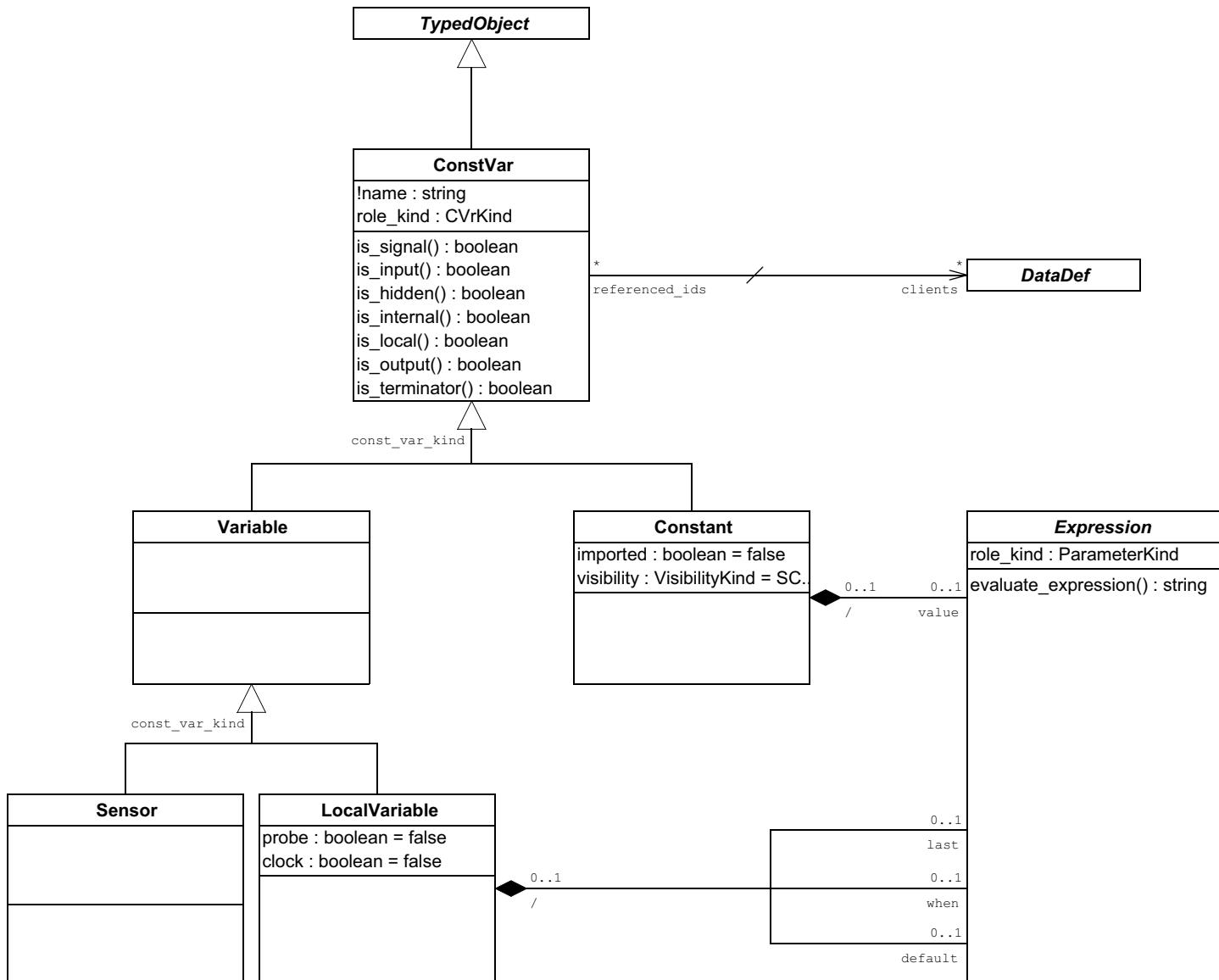
# Session (Python)



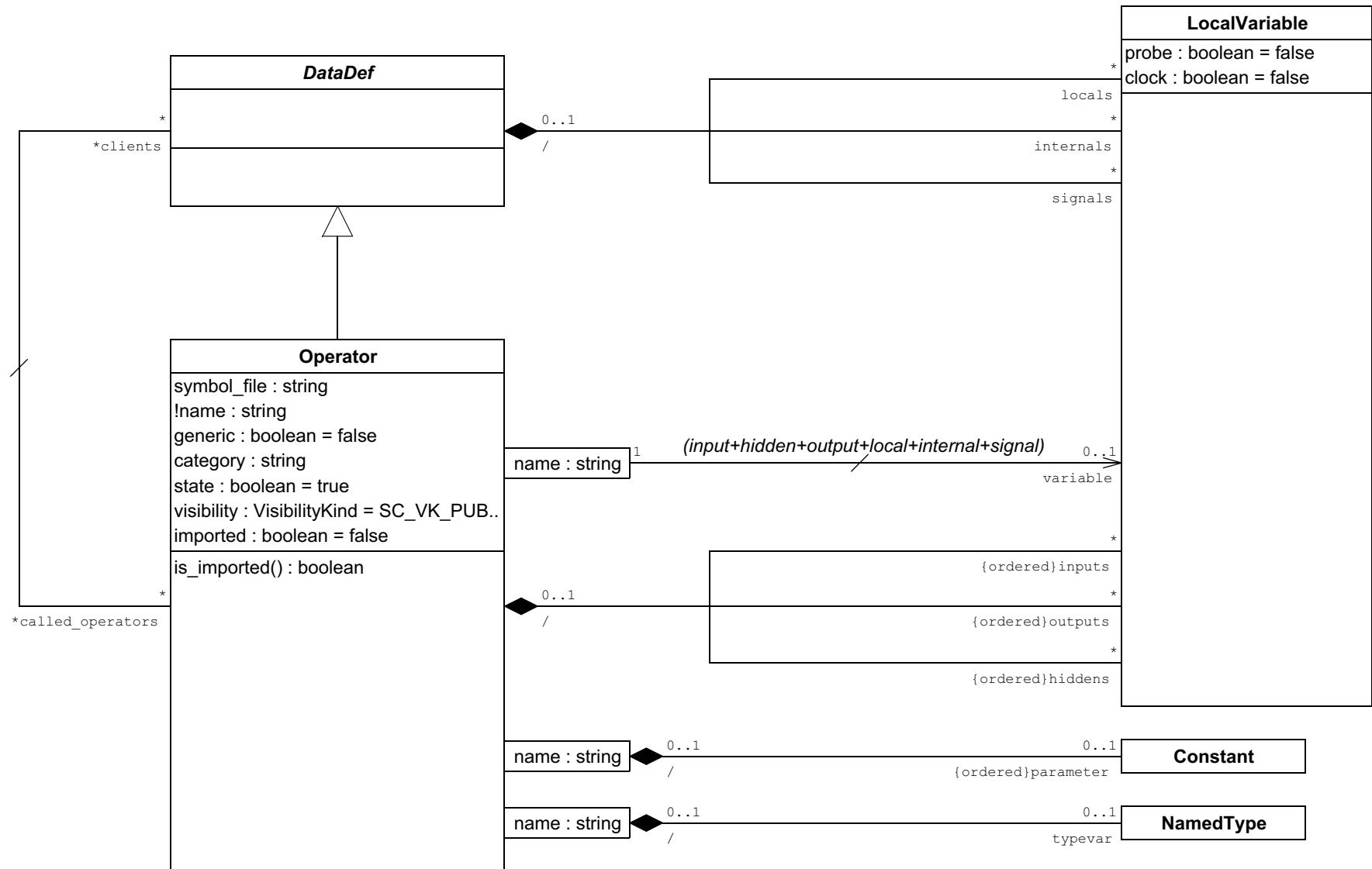
# Model (Python)



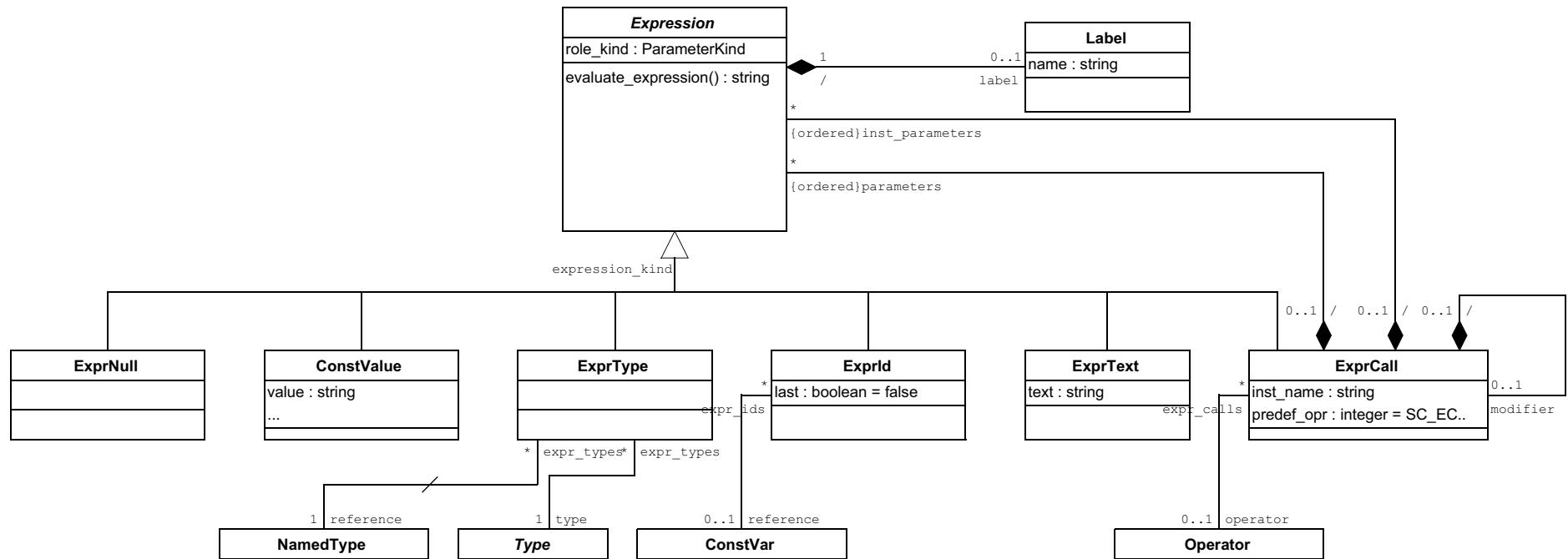
# Variable (Python)



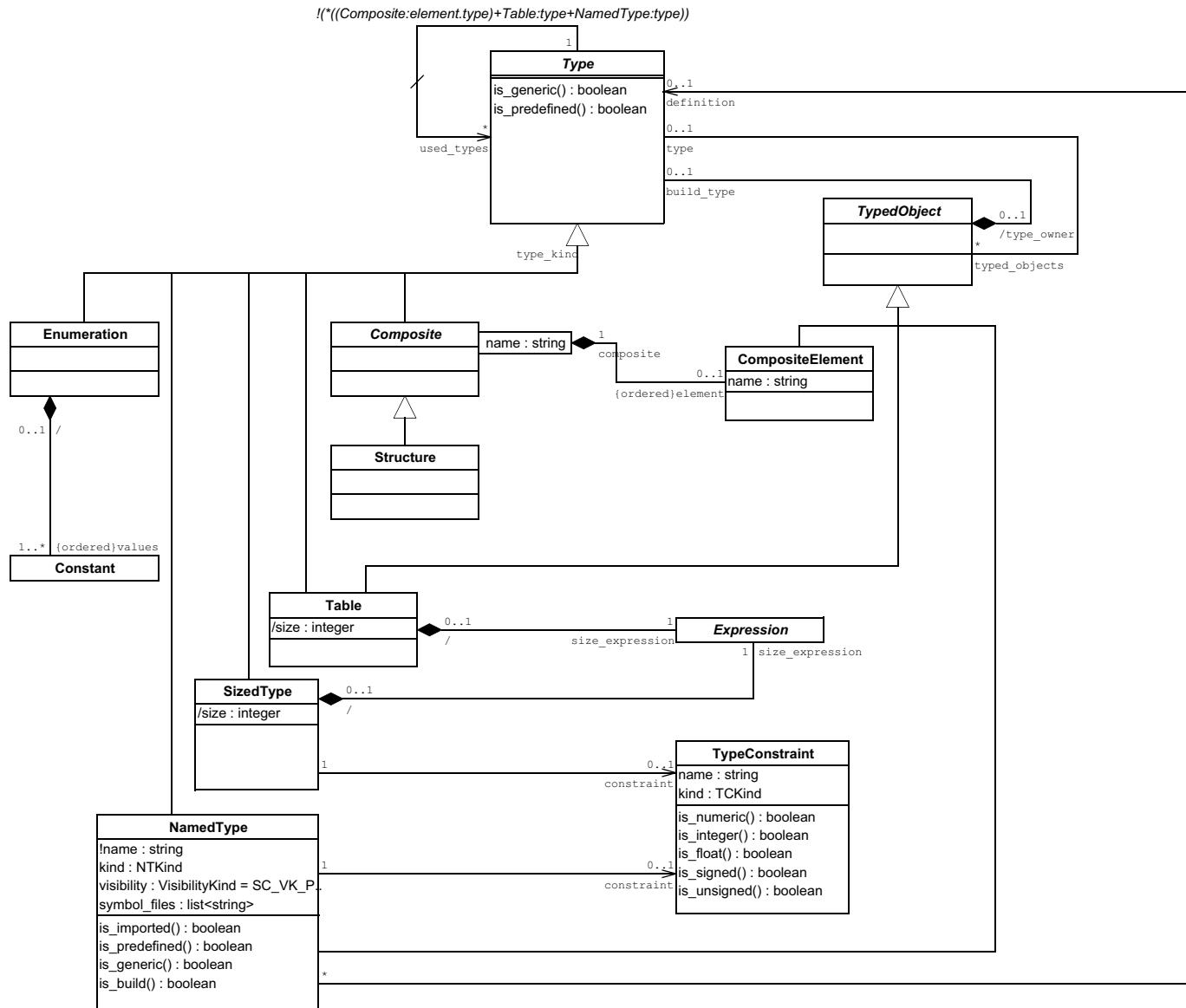
# Operator (Python)



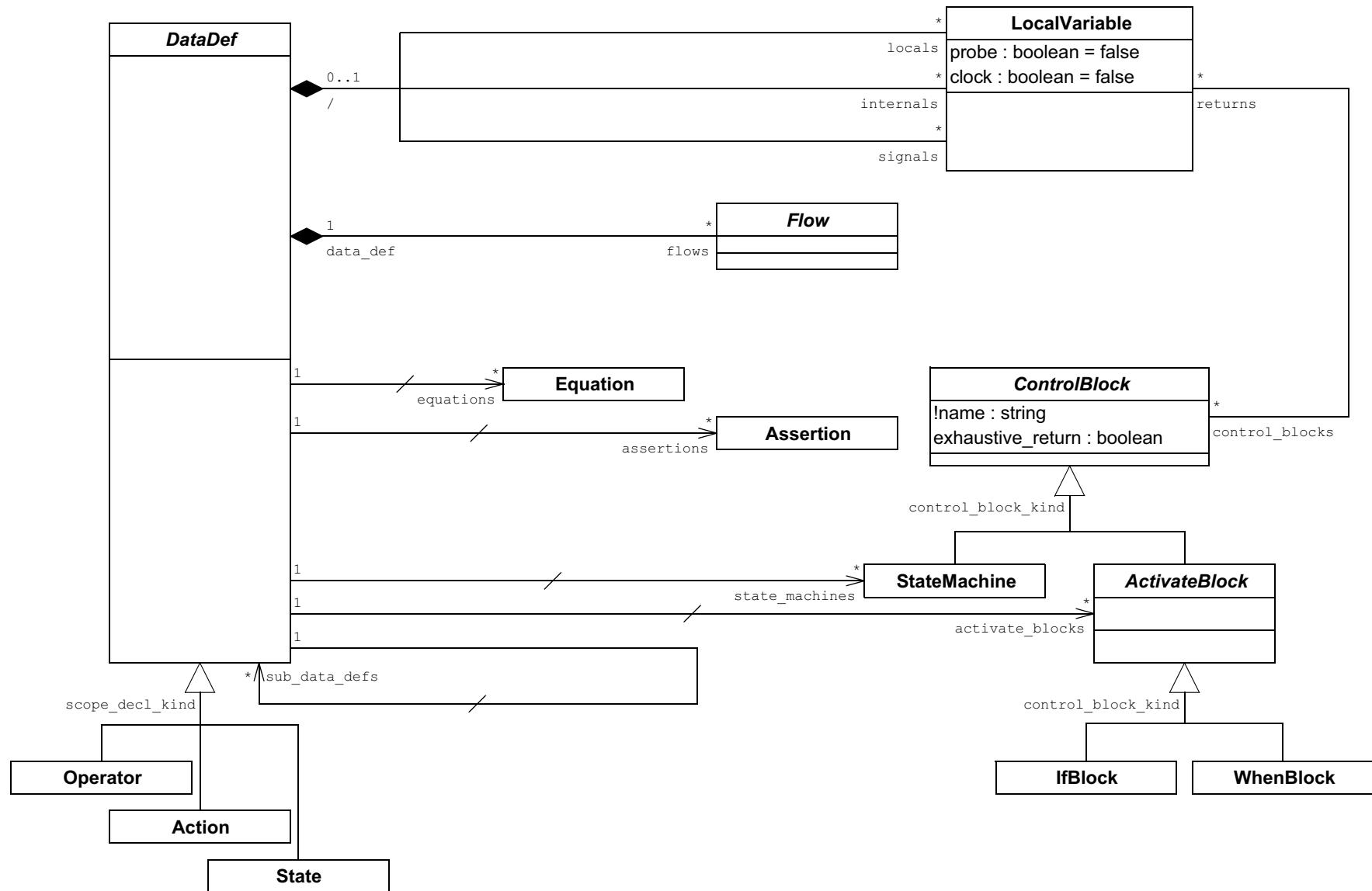
# Expression (Python)



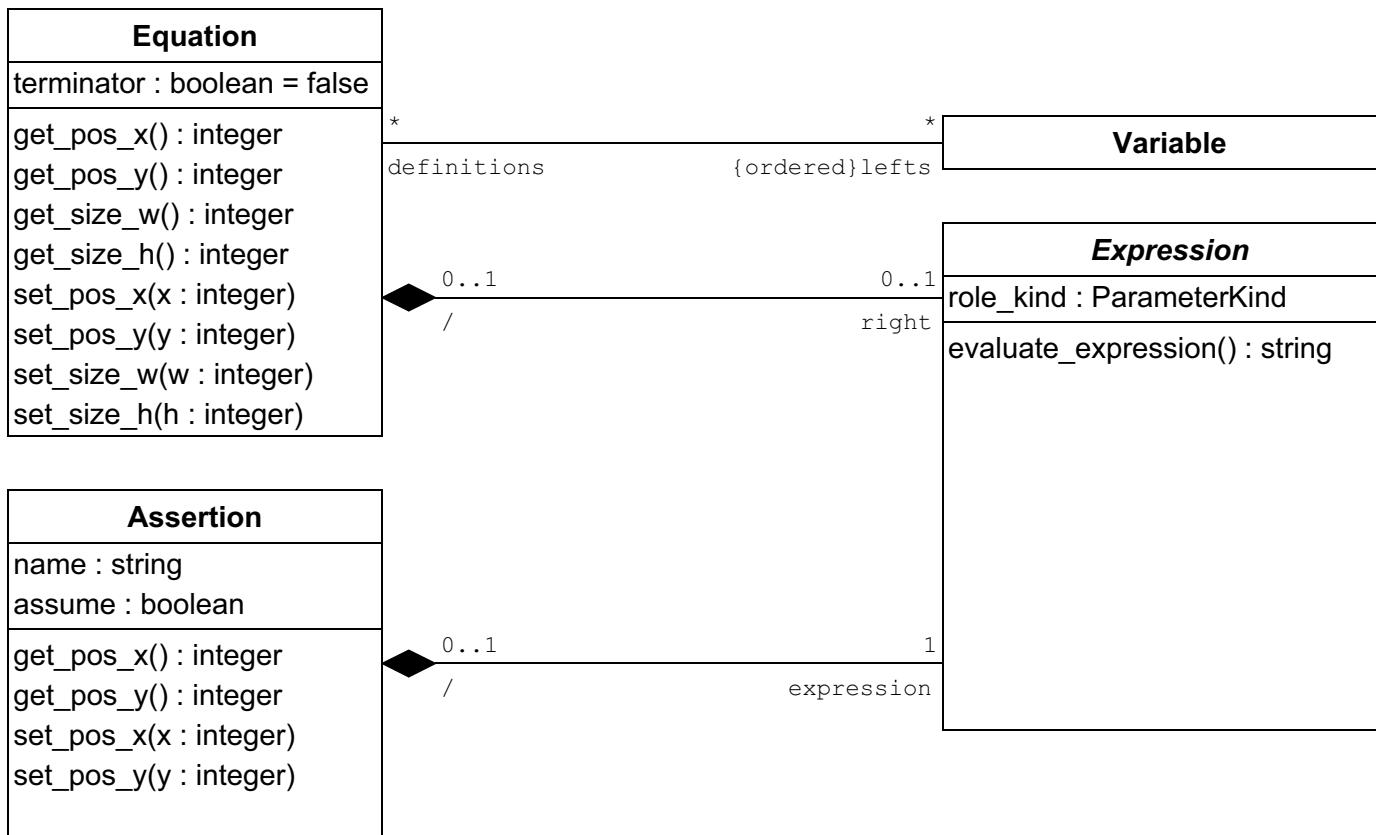
# Type (Python)



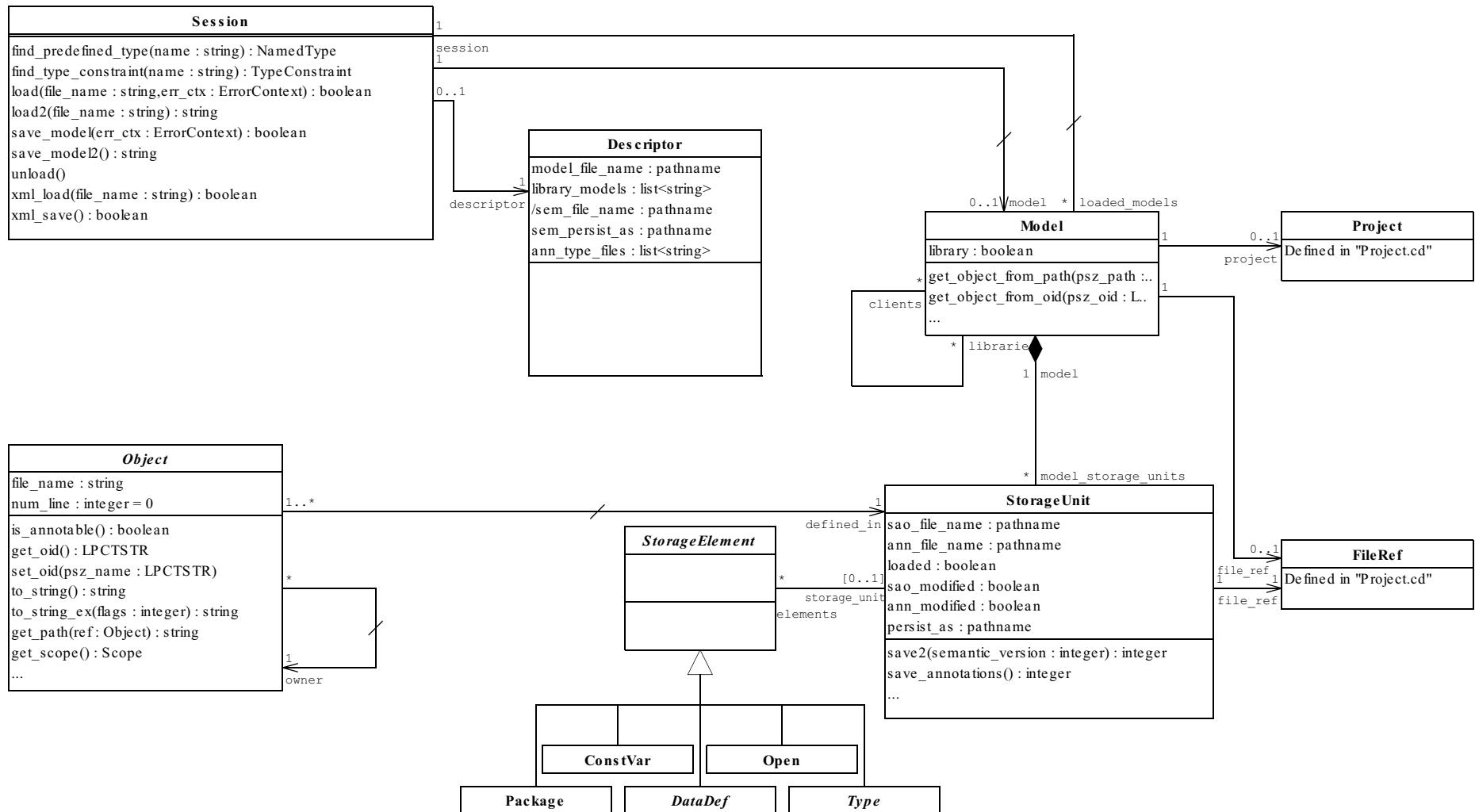
# Data Definition (Python)



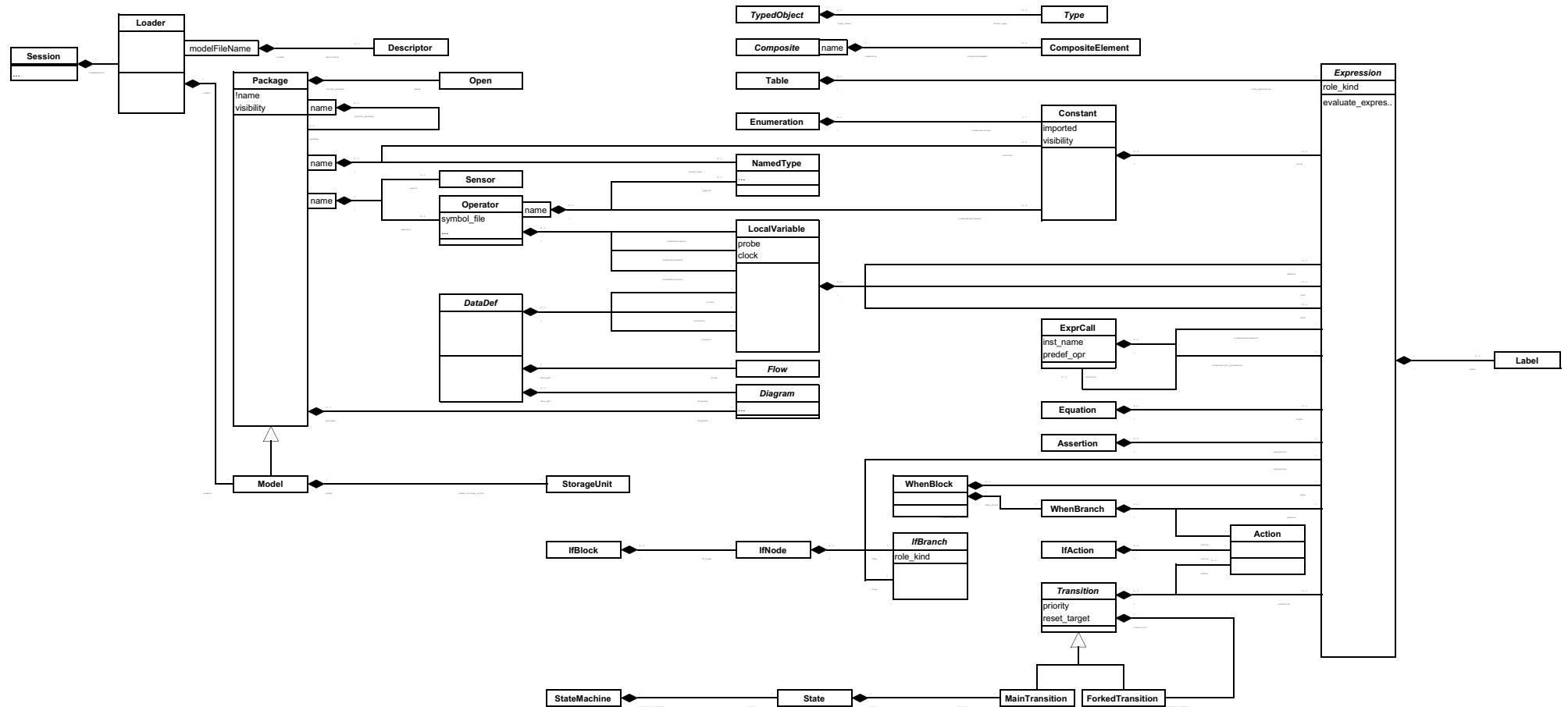
# Data Flow (Python)



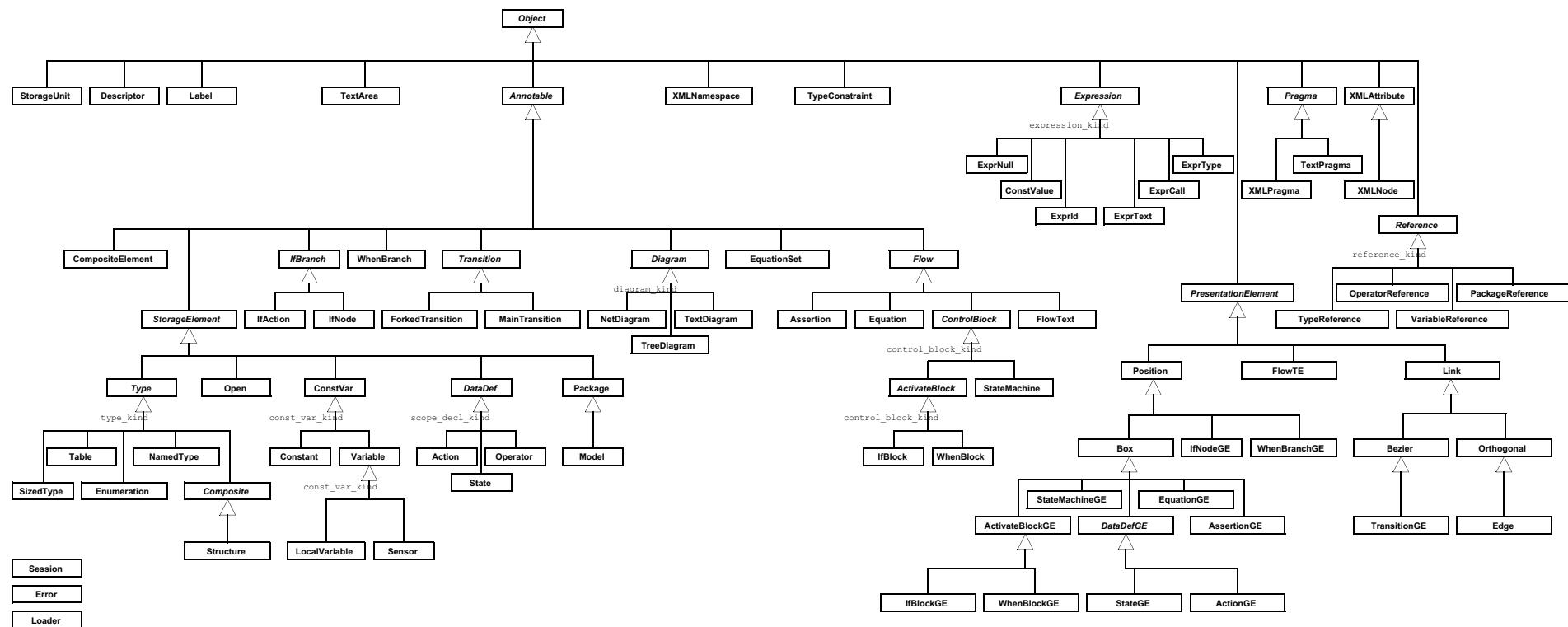
# Model Storage - IO (Python)



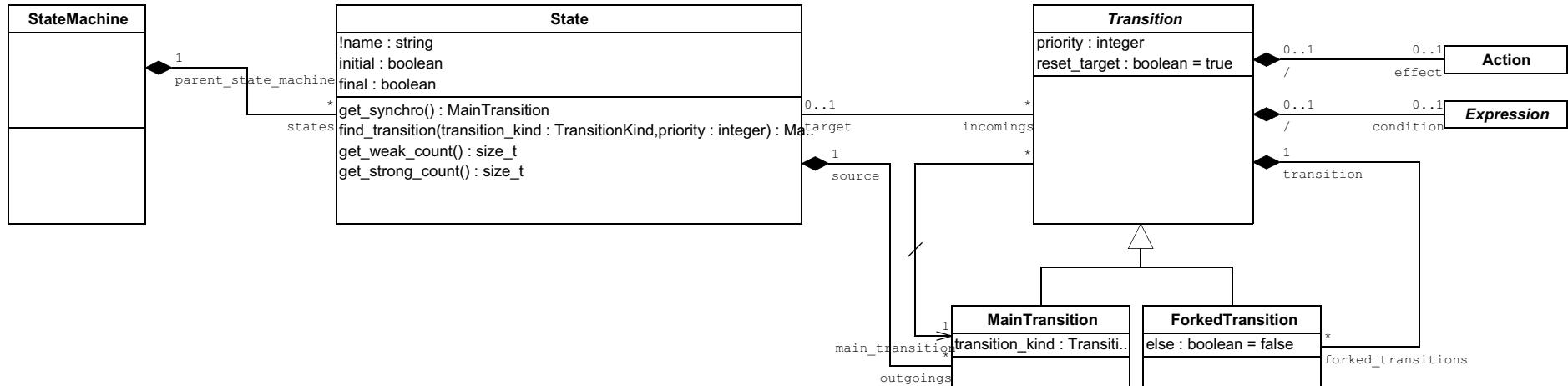
# Composition (Python)



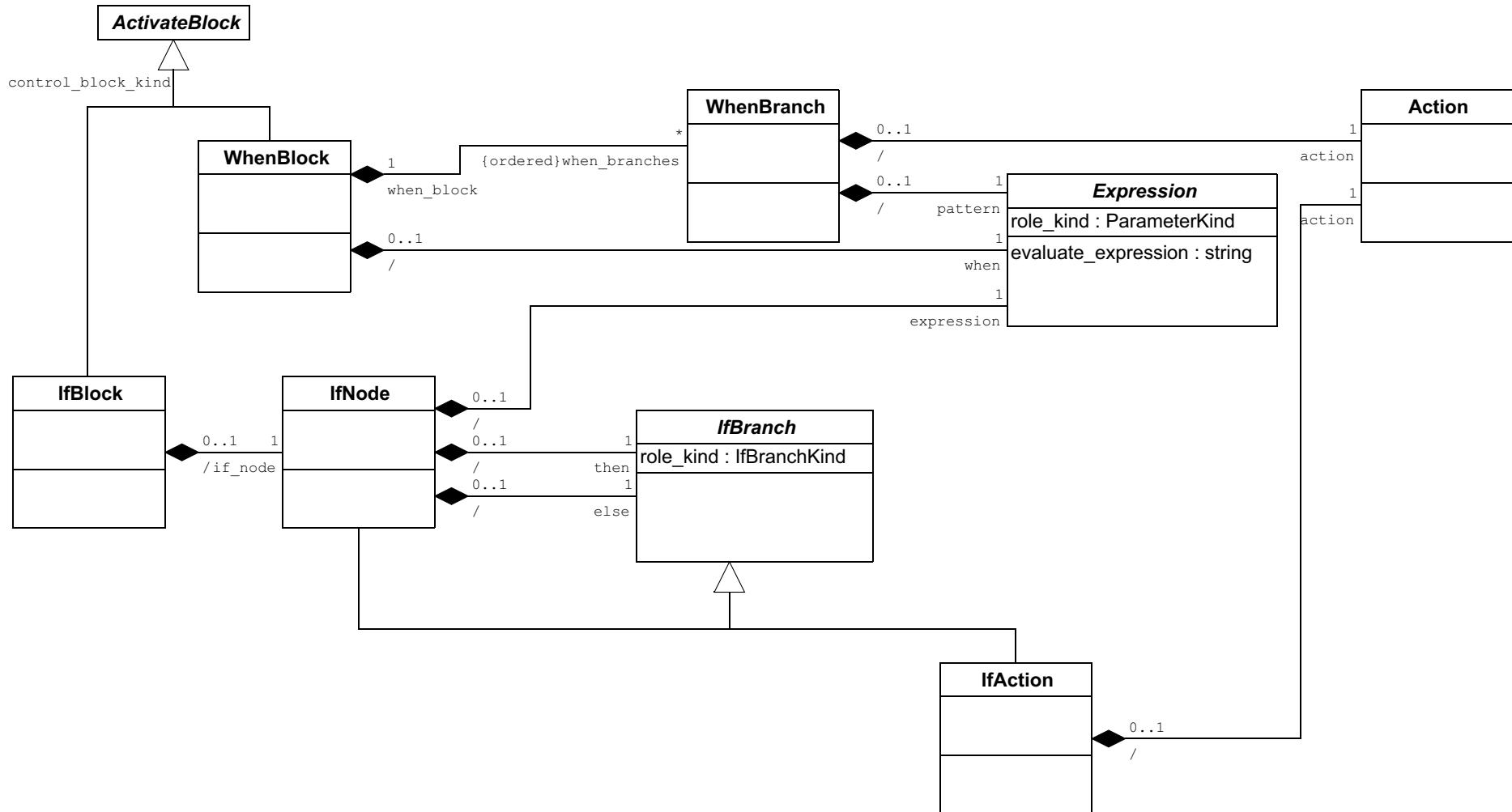
# Inheritance (Python)



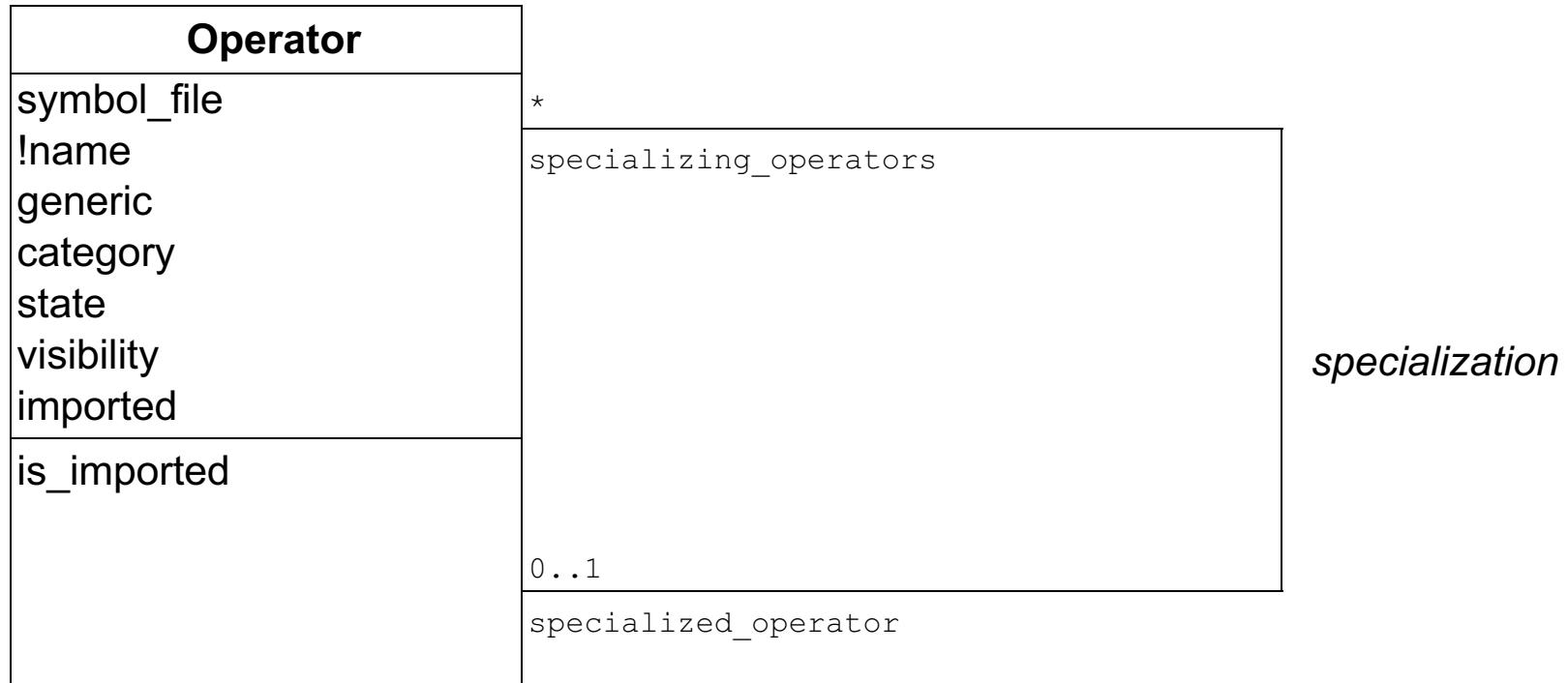
# State Machine (Python)



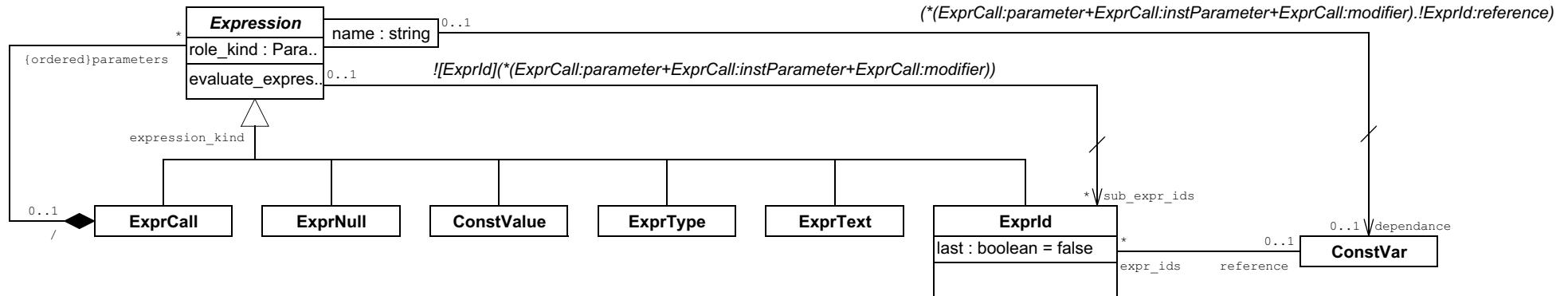
# ActivateBlock (Python)



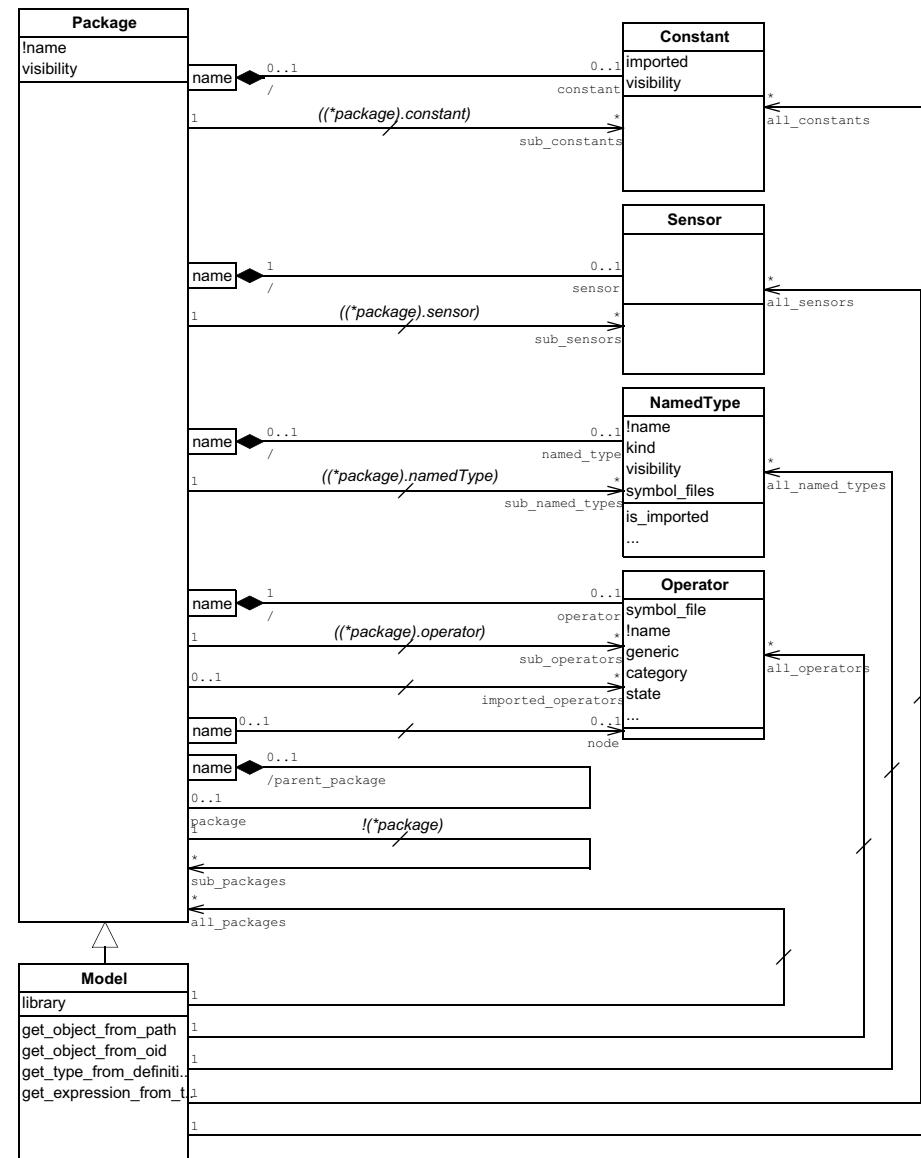
## Specialization (Python)



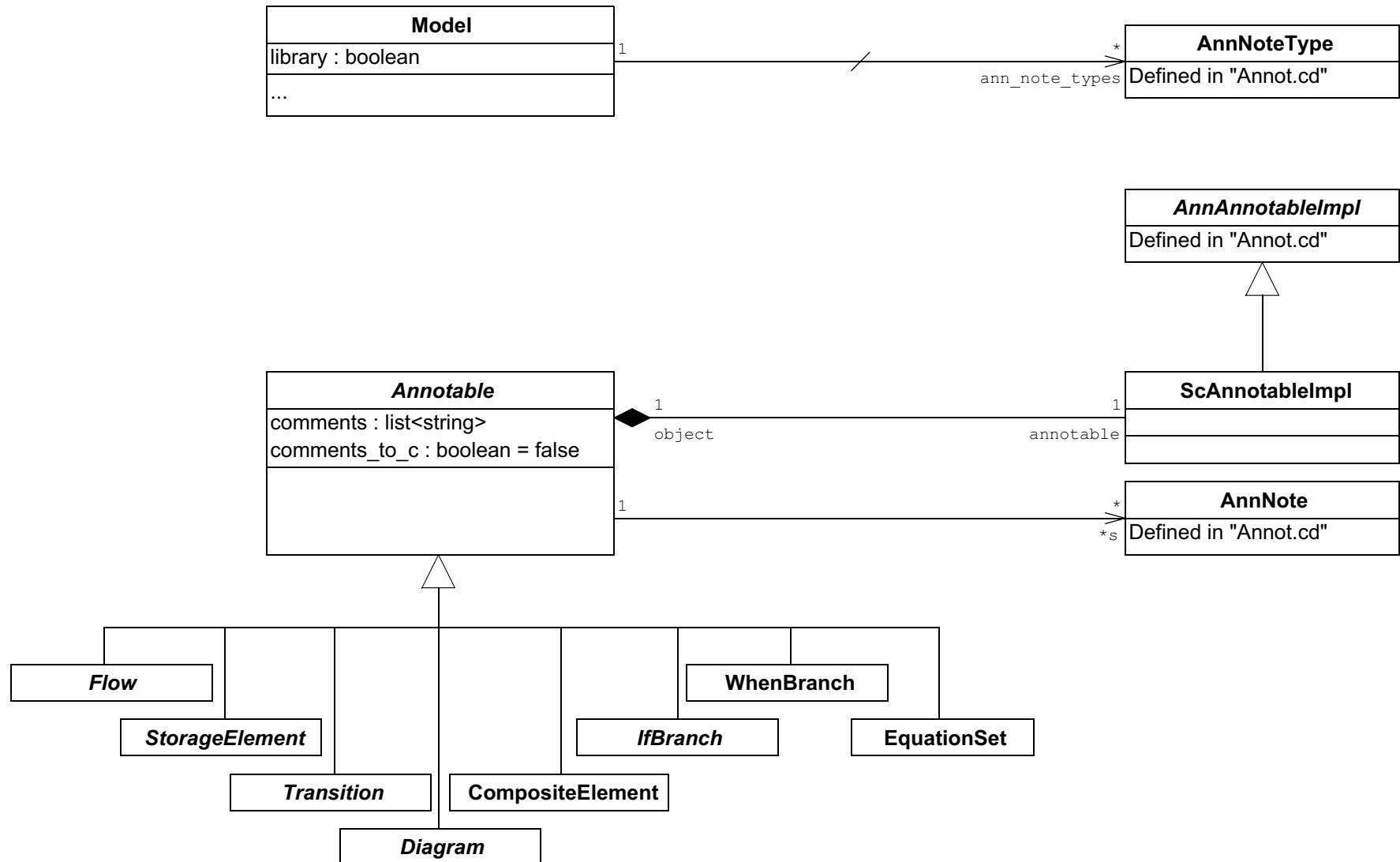
# Expression Map (Python)



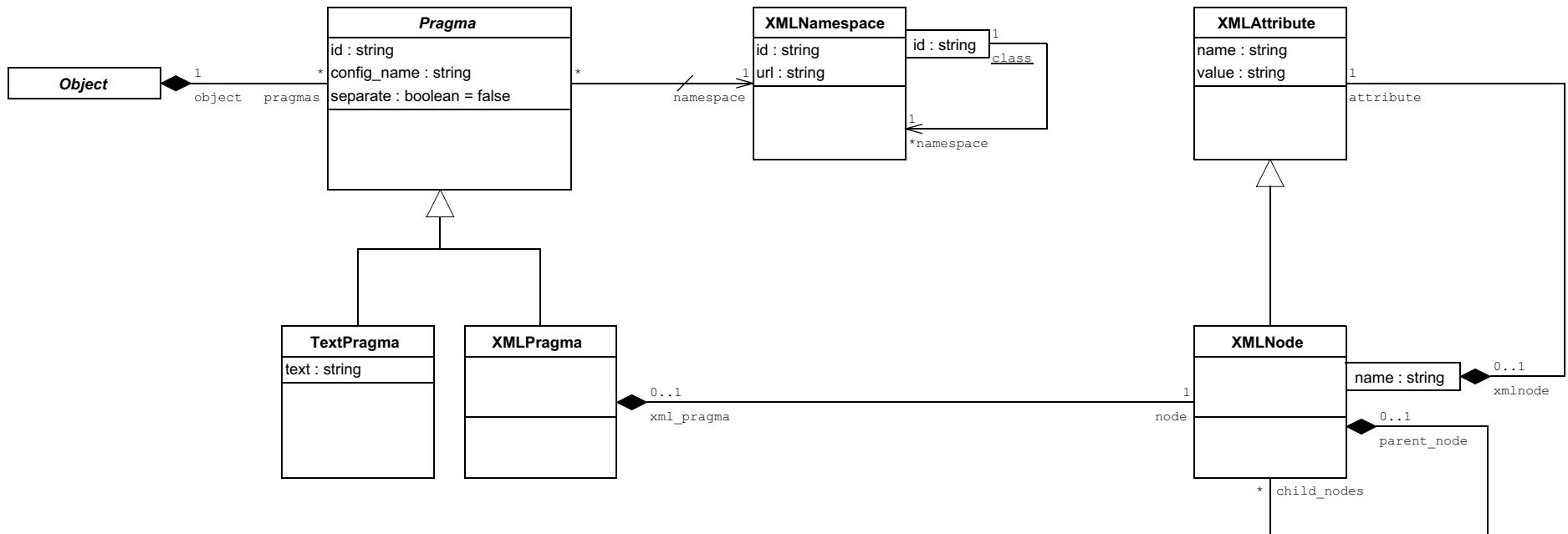
# Model and Package Maps (Python)



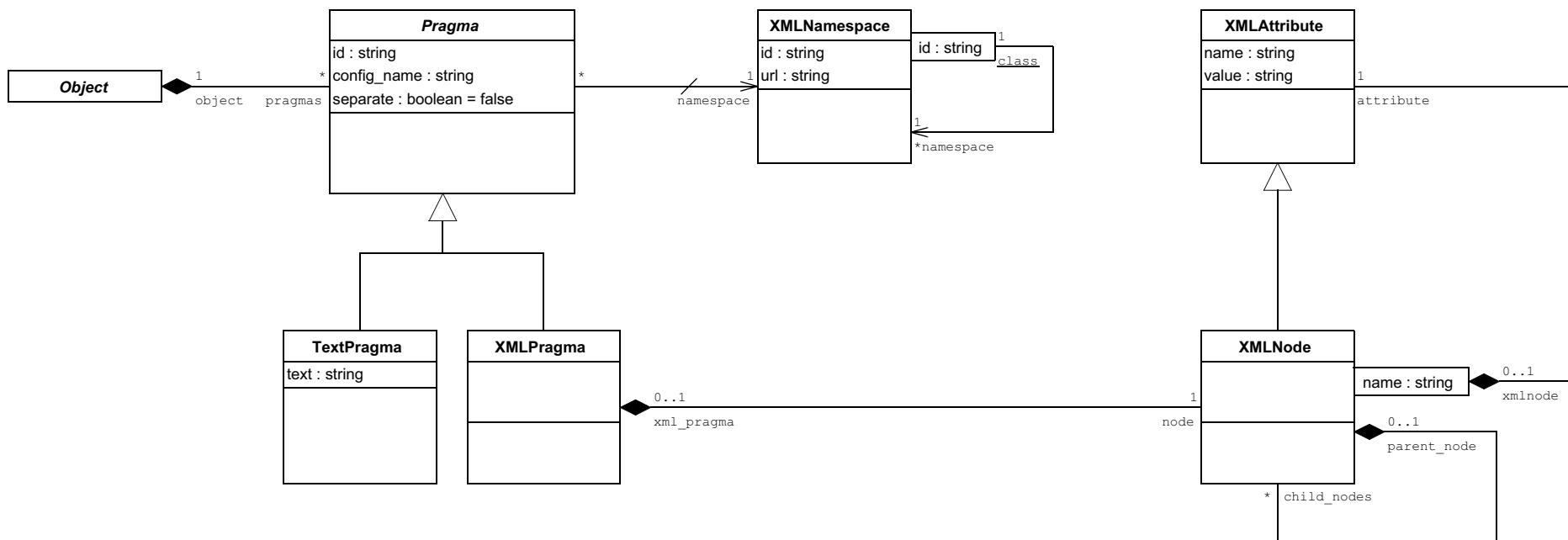
# Annotable (Python)



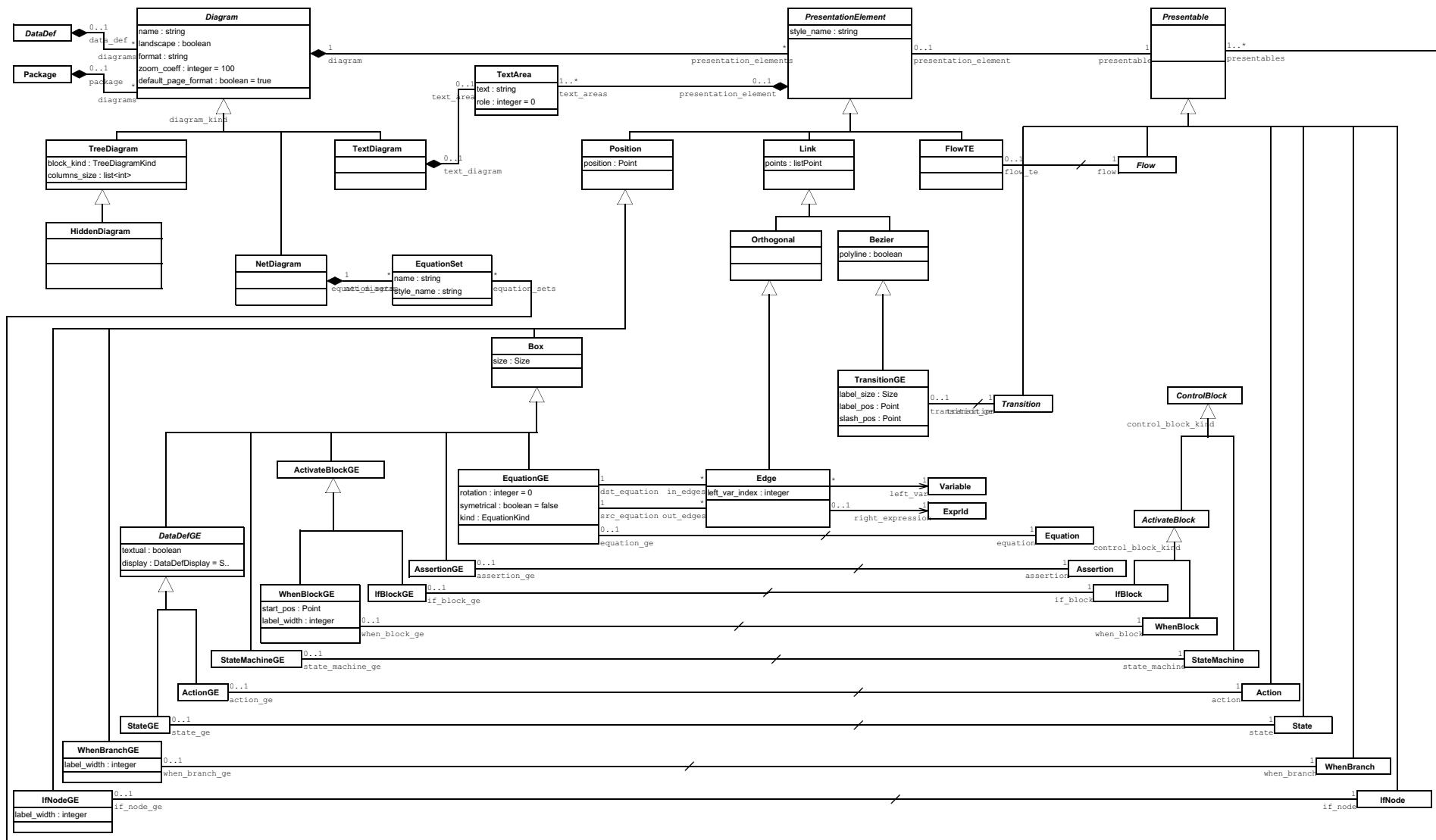
# Pragmas (Python)



# Scopes (Python)



# Graphical (Python)

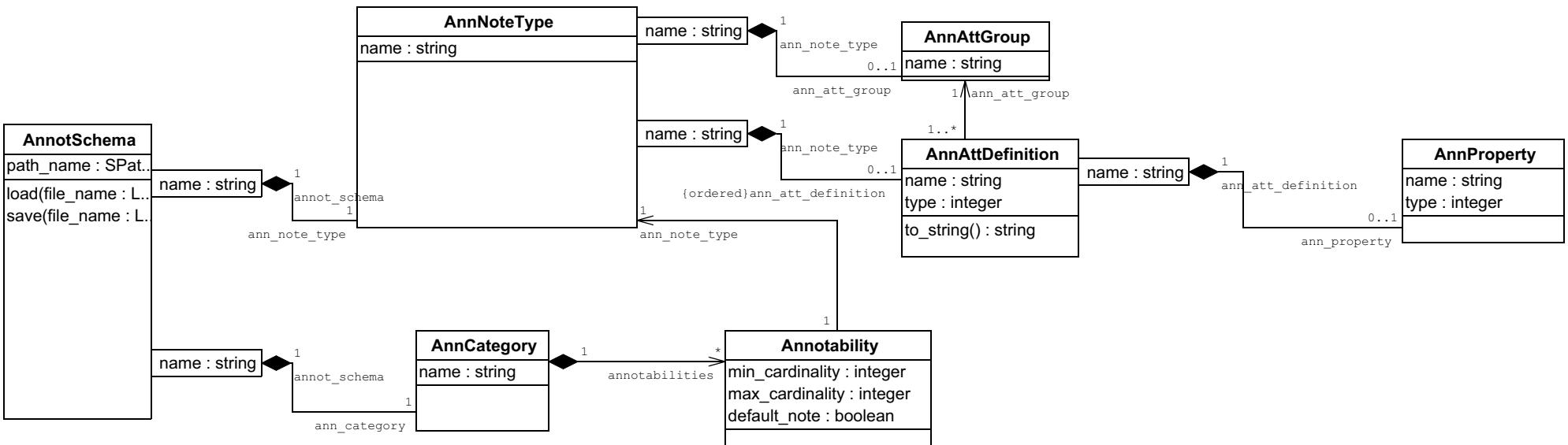


## 12 /Annotation Metamodels (Python API)

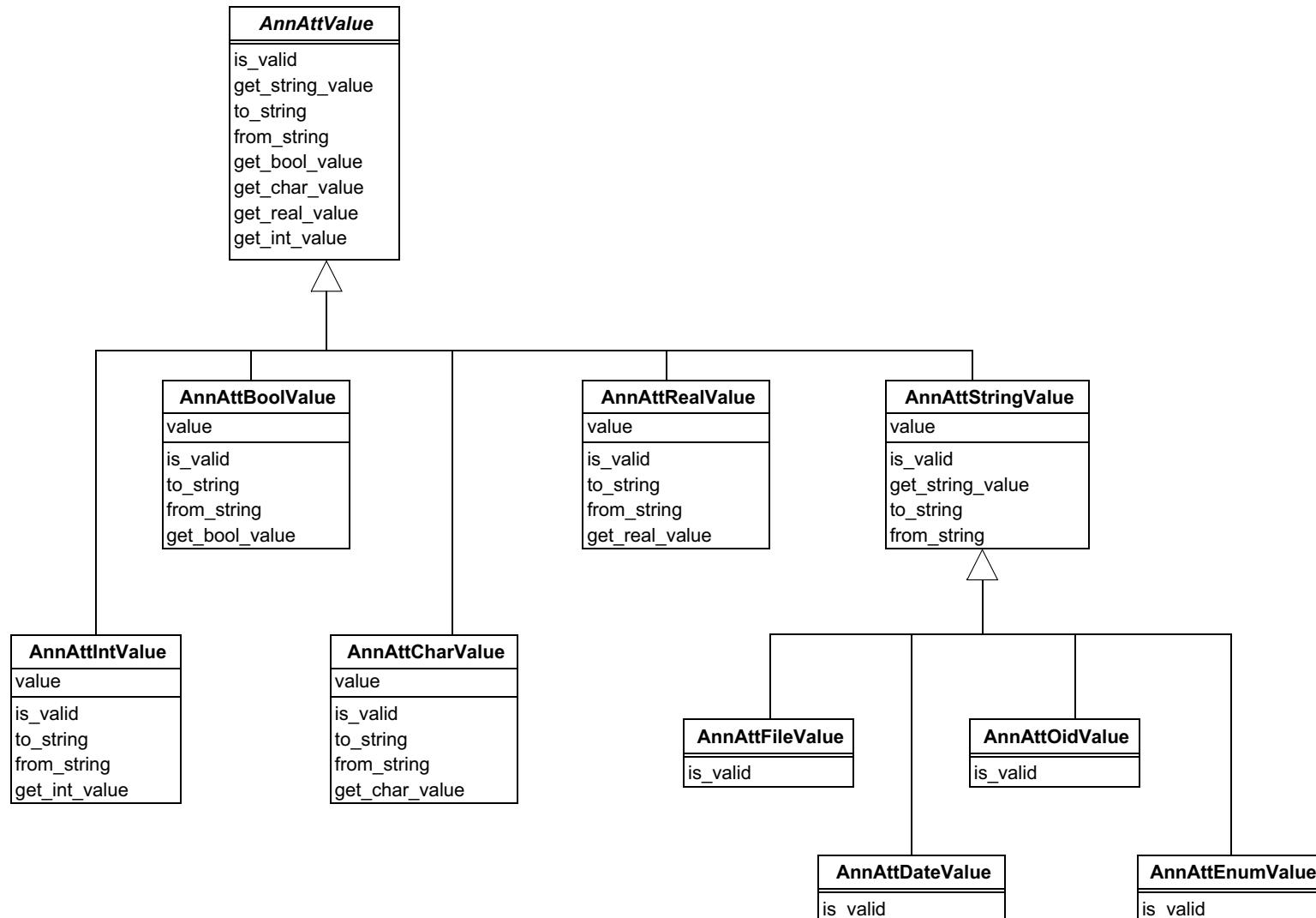
These metamodels present the data structures that give access to SCADE Suite annotations mechanisms using Python API:

- [Note Types \(Python\)](#)
- [Notes Attribute Values \(Python\)](#)
- [Notes Property Values \(Python\)](#)

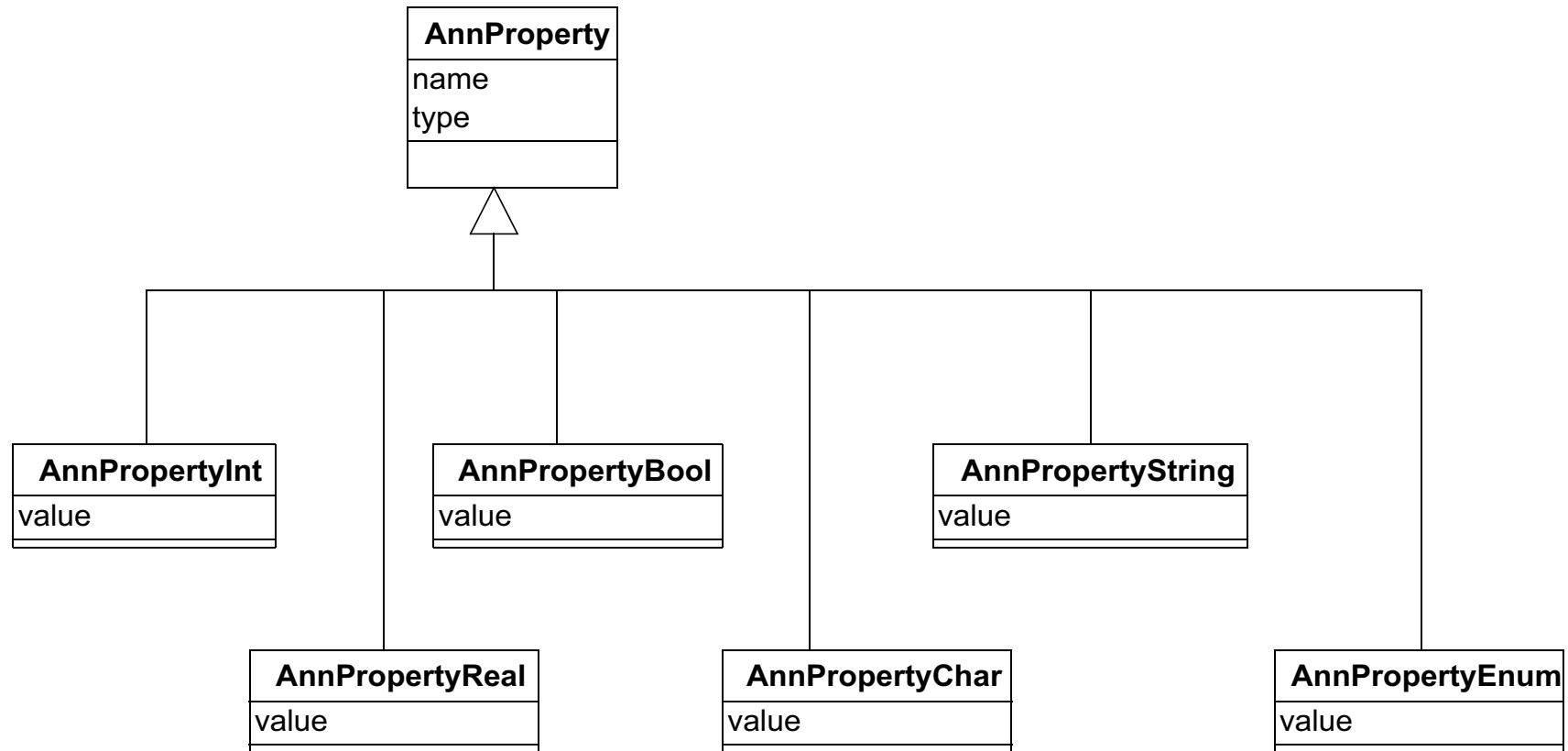
# Note Types (Python)



# Notes Attribute Values (Python)

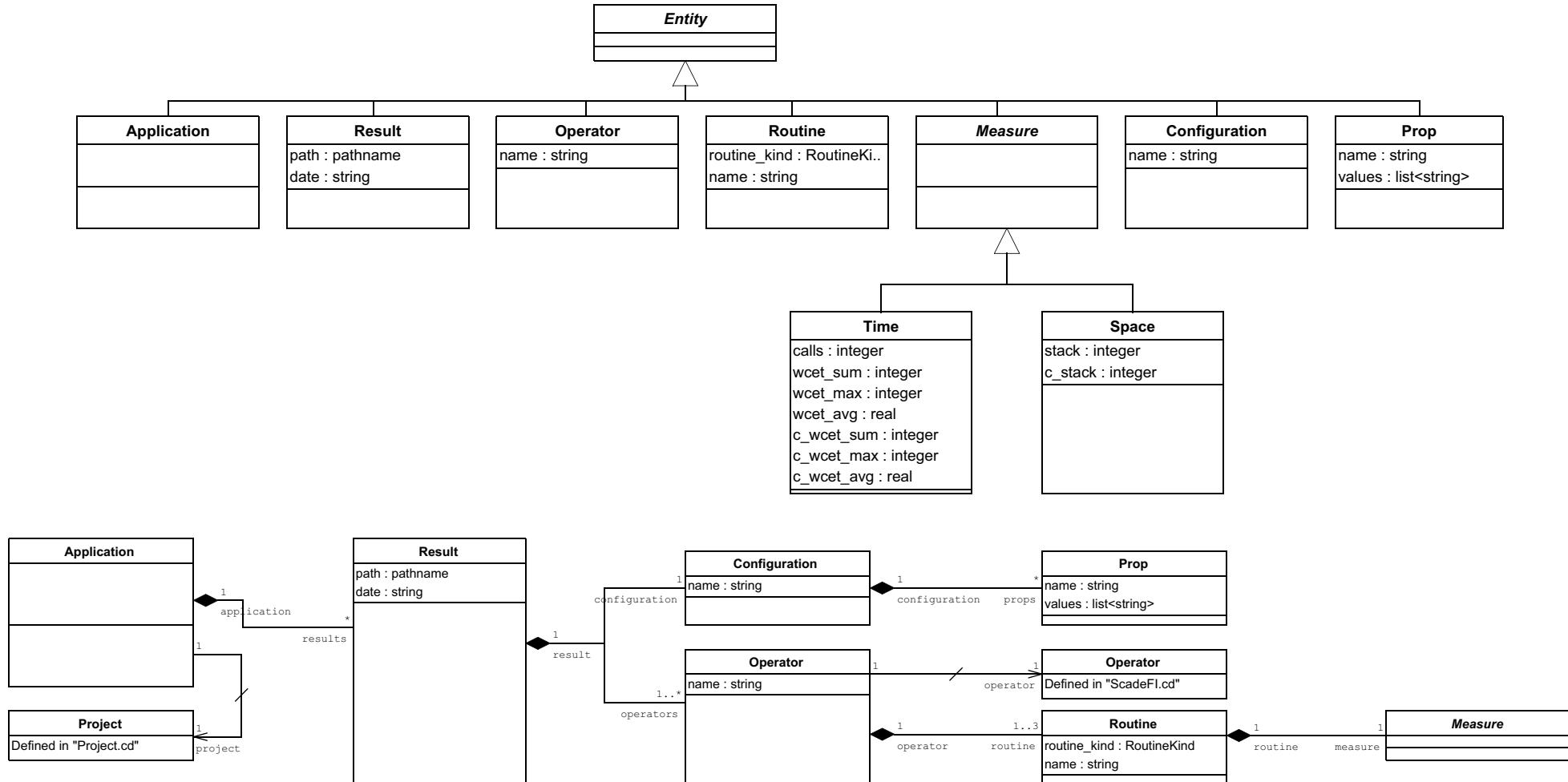


## Notes Property Values (Python)



# 13 /Timing and Stack Analysis Metamodel (Python API)

This metamodel presents the data structures that give access to SCADE Suite timing and stack analysis results Python API:

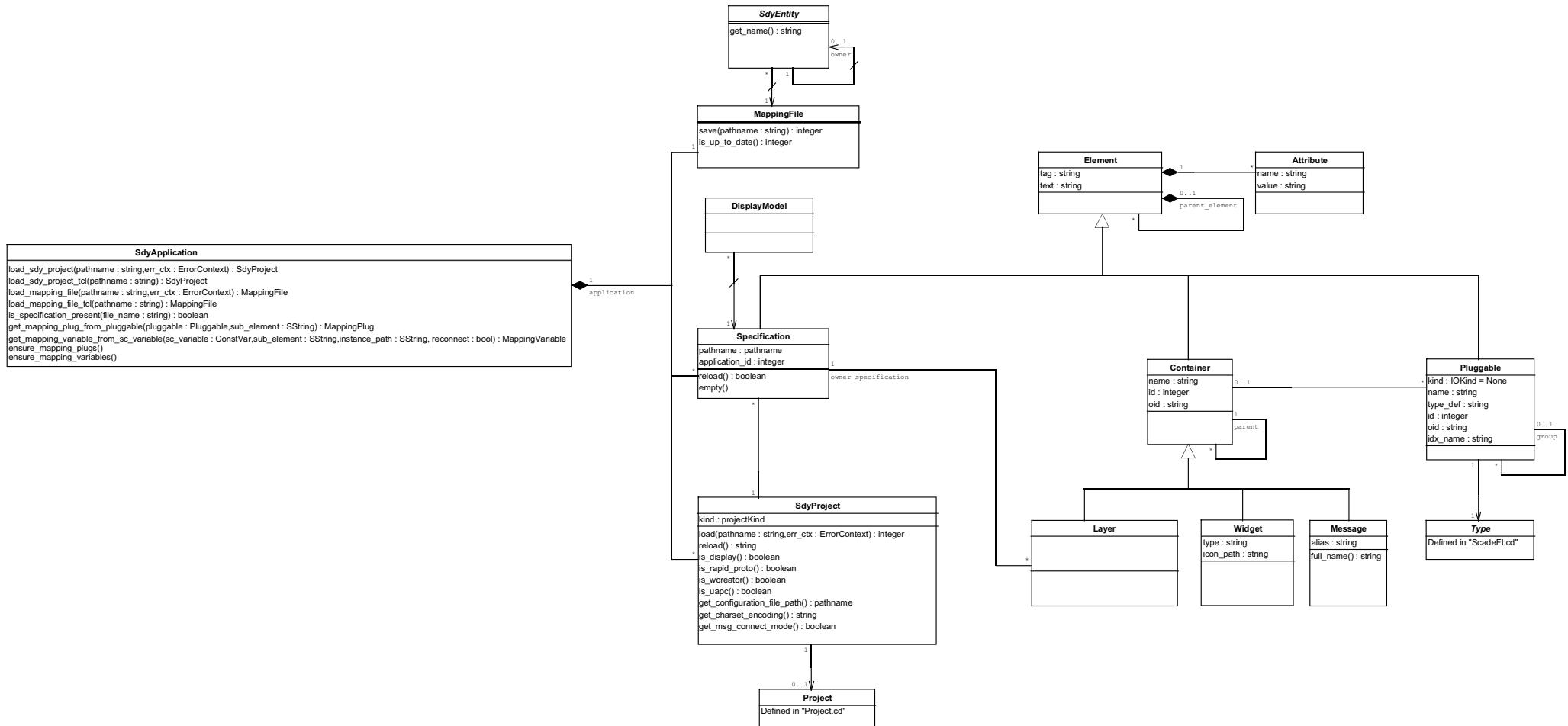


## 14 /Graphical Panel Coupling Metamodel (Python API)

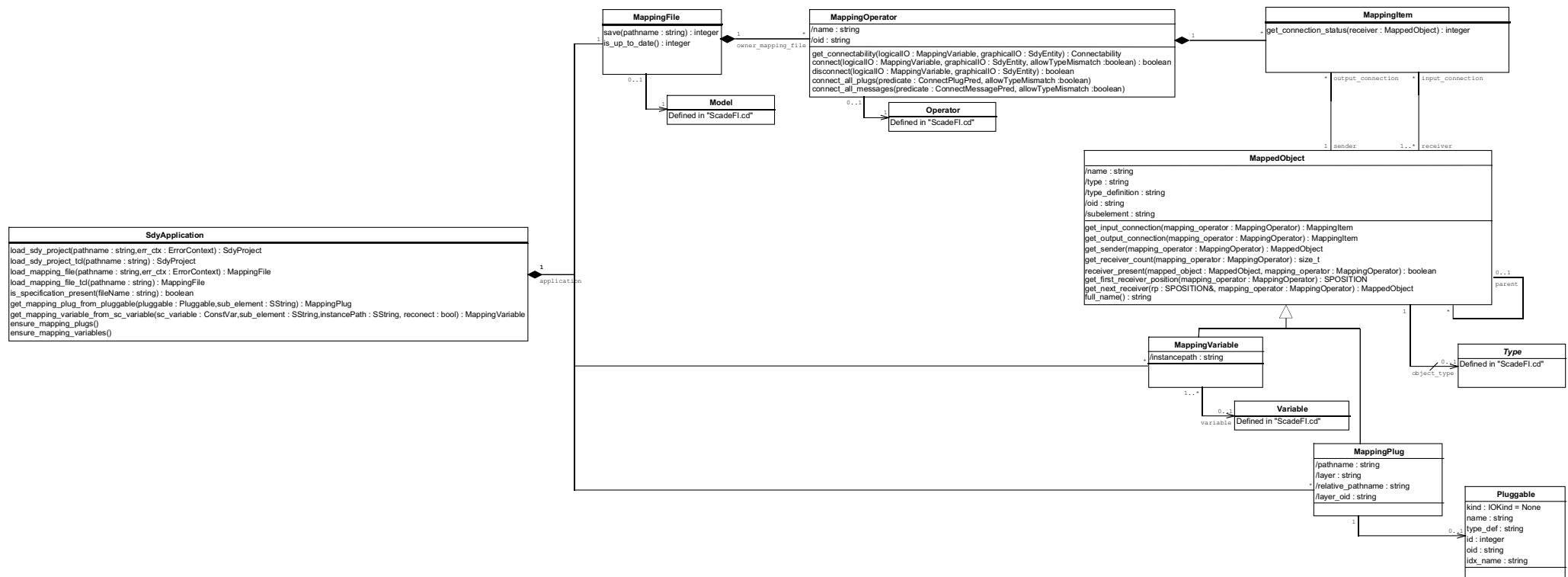
These metamodels present the data structures that give access to connection data using SCADE Display Coupling Python API:

- ["Graphical Panel Coupling \(main\)"](#)
- ["Graphical Panel Coupling \(mapping file\)"](#)
- ["Graphical Panel Coupling \(mapping plug for ARINC 661\)"](#)
- ["Graphical Panel Coupling \(inheritance\)"](#)

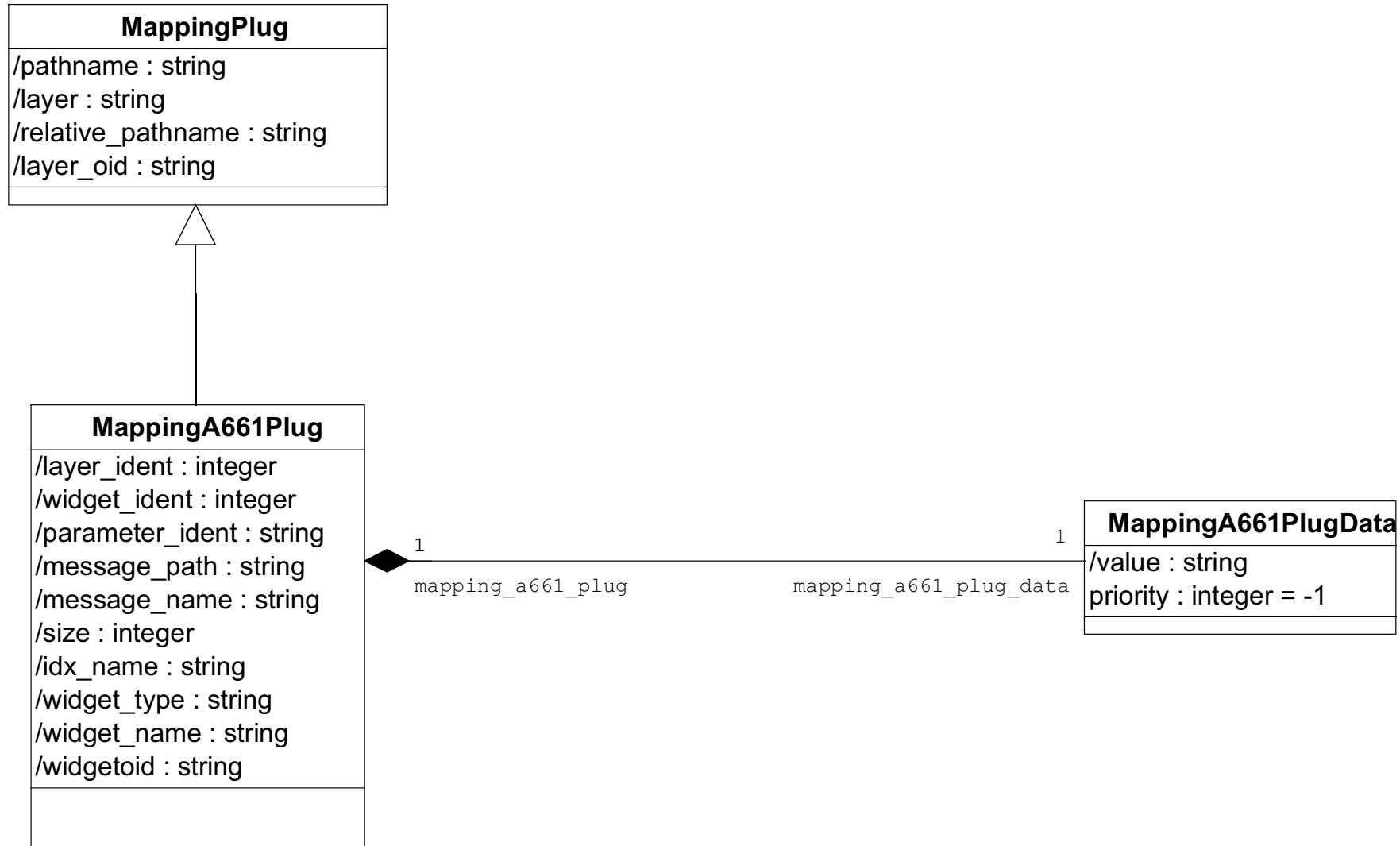
# Graphical Panel Coupling (main)



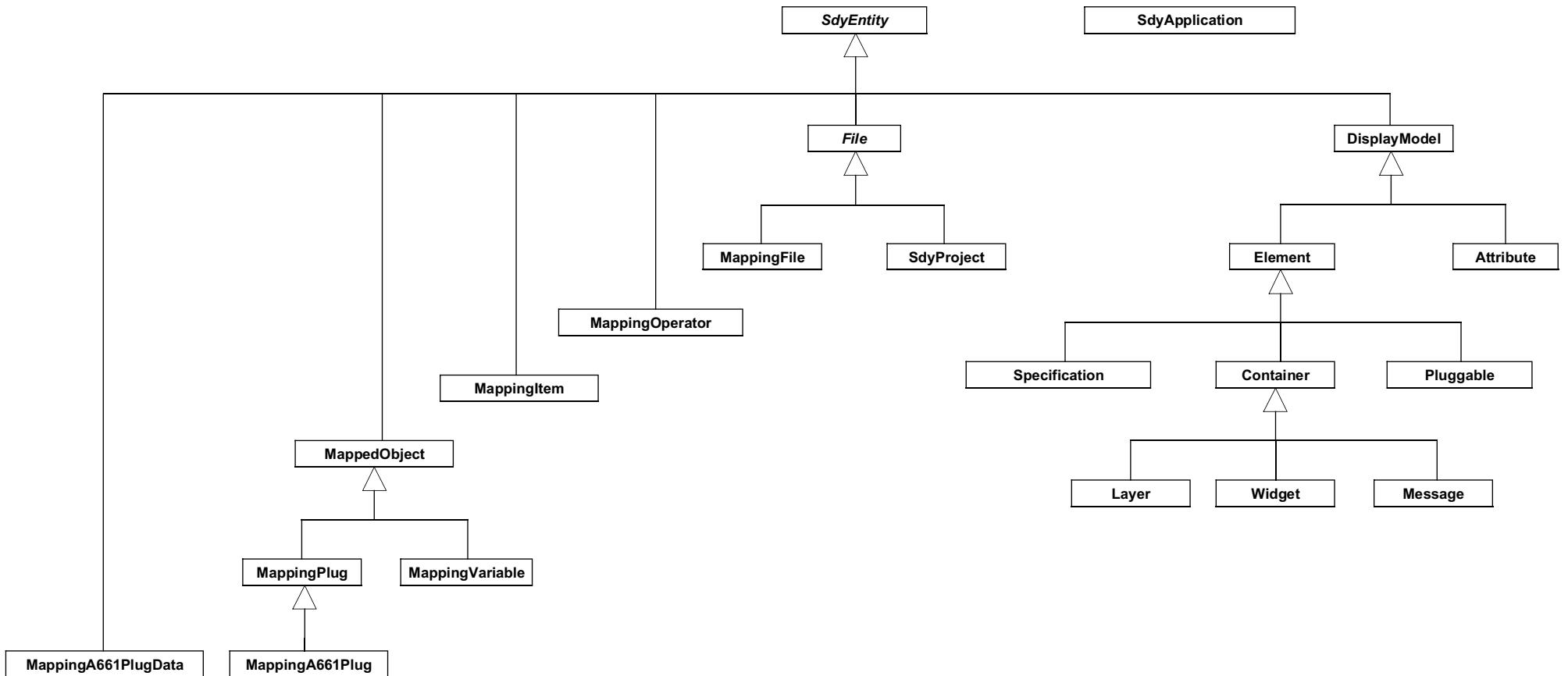
# Graphical Panel Coupling (mapping file)



# Graphical Panel Coupling (mapping plug for ARINC 661)



# Graphical Panel Coupling (inheritance)



## Part 3

# SCADE Suite Metamodels for Tcl API

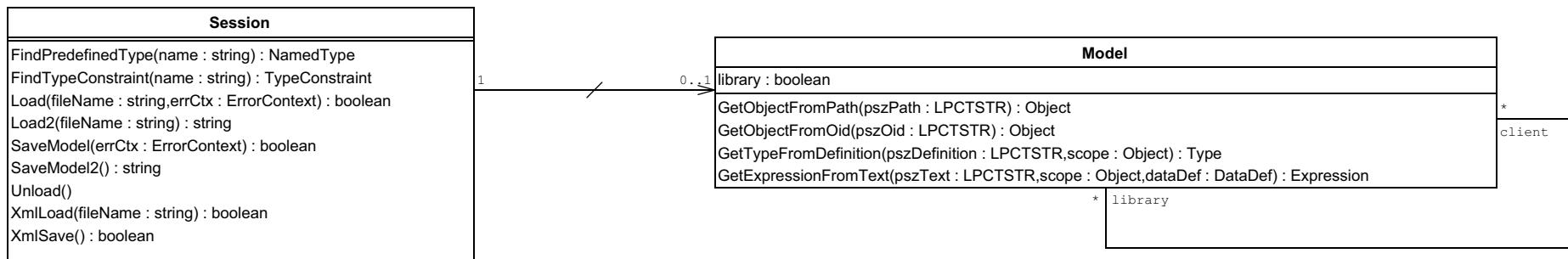
- 15/ ["Editor Metamodels \(Tcl API\)"](#)
- 16/ ["Annotation Metamodels \(Tcl API\)"](#)
- 17/ ["Timing and Stack Analysis Metamodel \(Tcl API\)"](#)

# 15 /Editor Metamodels (Tcl API)

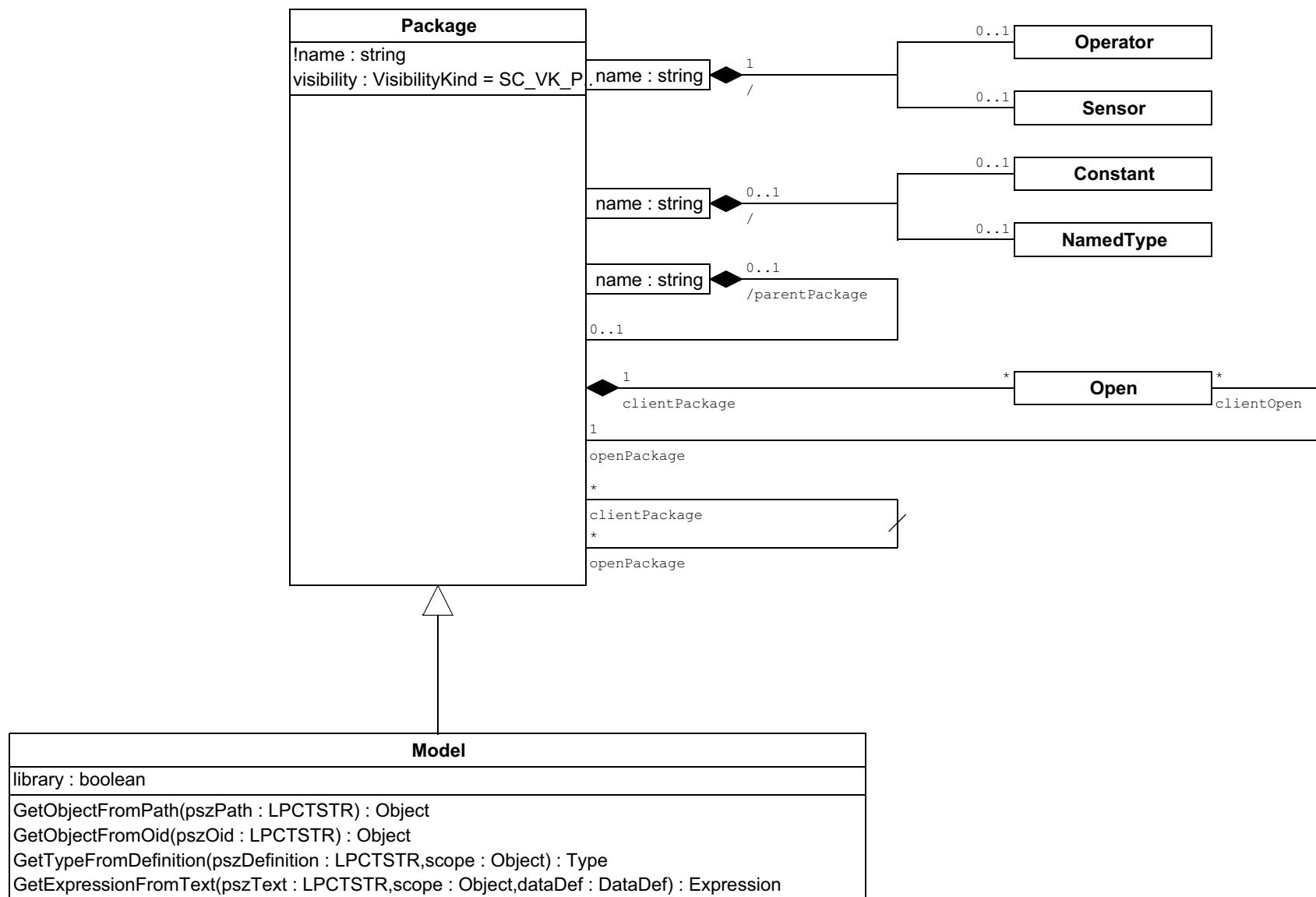
These metamodels present the data structures that give access to SCADE Suite Editor environment using Tcl API:

- ["Session \(Tcl\)"](#)
- ["Model \(Tcl\)"](#)
- ["Variable \(Tcl\)"](#)
- ["Operator \(Tcl\)"](#)
- ["Expression \(Tcl\)"](#)
- ["Type \(Tcl\)"](#)
- ["Data Definition \(Tcl\)"](#)
- ["Data Flow \(Tcl\)"](#)
- ["Model Storage - IO \(Tcl\)"](#)
- ["Composition \(Tcl\)"](#)
- ["Inheritance \(Tcl\)"](#)
- ["State Machine \(Tcl\)"](#)
- ["ActivateBlock \(Tcl\)"](#)
- ["Specialization \(Tcl\)"](#)
- ["Expression Maps \(Tcl\)"](#)
- ["Model and Package Maps \(Tcl\)"](#)
- ["Annotable \(Tcl\)"](#)
- ["Pragmas \(Tcl\)"](#)
- ["Scopes \(Tcl\)"](#)
- ["Graphical \(Tcl\)"](#)
- ["Traceability \(Tcl\)"](#)

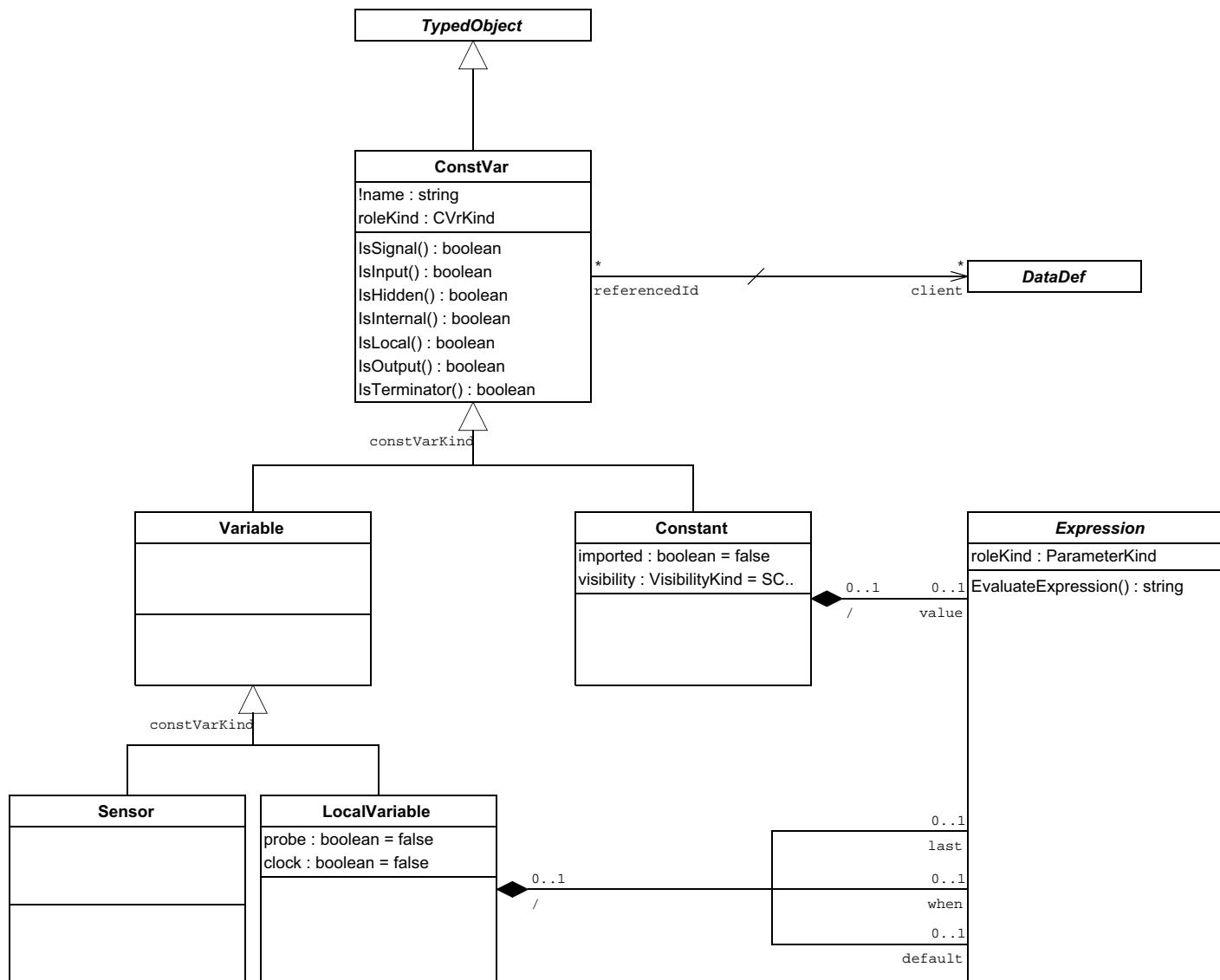
## Session (Tcl)



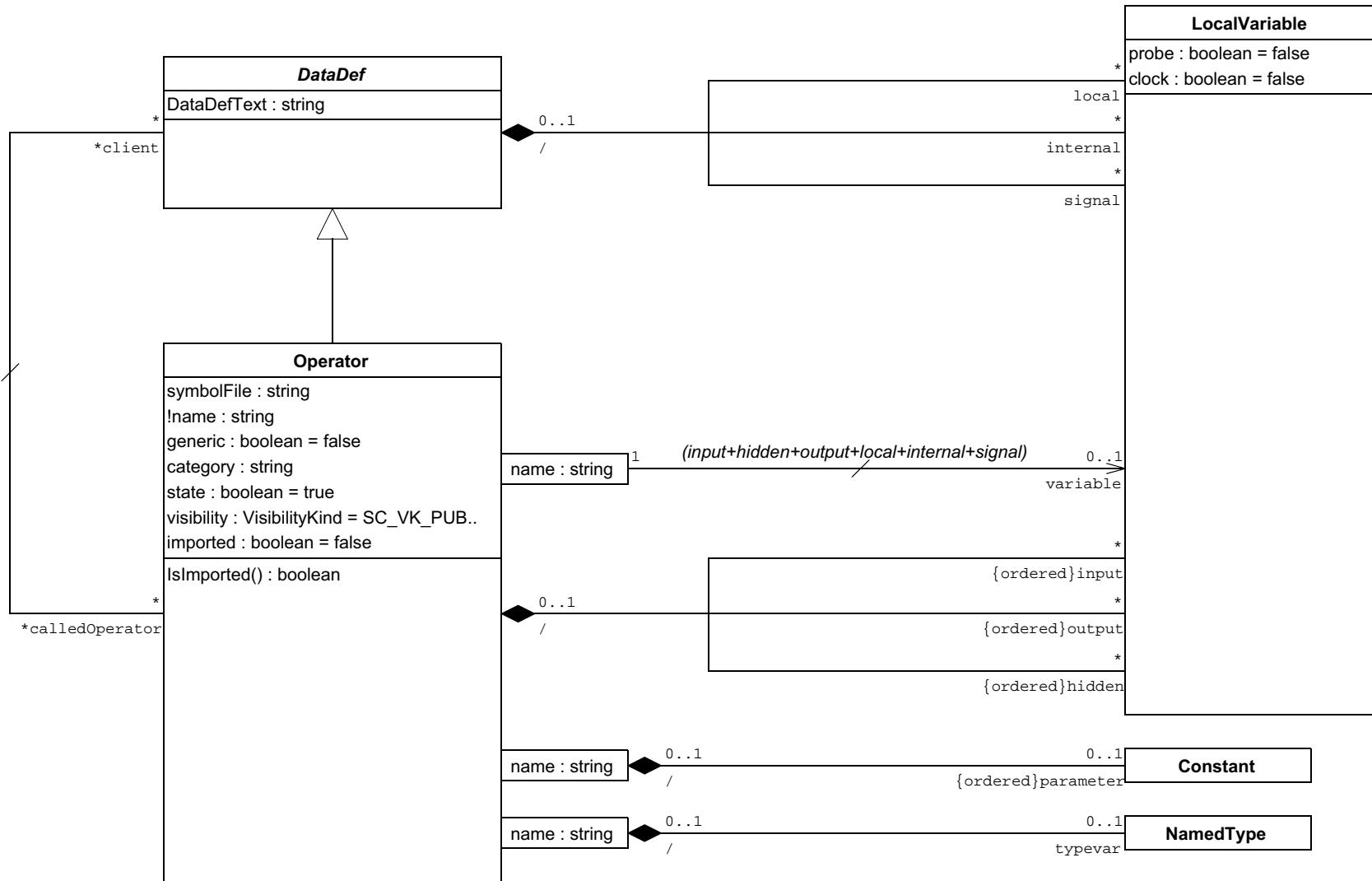
# Model (Tcl)



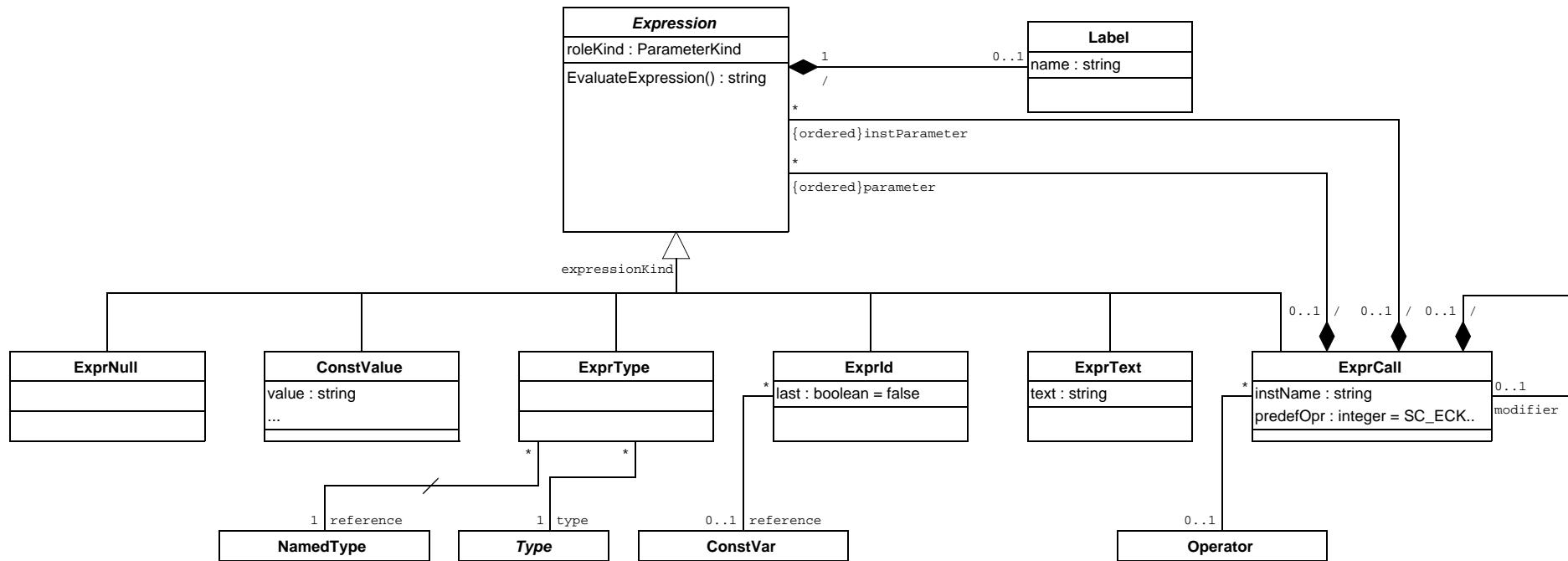
# Variable (Tcl)



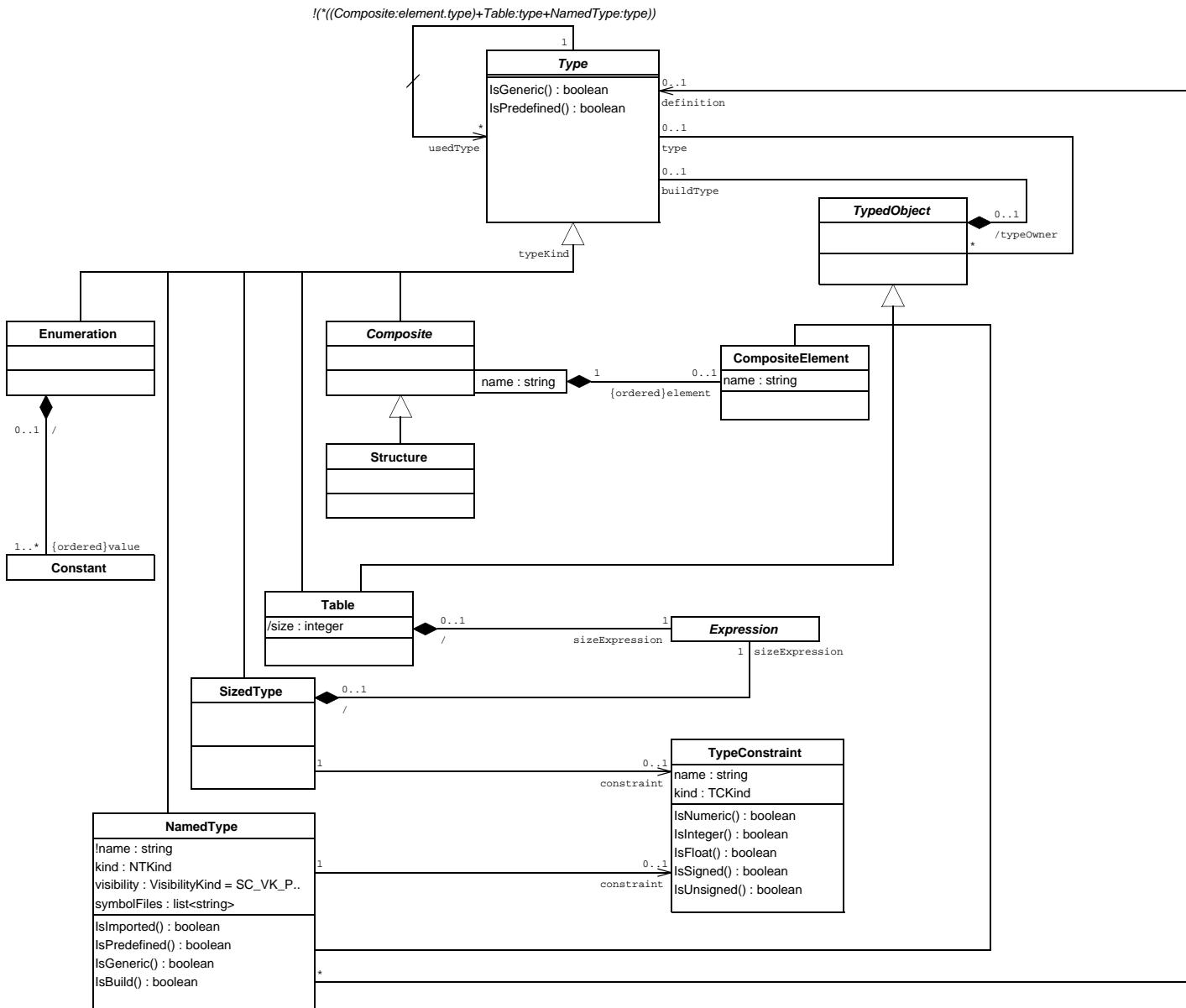
# Operator (Tcl)



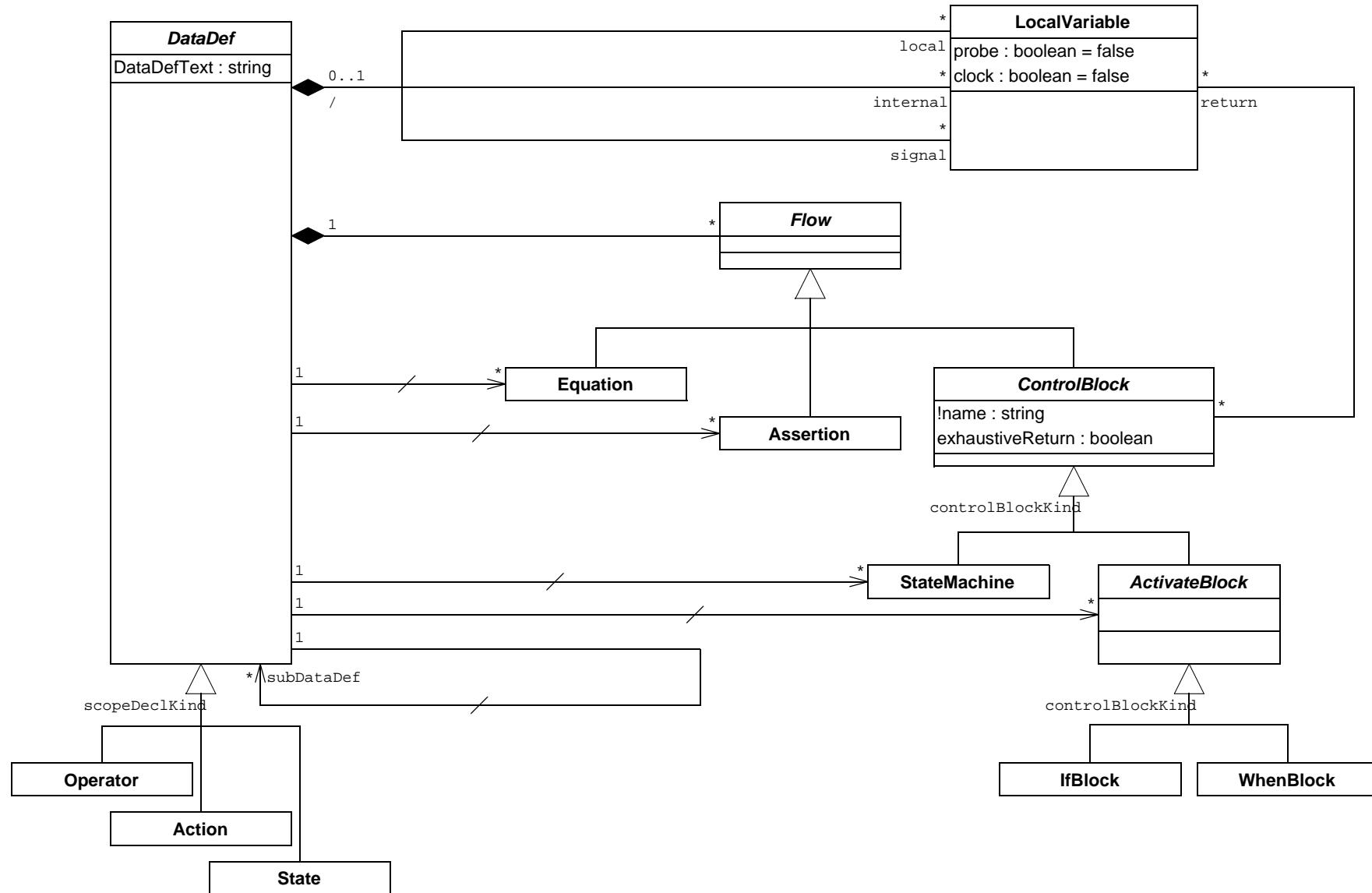
# Expression (Tcl)



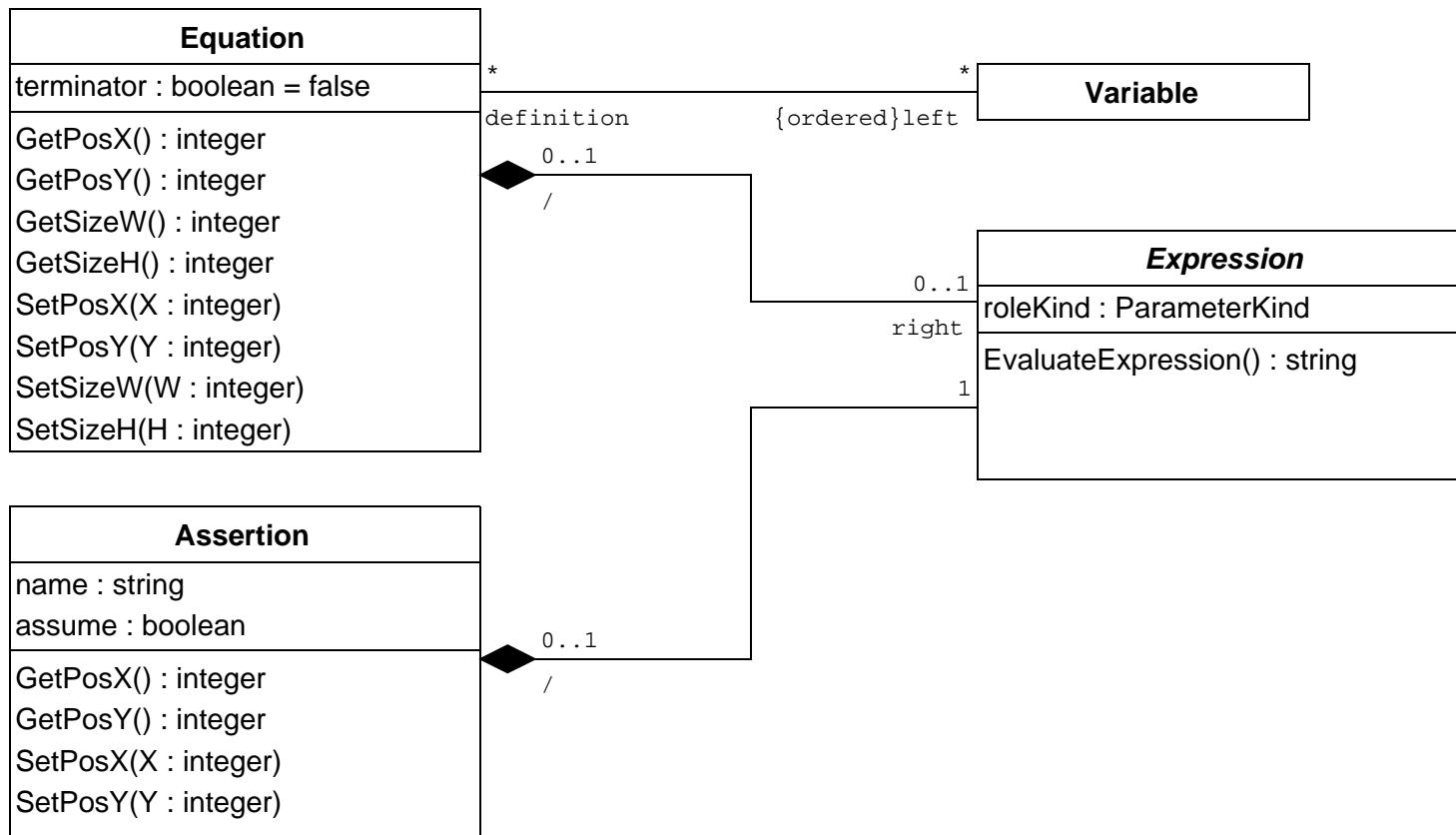
# Type (Tcl)



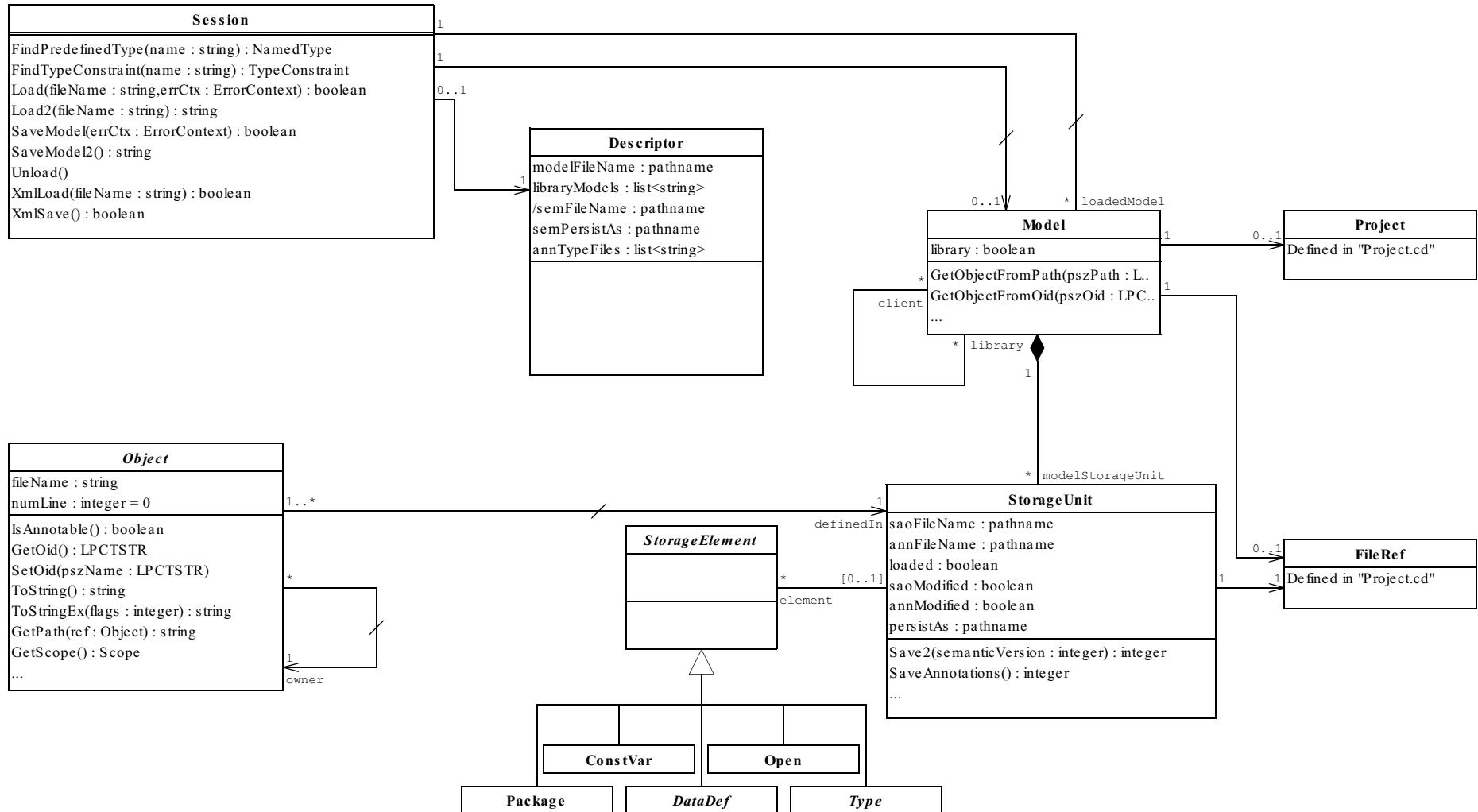
# Data Definition (Tcl)



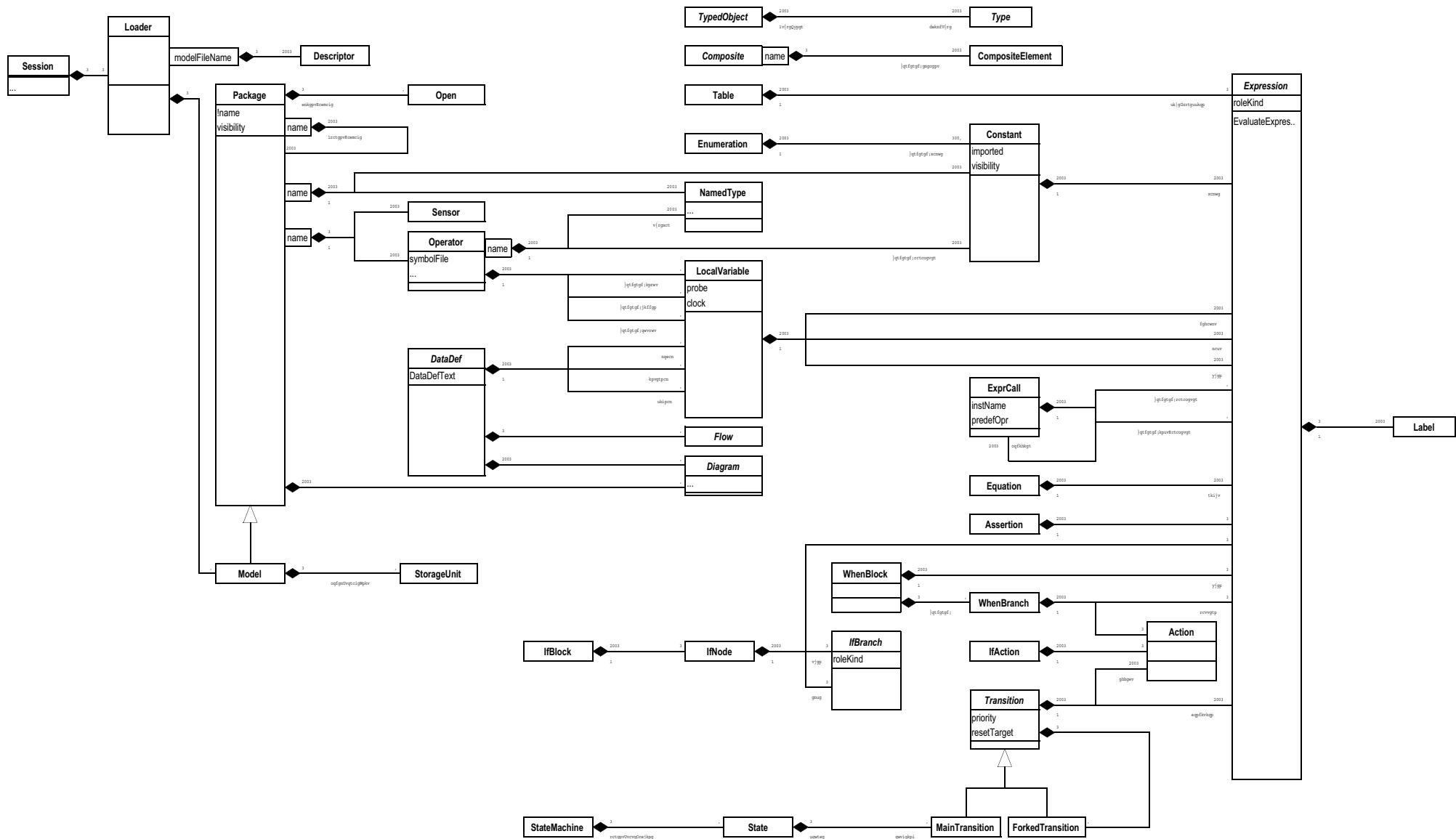
## Data Flow (Tcl)



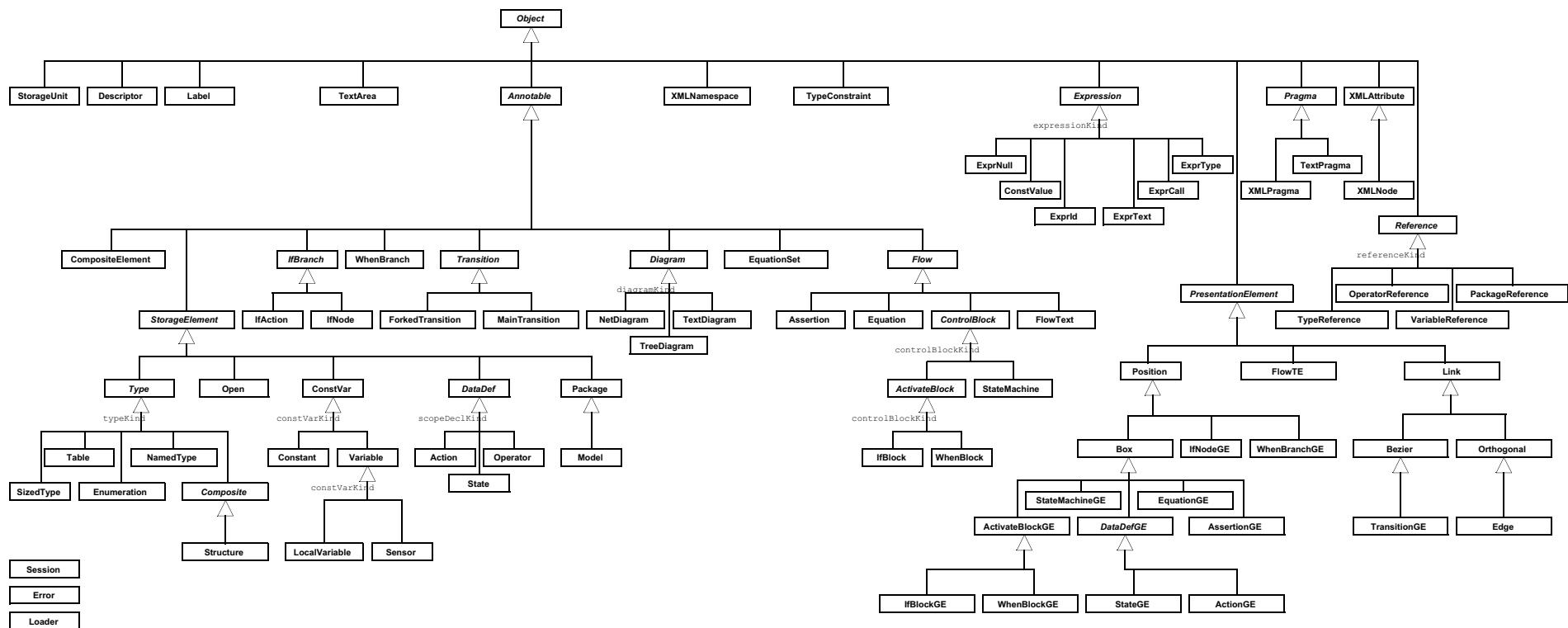
# Model Storage - IO (Tcl)



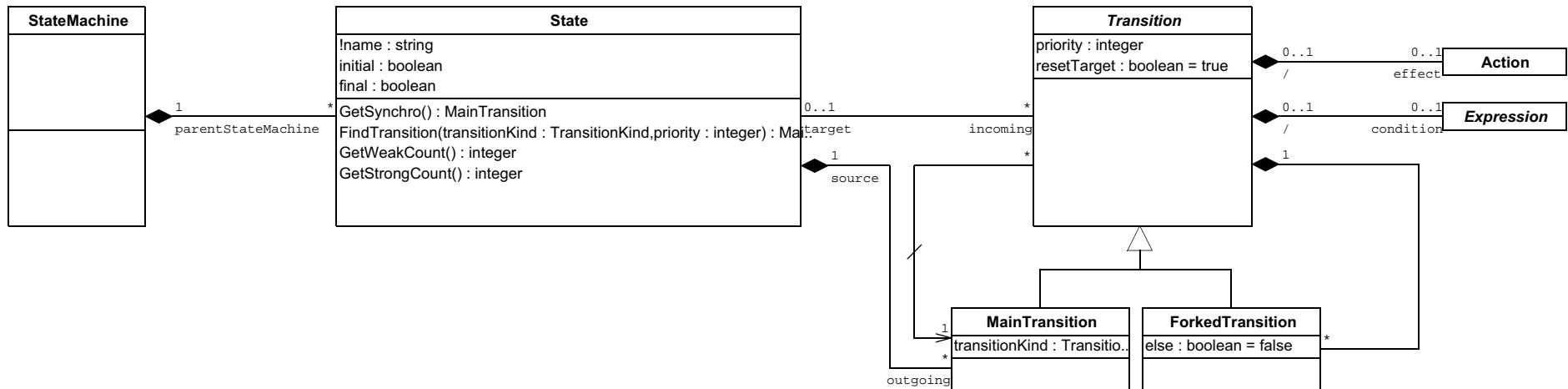
# Composition (Tcl)



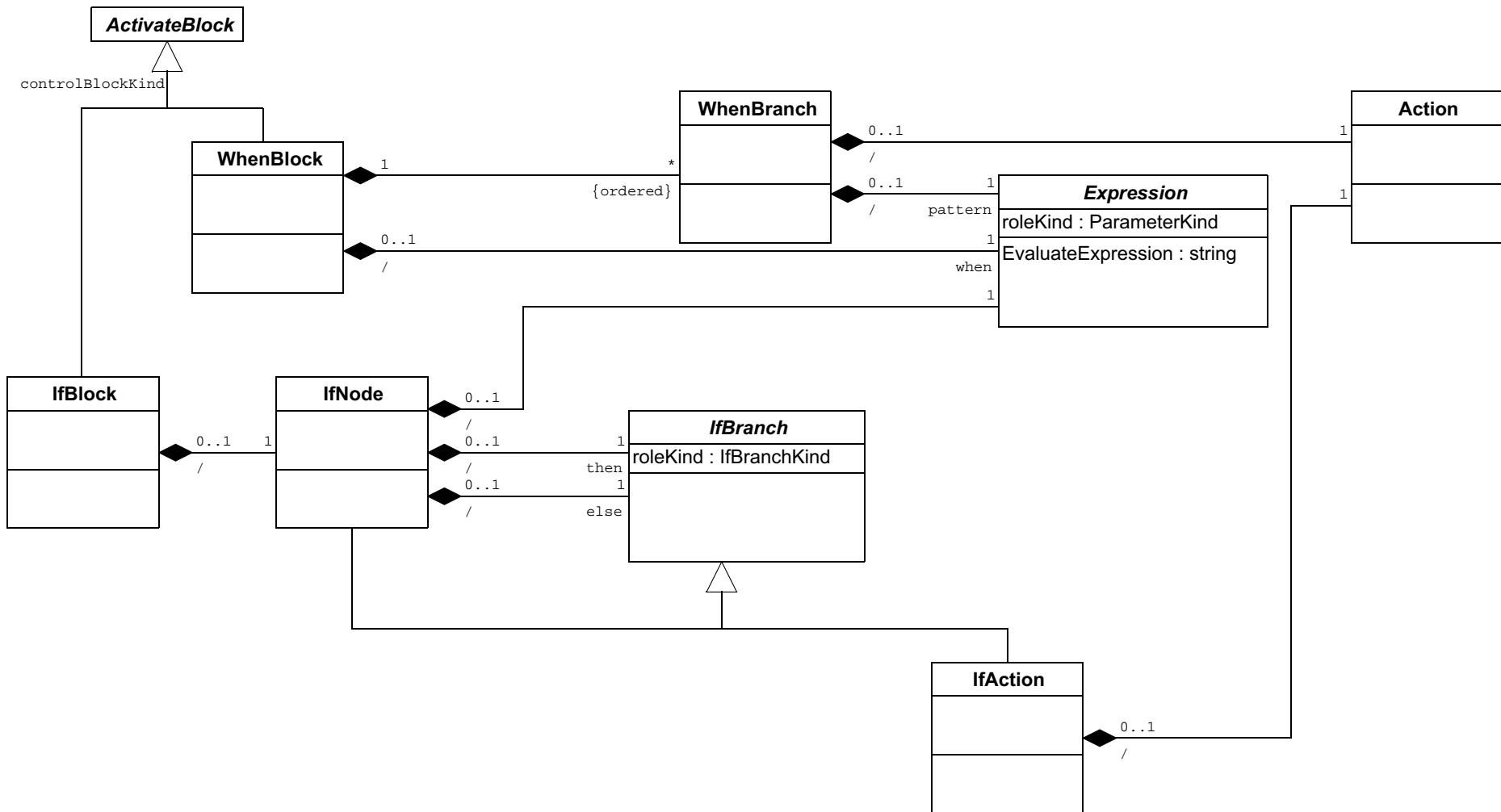
# Inheritance (Tcl)



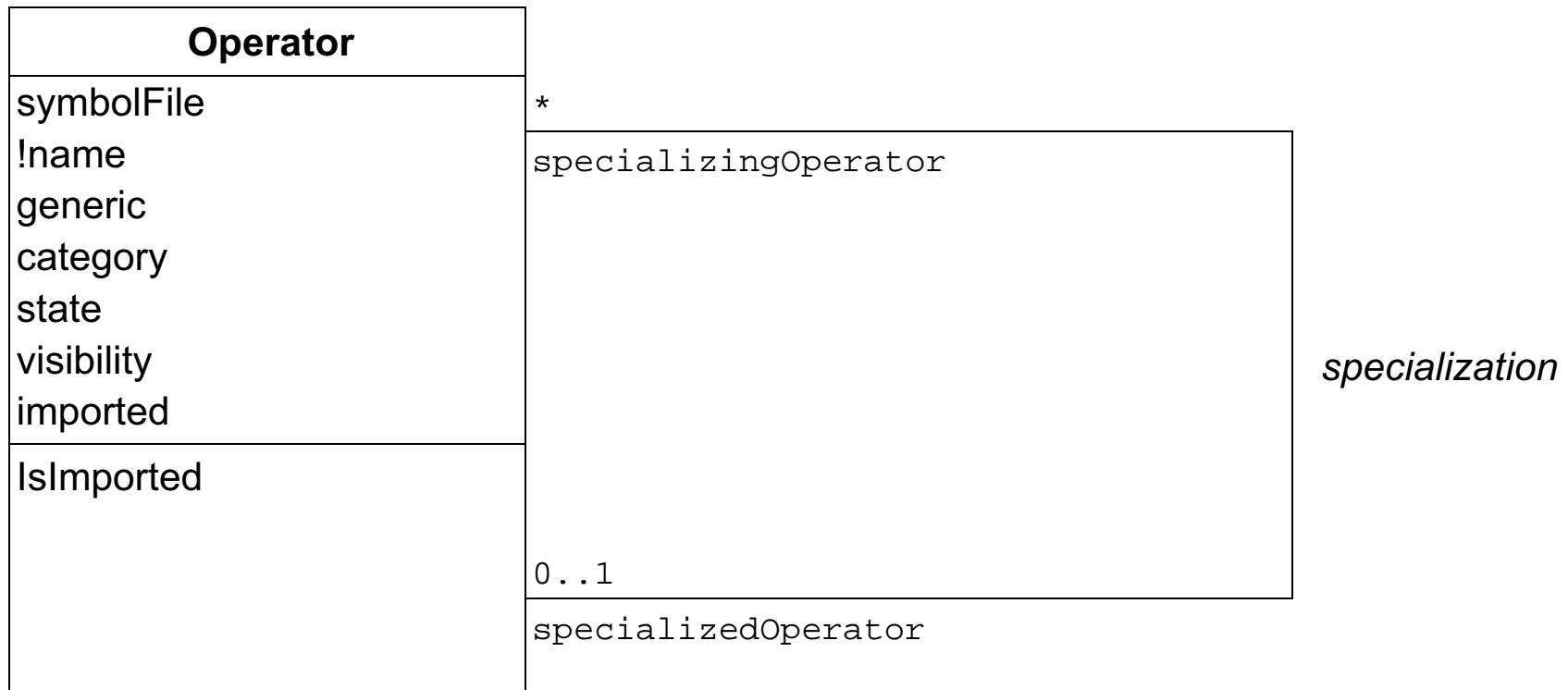
# State Machine (Tcl)



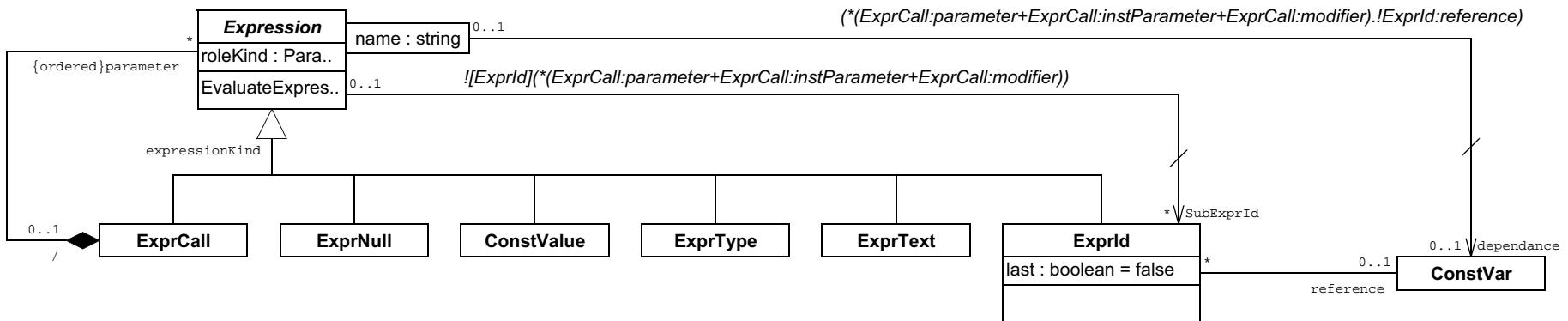
## ActivateBlock (Tcl)



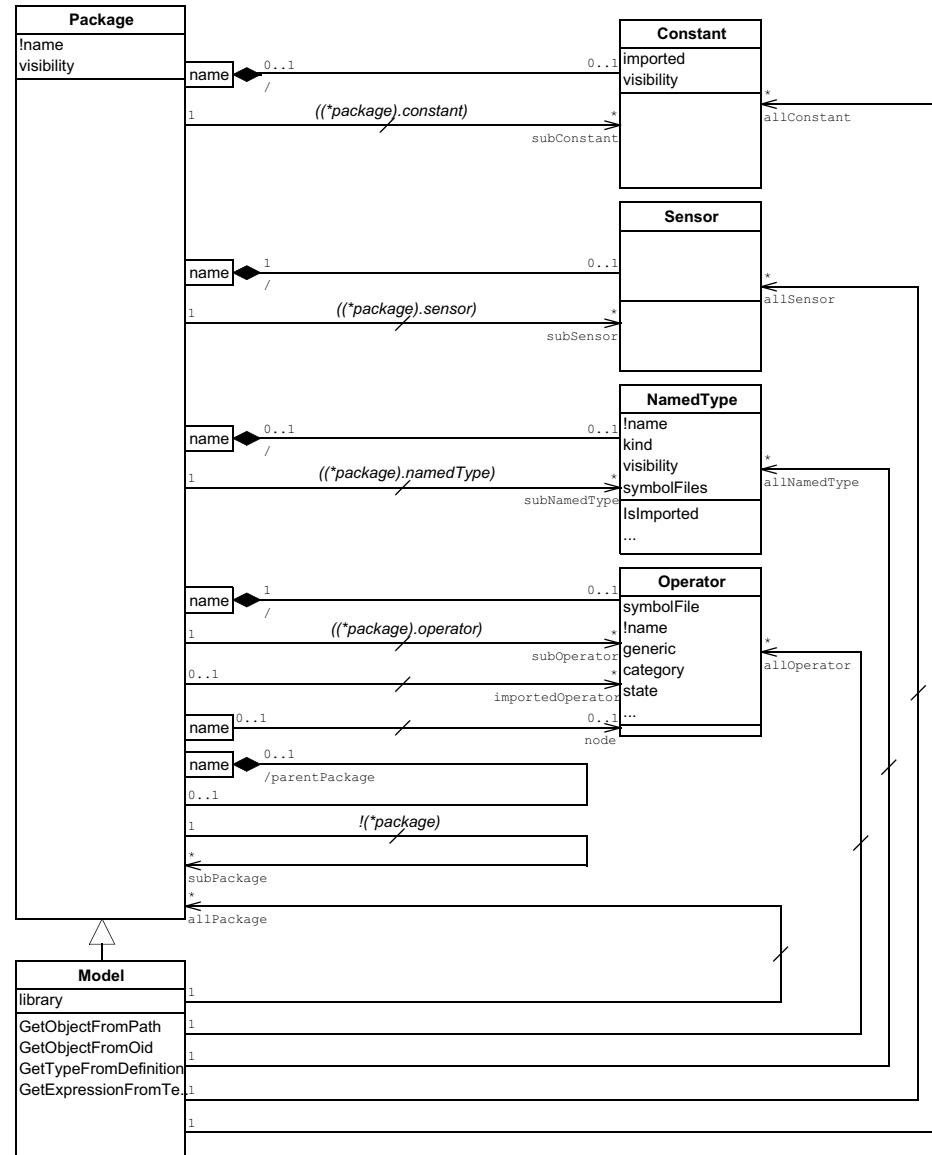
## Specialization (Tcl)



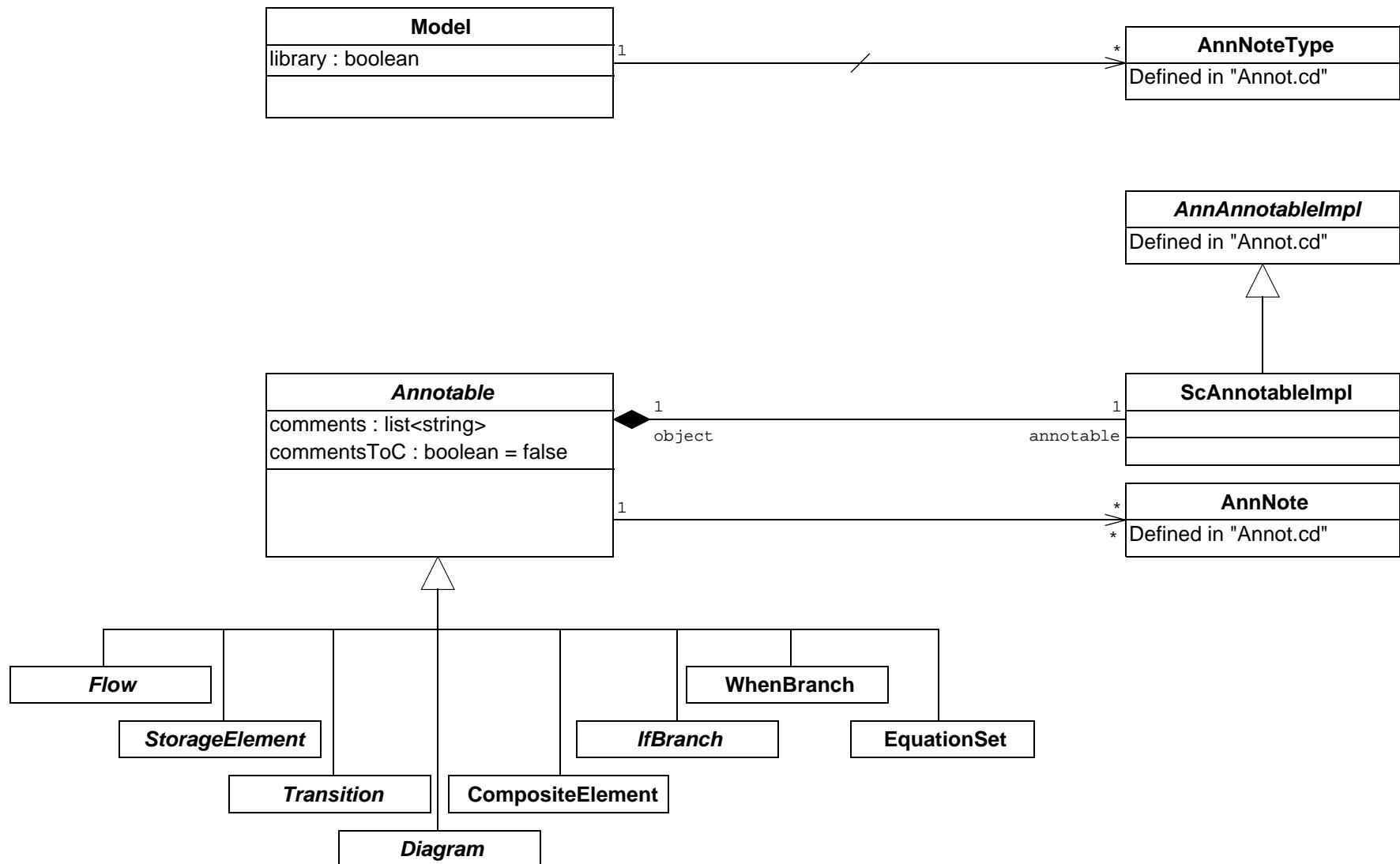
# Expression Maps (Tcl)



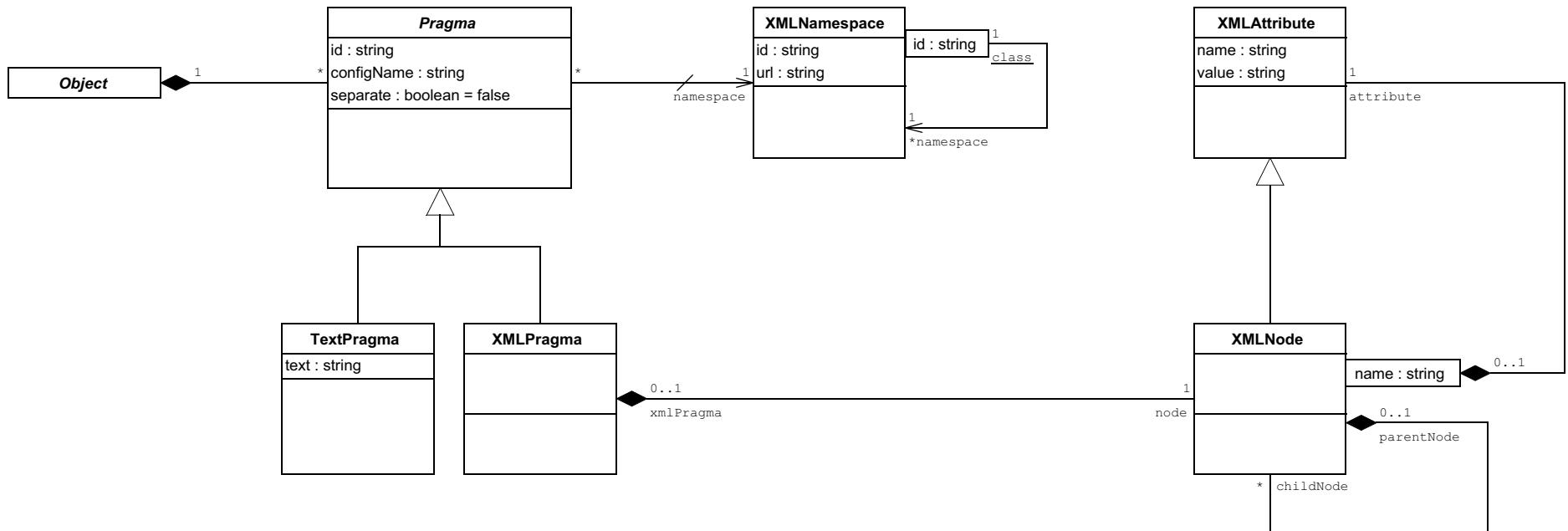
# Model and Package Maps (Tcl)



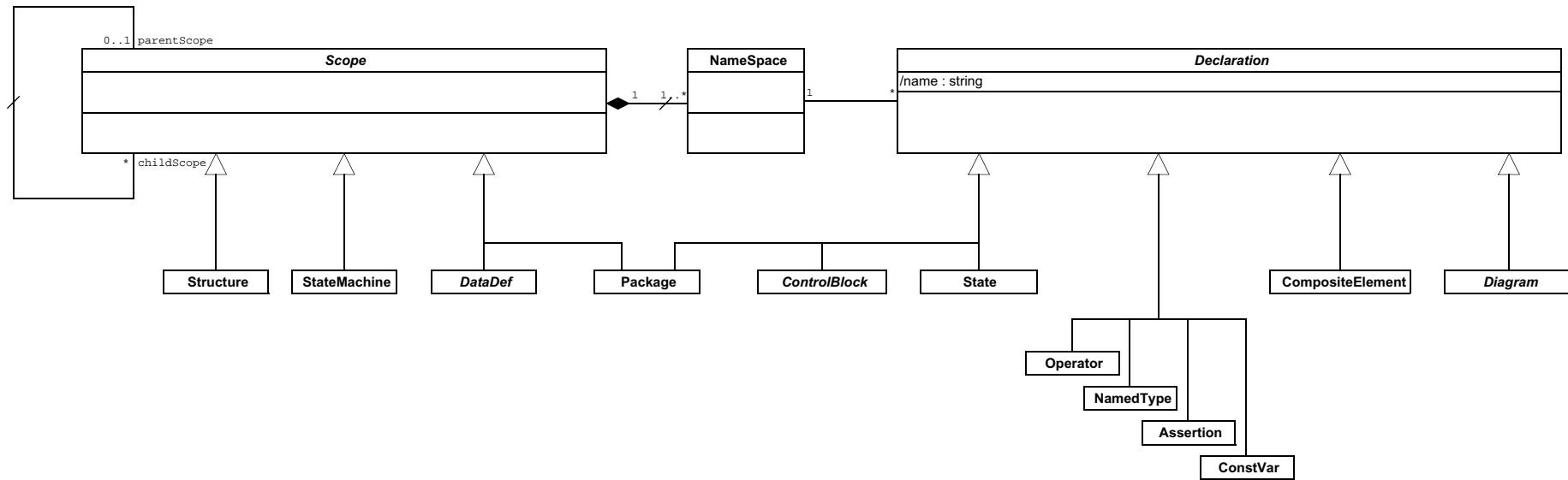
## Annotable (Tcl)



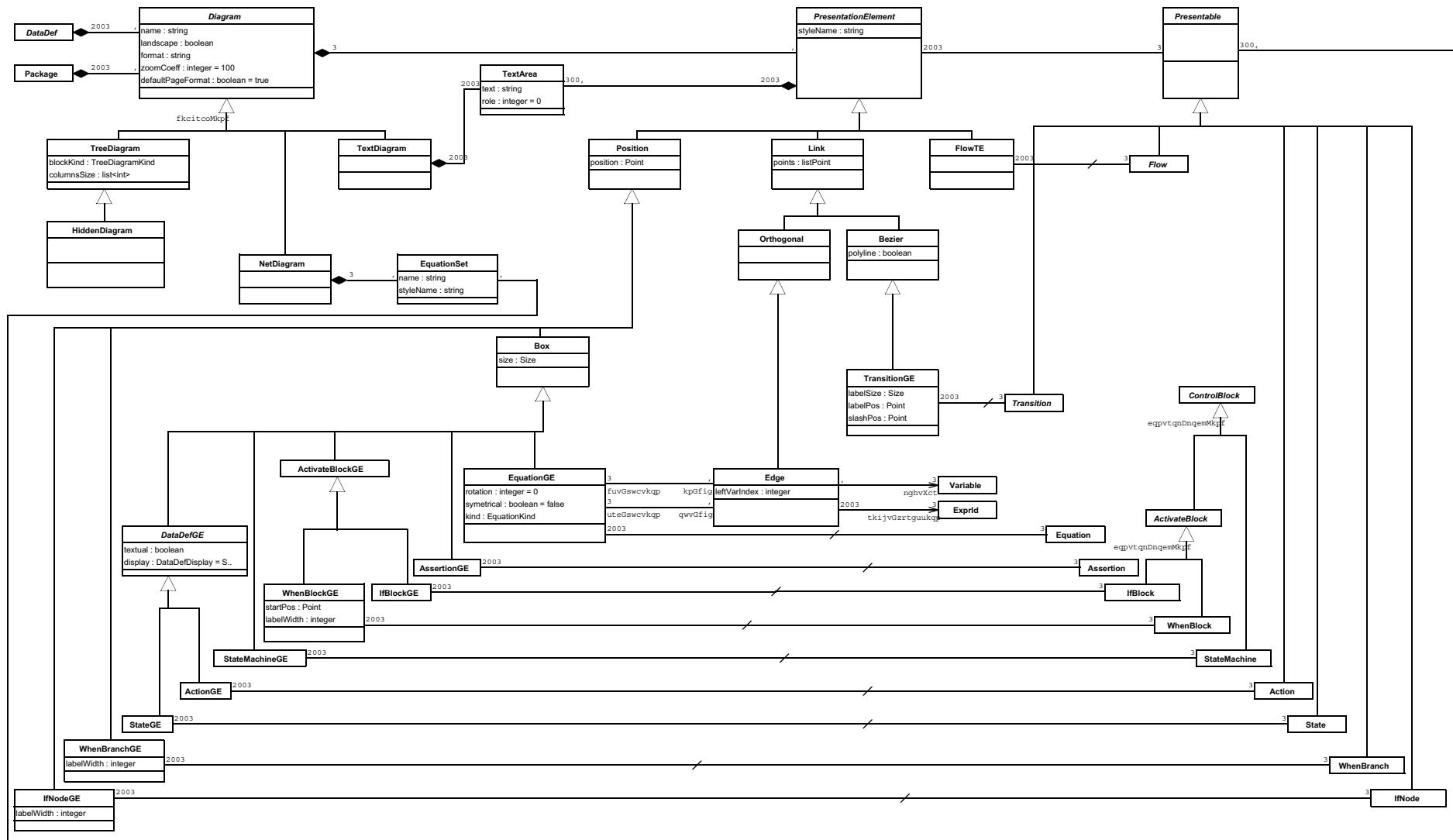
# Pragmas (Tcl)



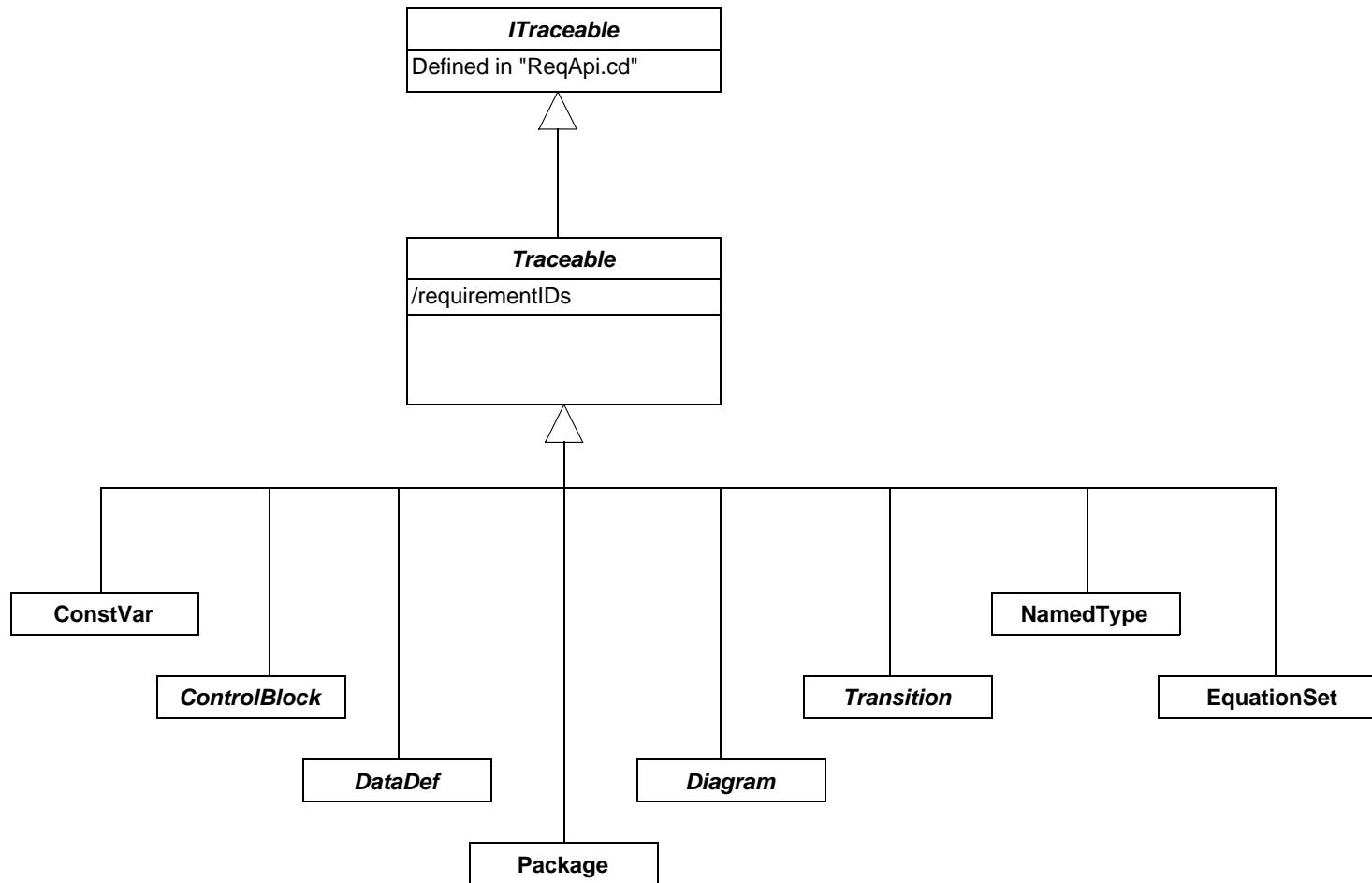
# Scopes (Tcl)



# Graphical (Tcl)



# Traceability (Tcl)

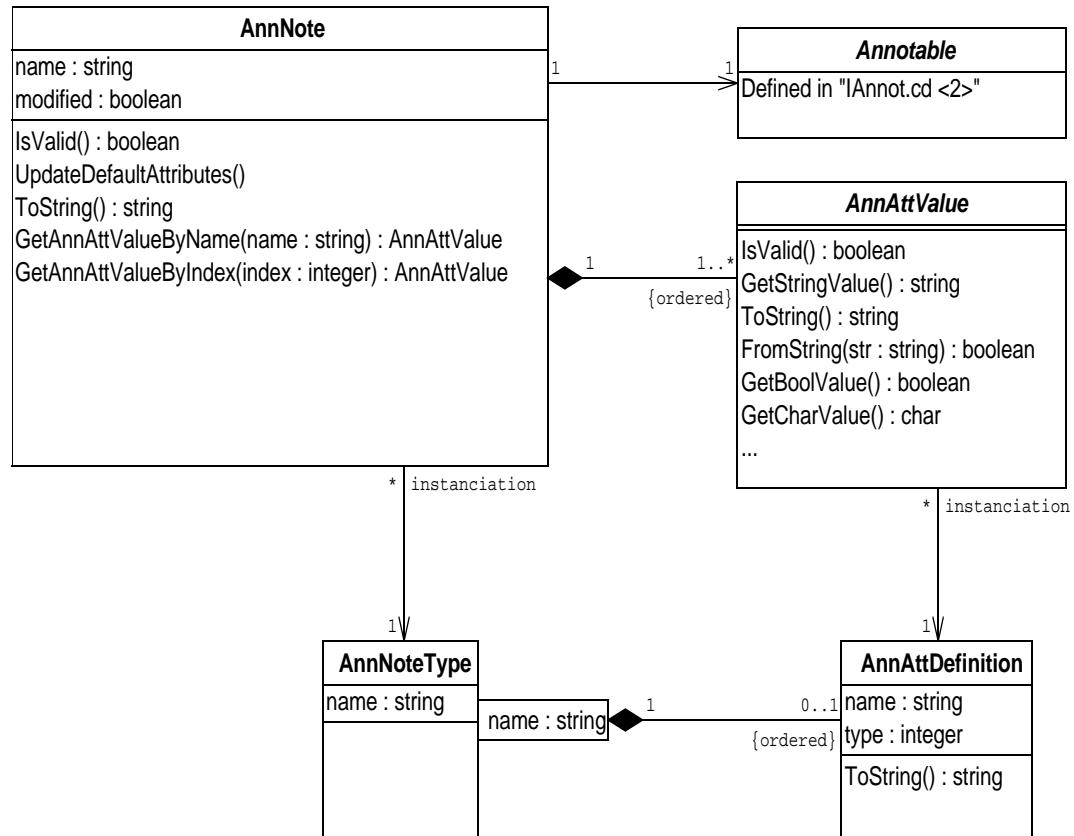


## 16 /Annotation Metamodels (Tcl API)

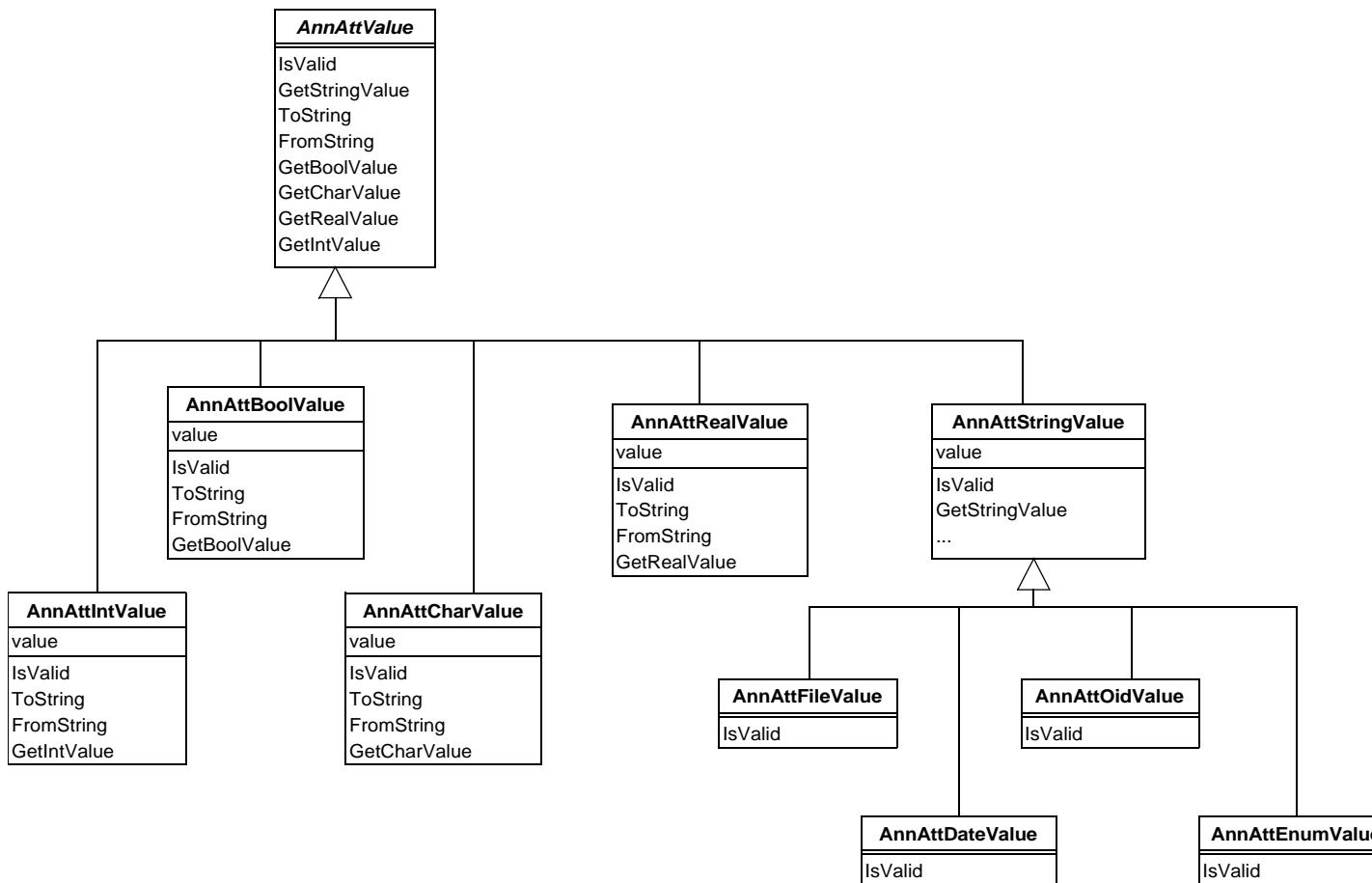
These metamodels present the data structures that give access to SCADE Suite annotations mechanisms using Tcl API:

- [“Notes \(Tcl\)”](#)
- [“Notes Attribute Values \(Tcl\)”](#)
- [“Notes Property Values \(Tcl\)”](#)

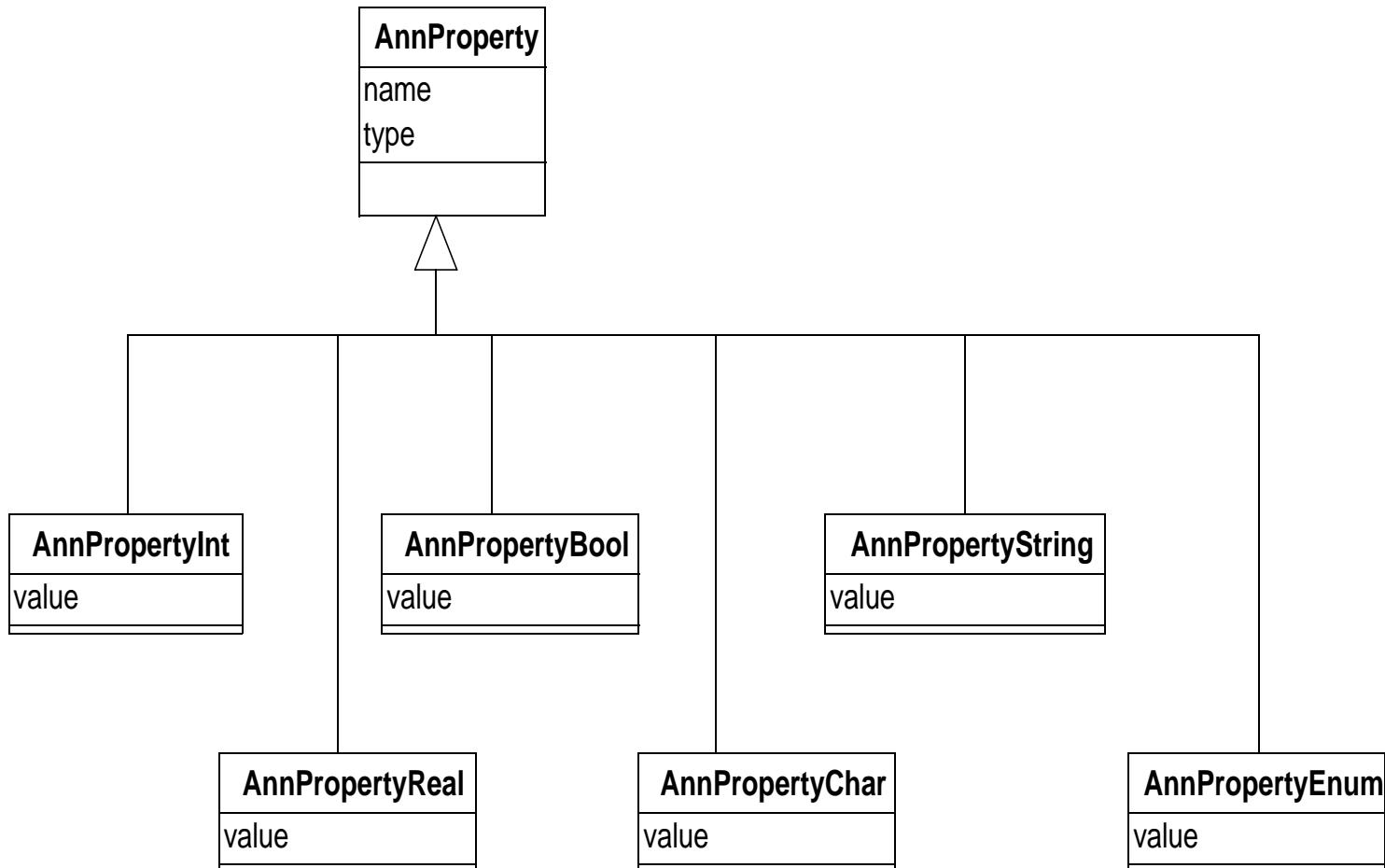
# Notes (Tcl)



# Notes Attribute Values (Tcl)

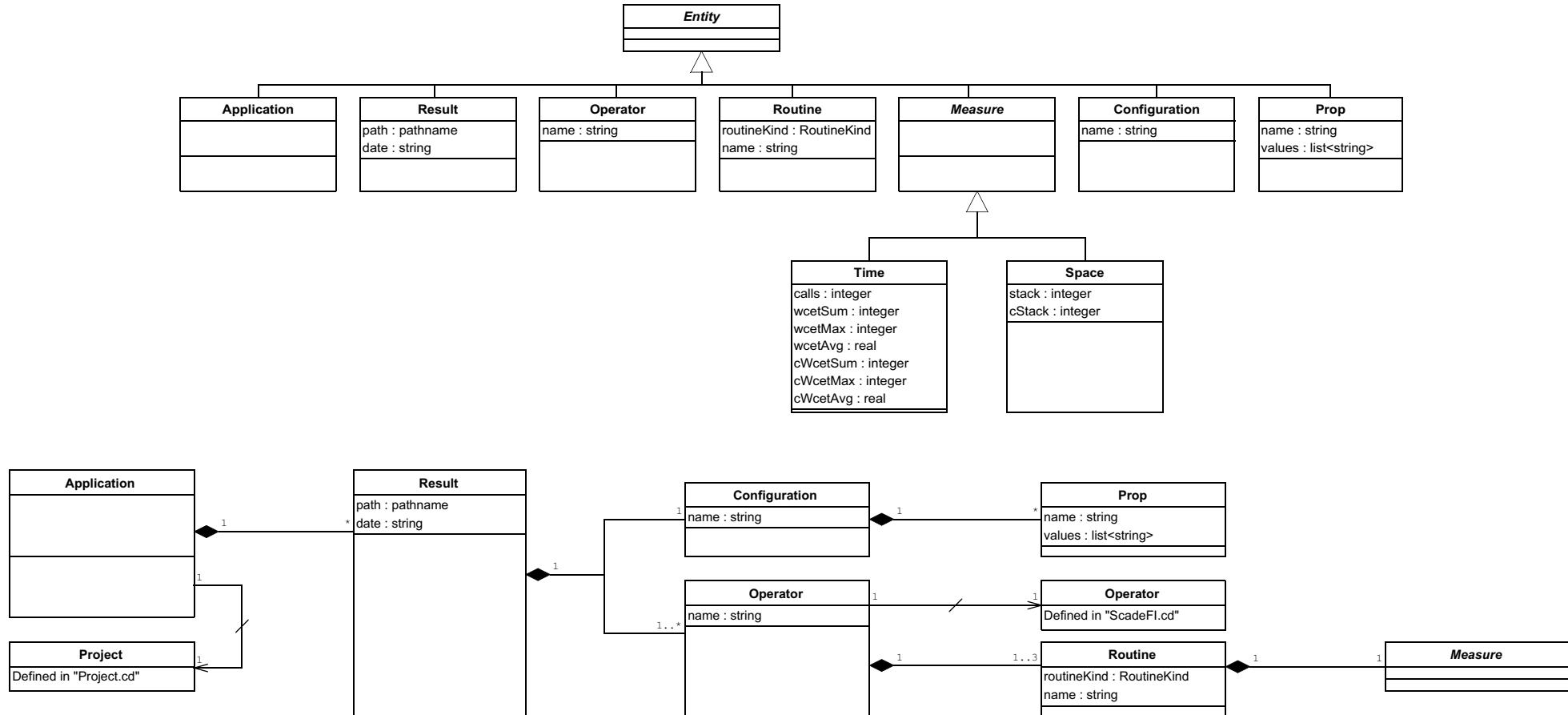


## Notes Property Values (Tcl)



# 17 /Timing and Stack Analysis Metamodel (Tcl API)

This metamodel presents the data structures that give access to SCADE Suite timing and stack analysis results Tcl API:



## Part 4

# SCADE Suite Metamodels for Java API

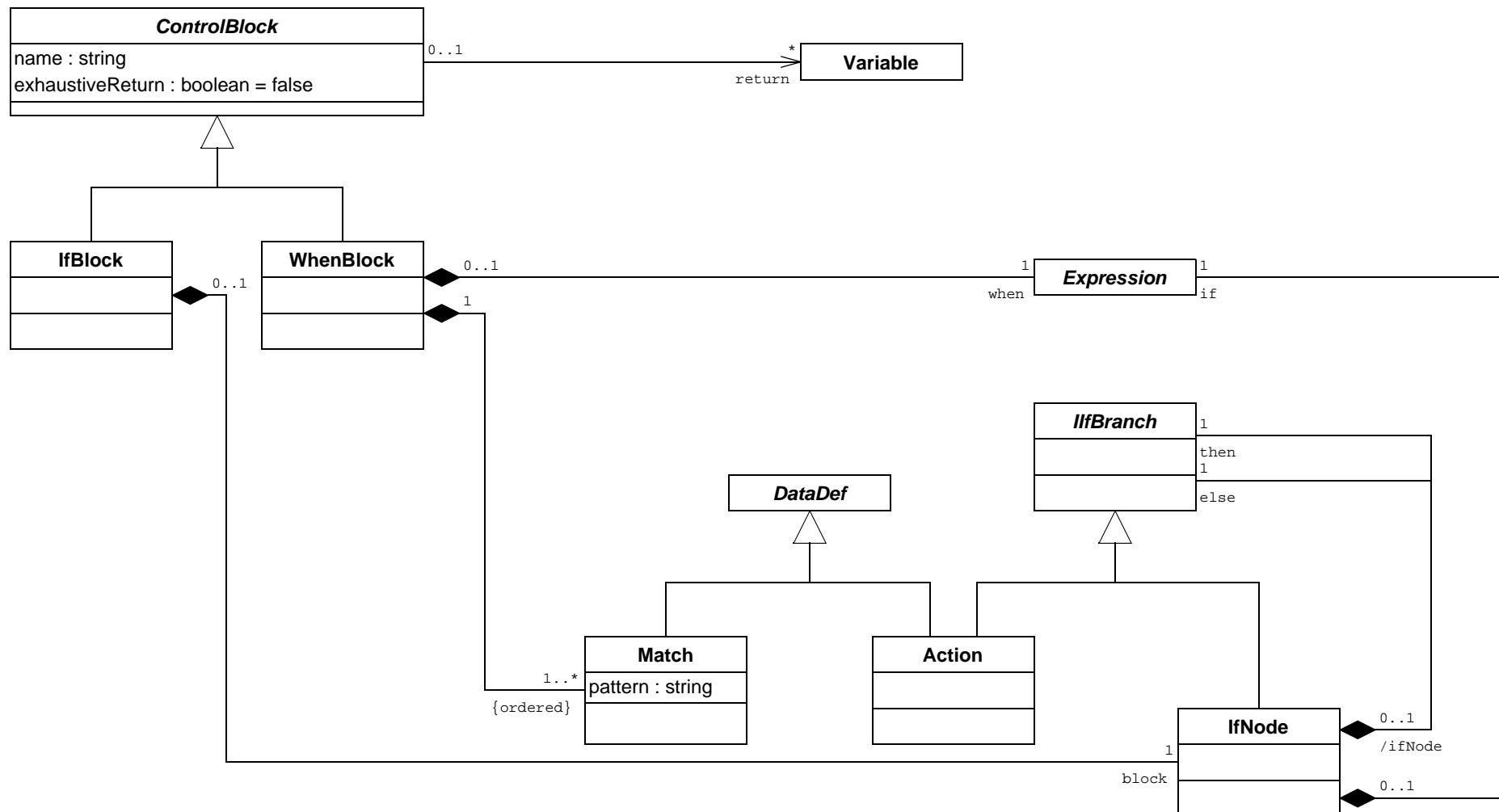
- 18/ ["Scade Language Metamodels"](#)
- 19/ ["Scade Graphics Metamodels"](#)

# 18 /Scade Language Metamodels

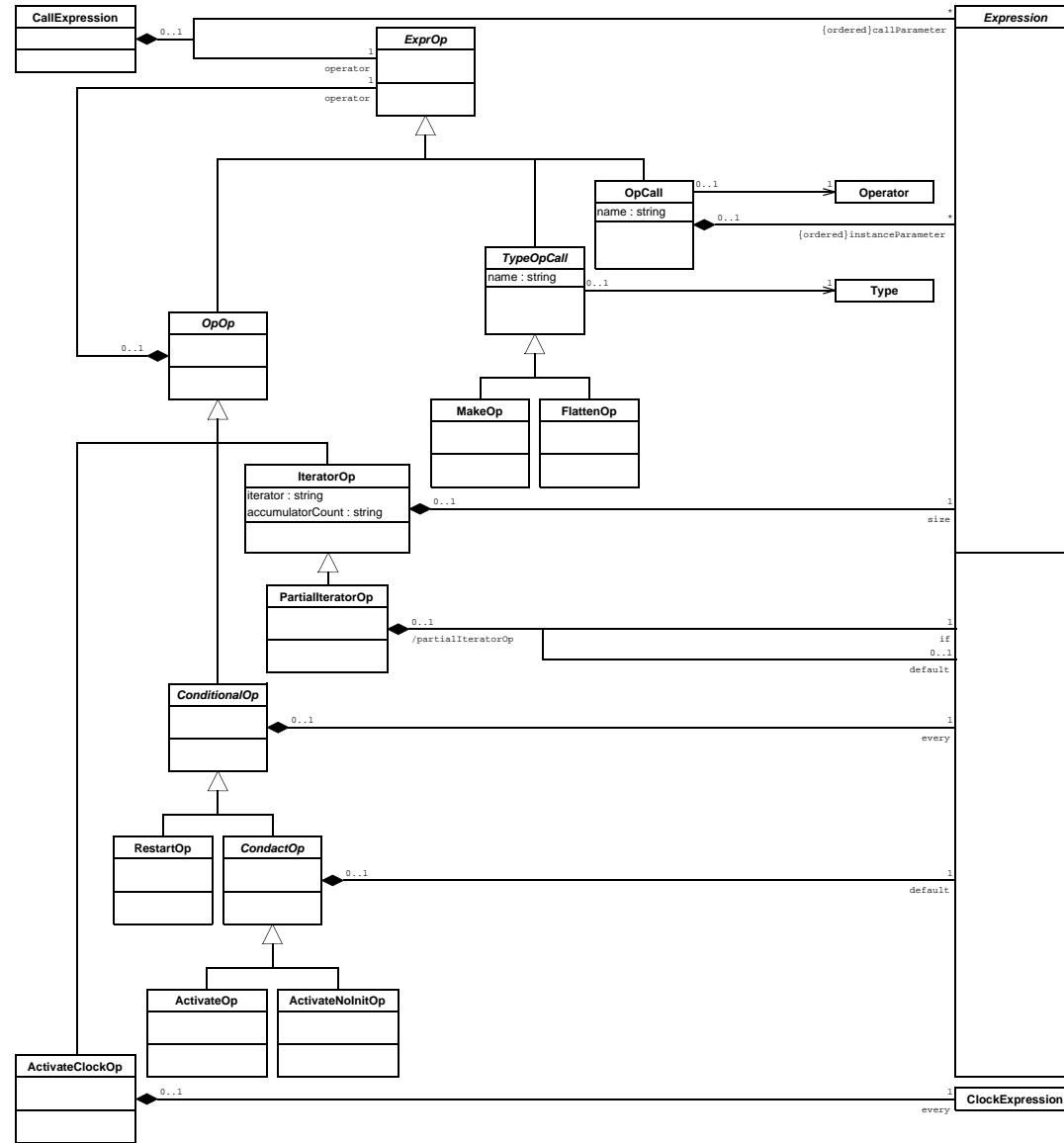
These metamodels present the data structures that give access to Scade language constructs using Java API:

- [“Activate Block”](#)
- [“Call Expression”](#)
- [“Clocks”](#)
- [“Data Definition”](#)
- [“Data Flow”](#)
- [“Data Operators”](#)
- [“Expressions”](#)
- [“Flow Operators”](#)
- [“Inheritance”](#)
- [“Operators”](#)
- [“Pragmas”](#)
- [“Roots”](#)
- [“Simple and Control Operators”](#)
- [“Specialization”](#)
- [“State Machines”](#)
- [“Type”](#)
- [“Variable”](#)

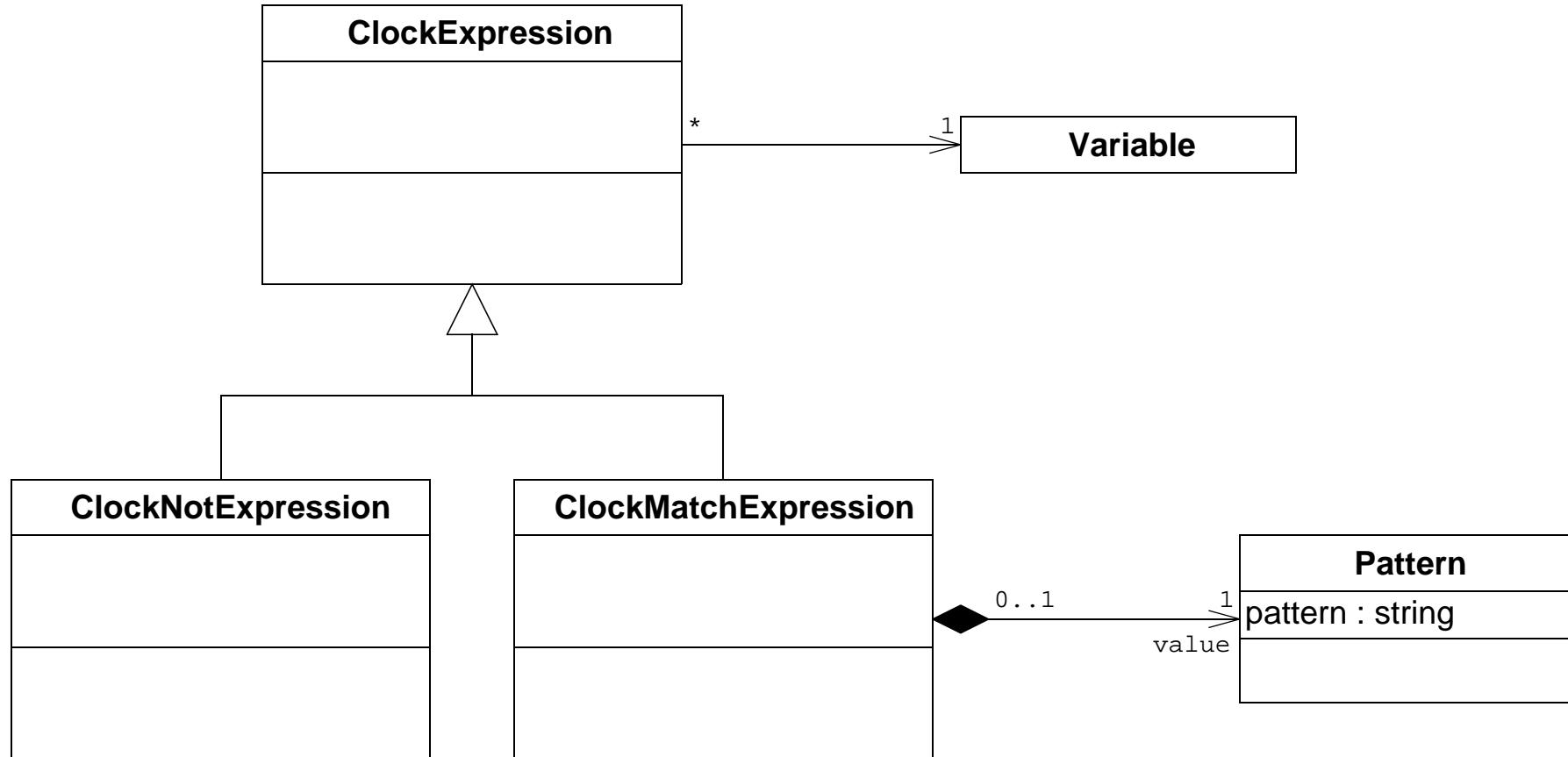
## Activate Block



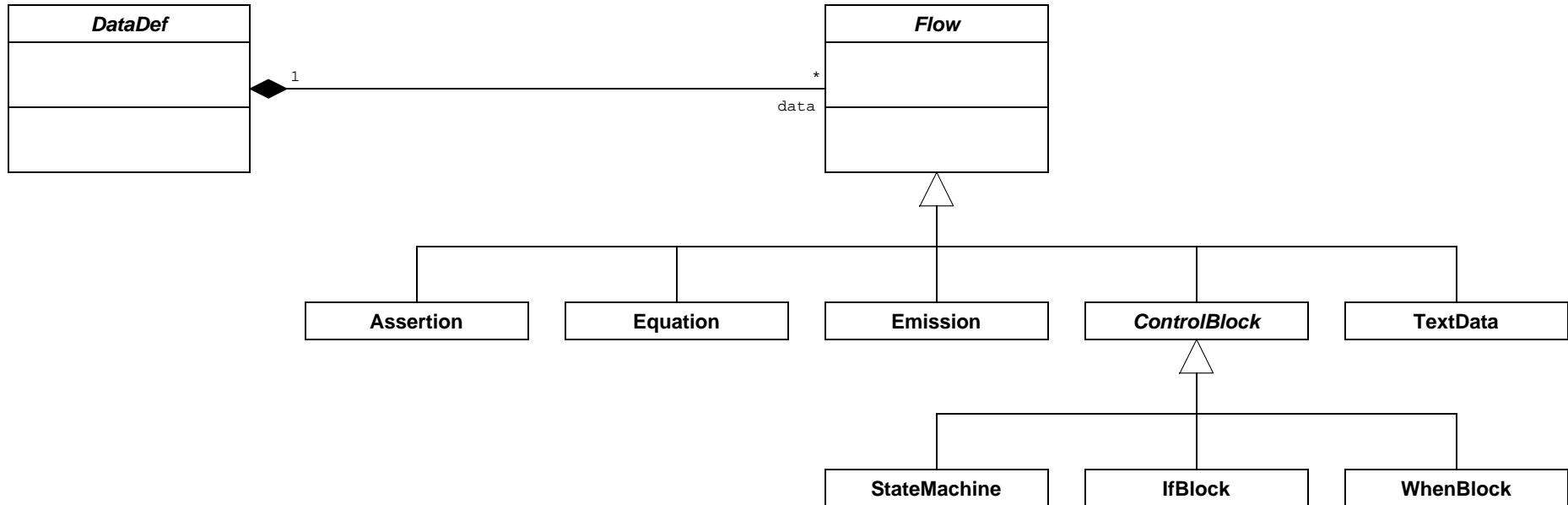
# Call Expression



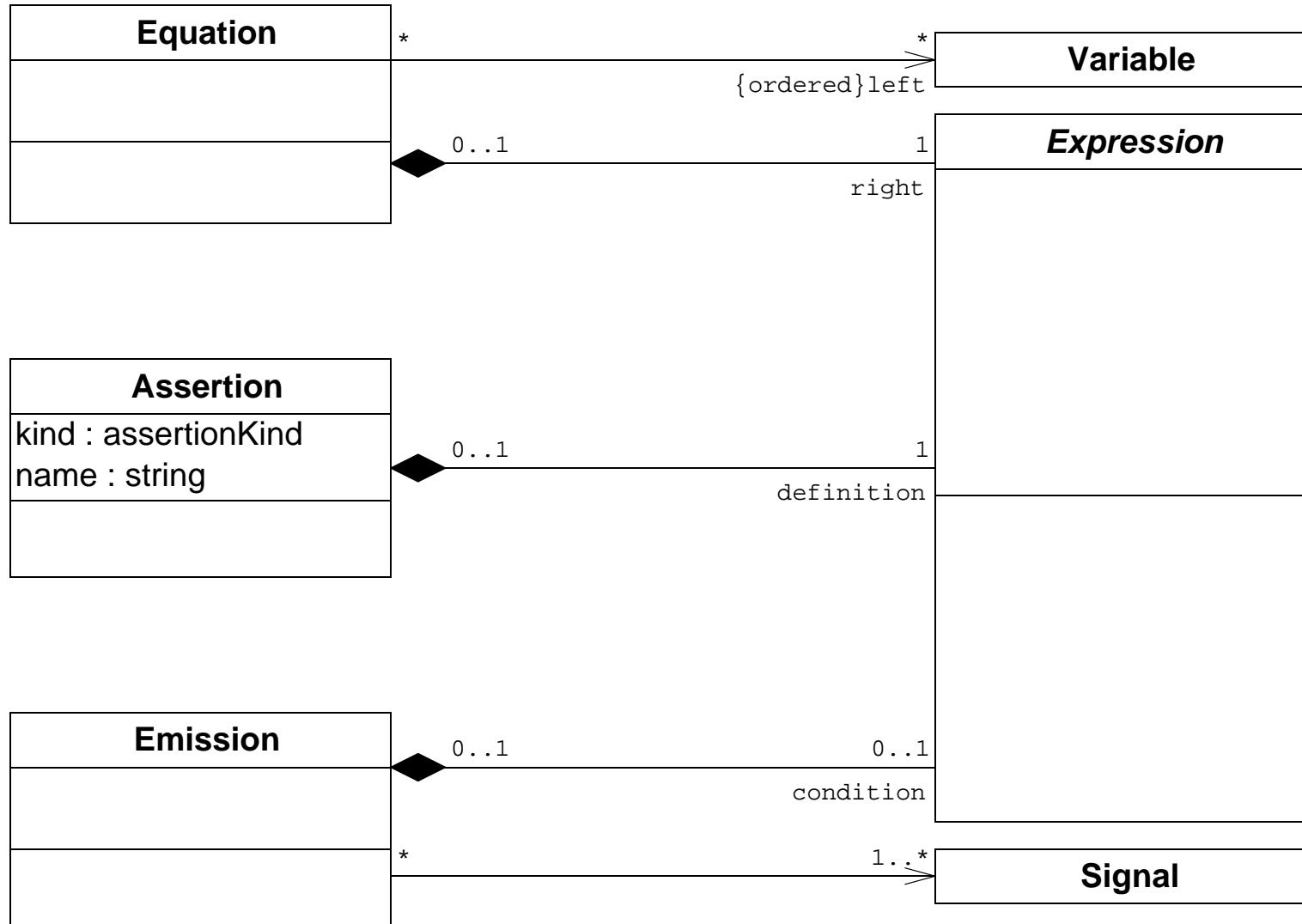
## Clocks



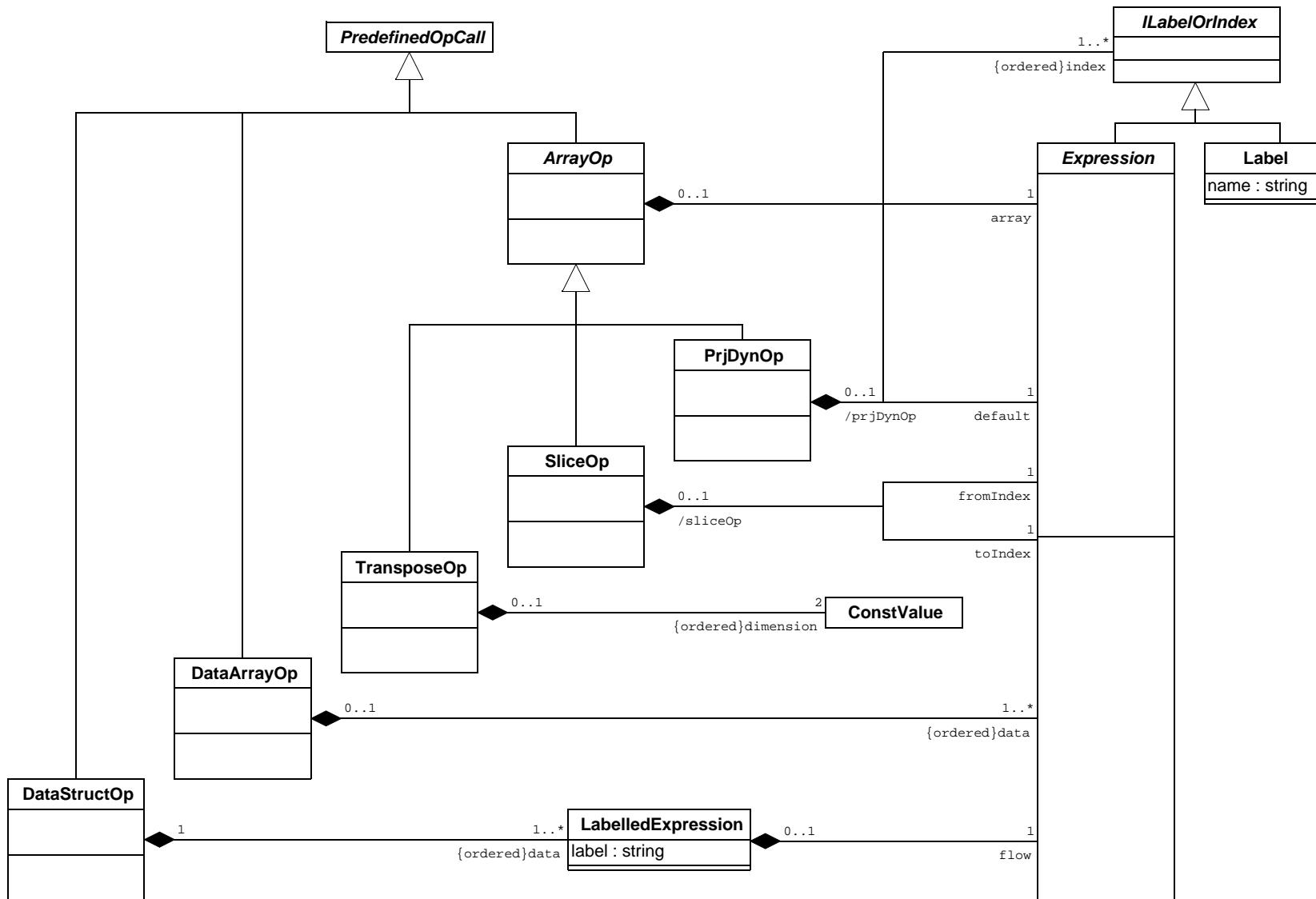
# Data Definition



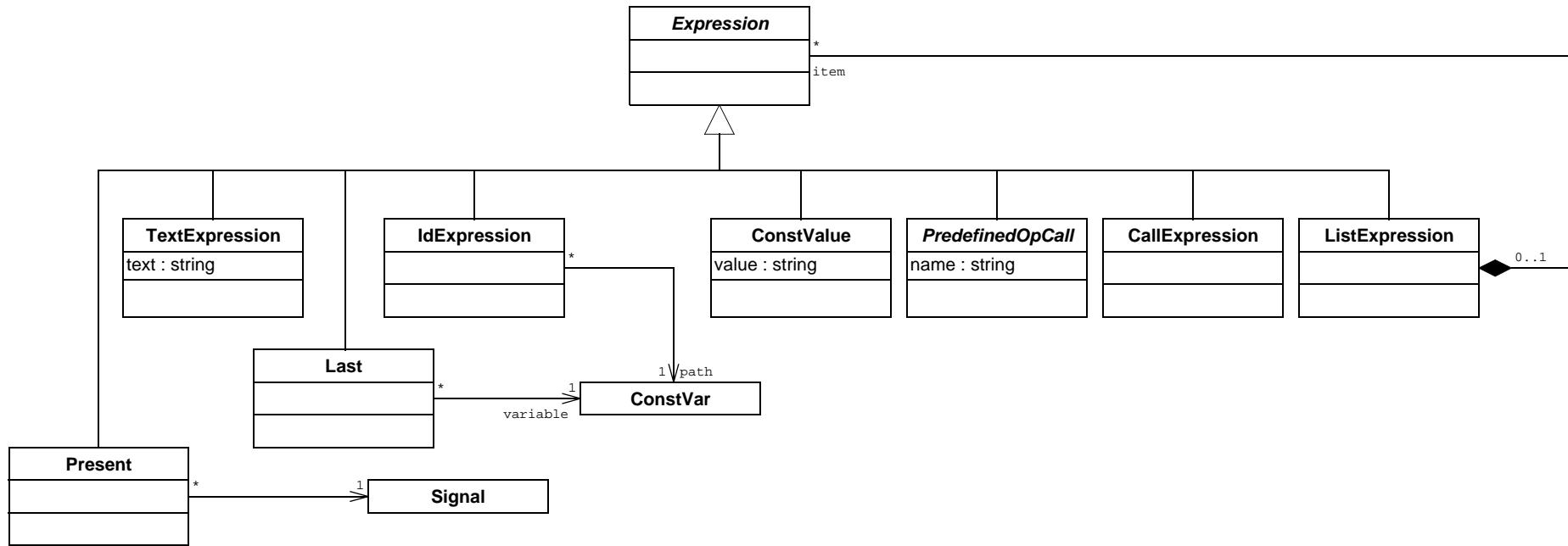
## Data Flow



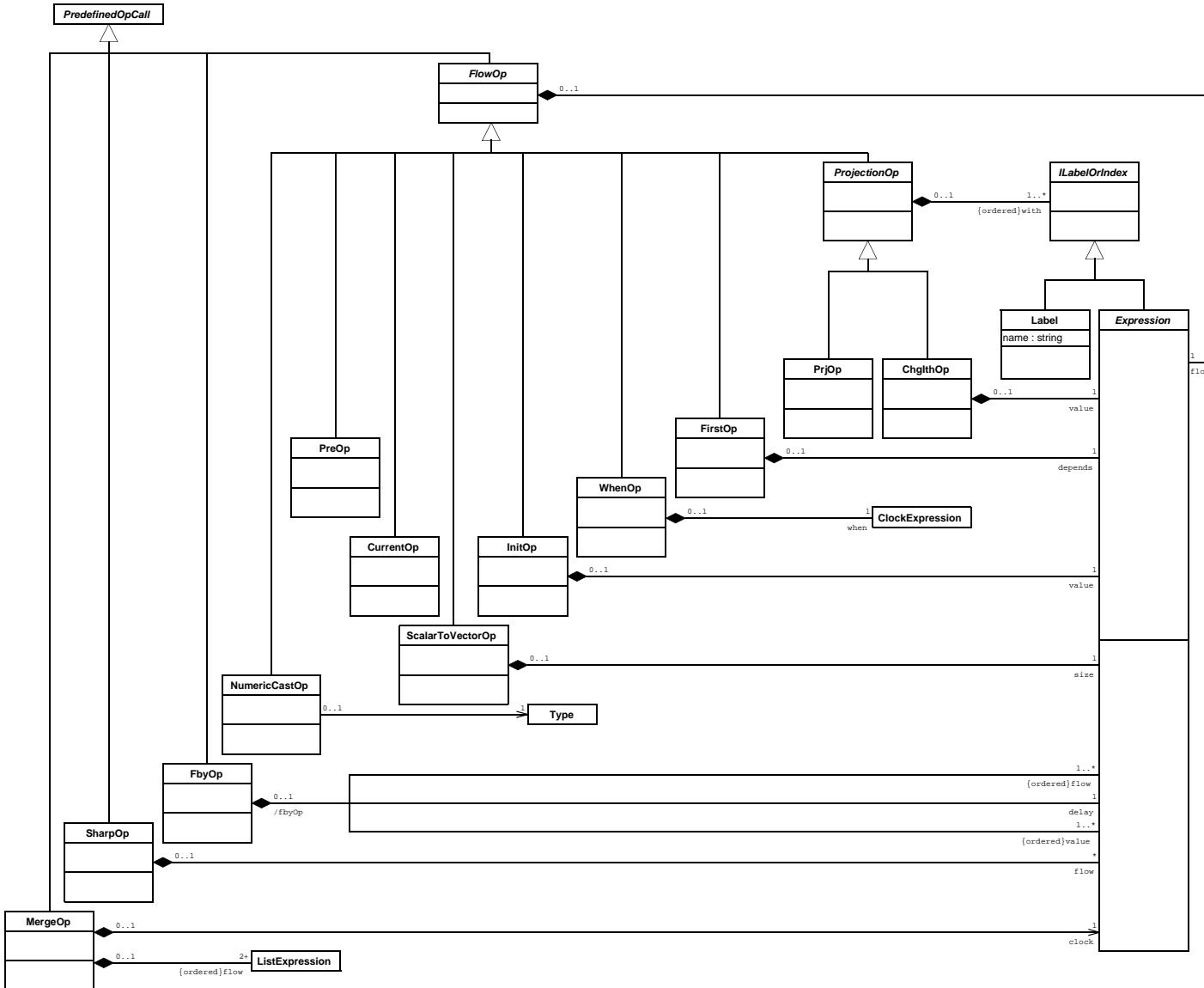
# Data Operators



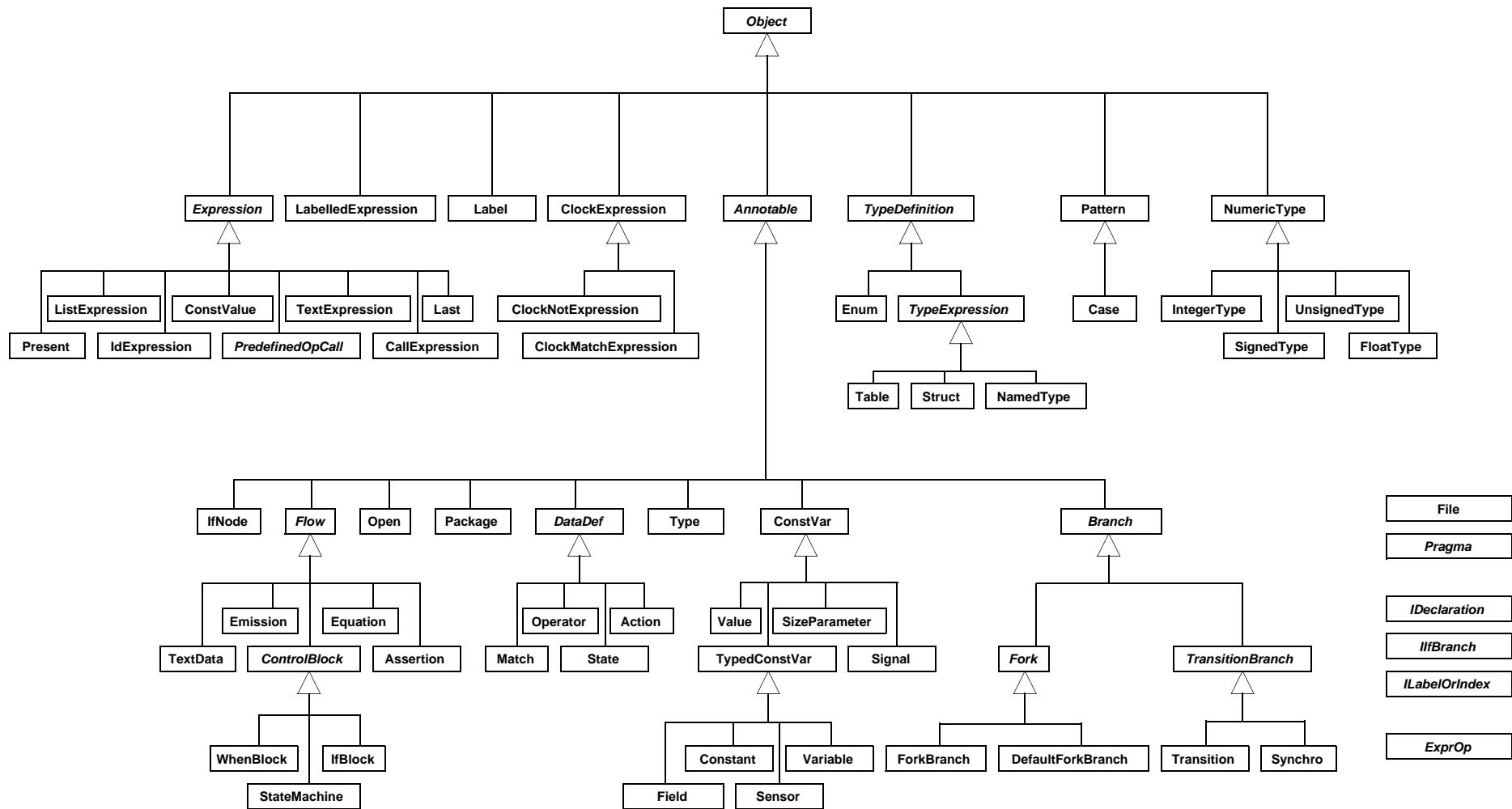
# Expressions



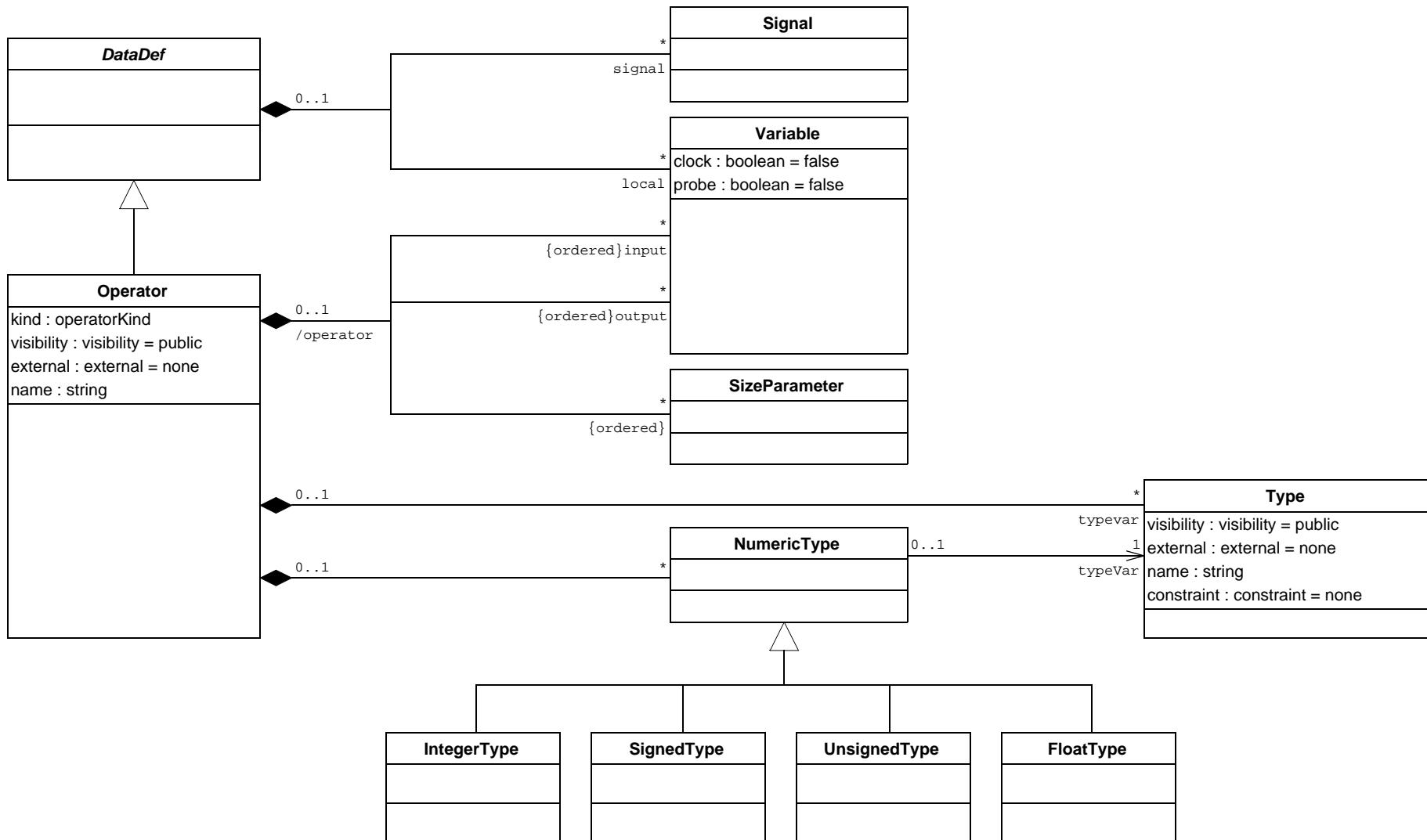
# Flow Operators



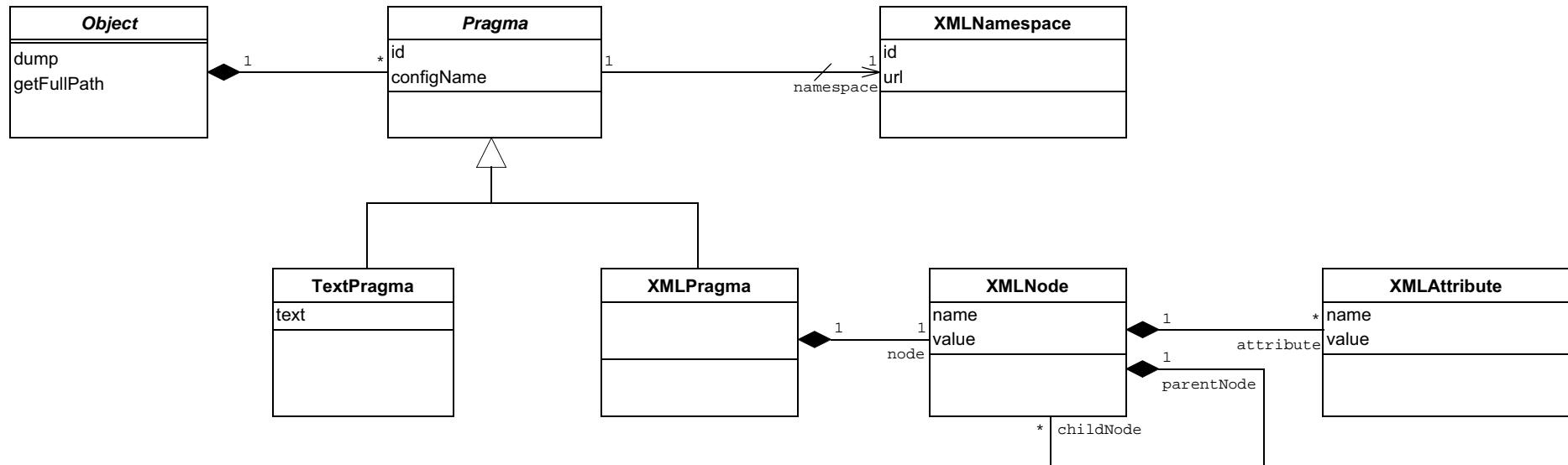
# Inheritance



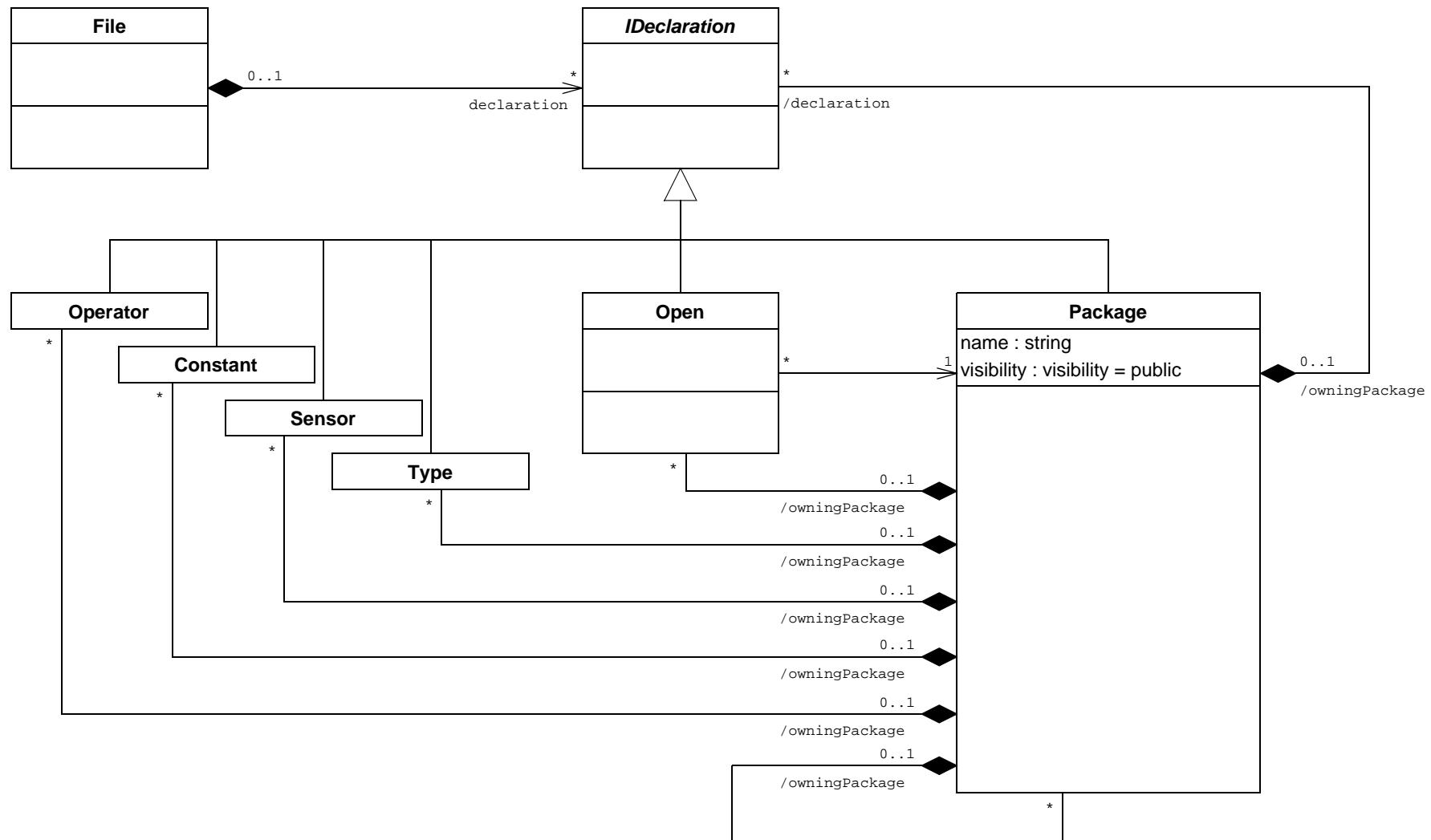
# Operators



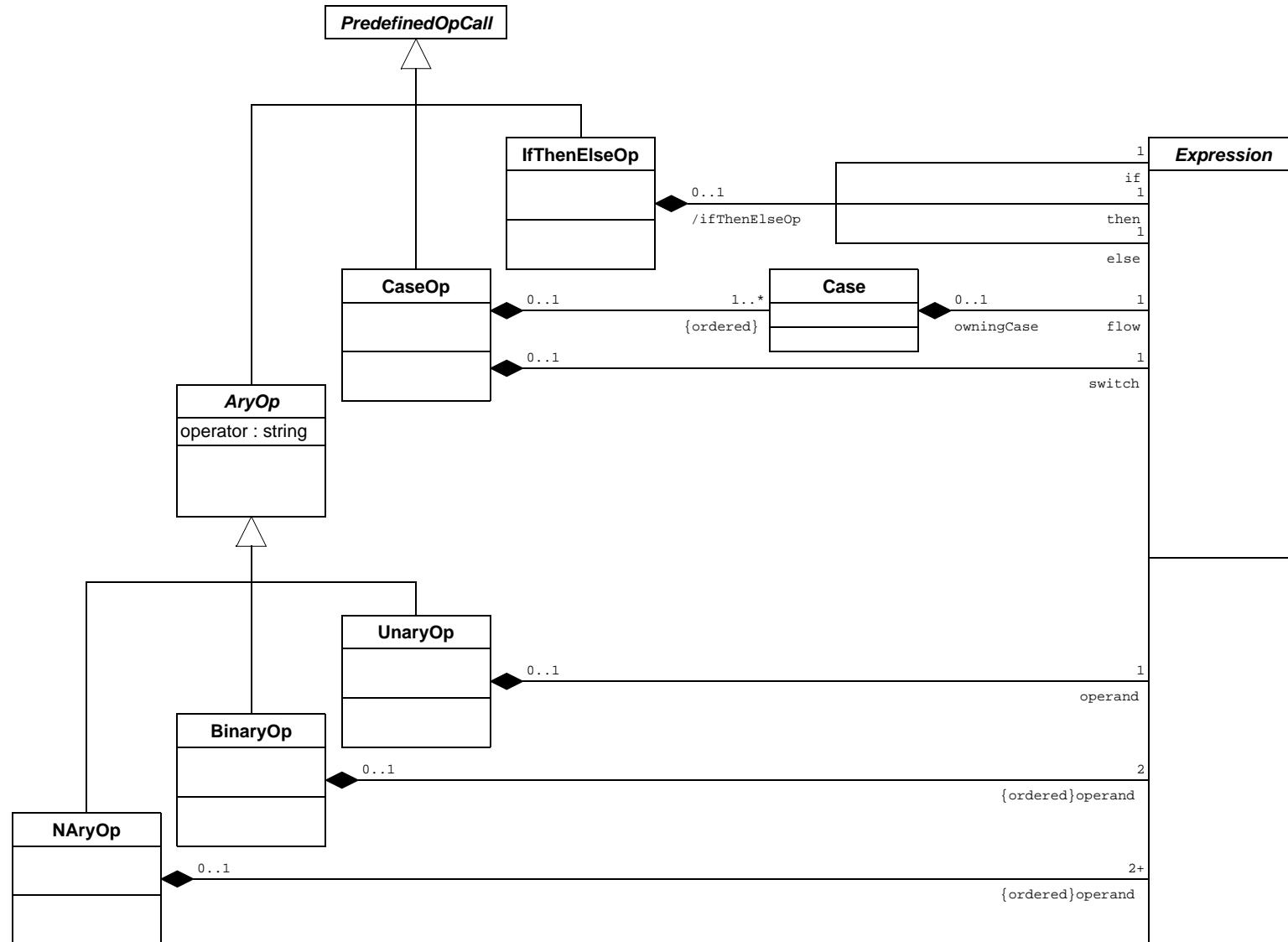
# Pragmas



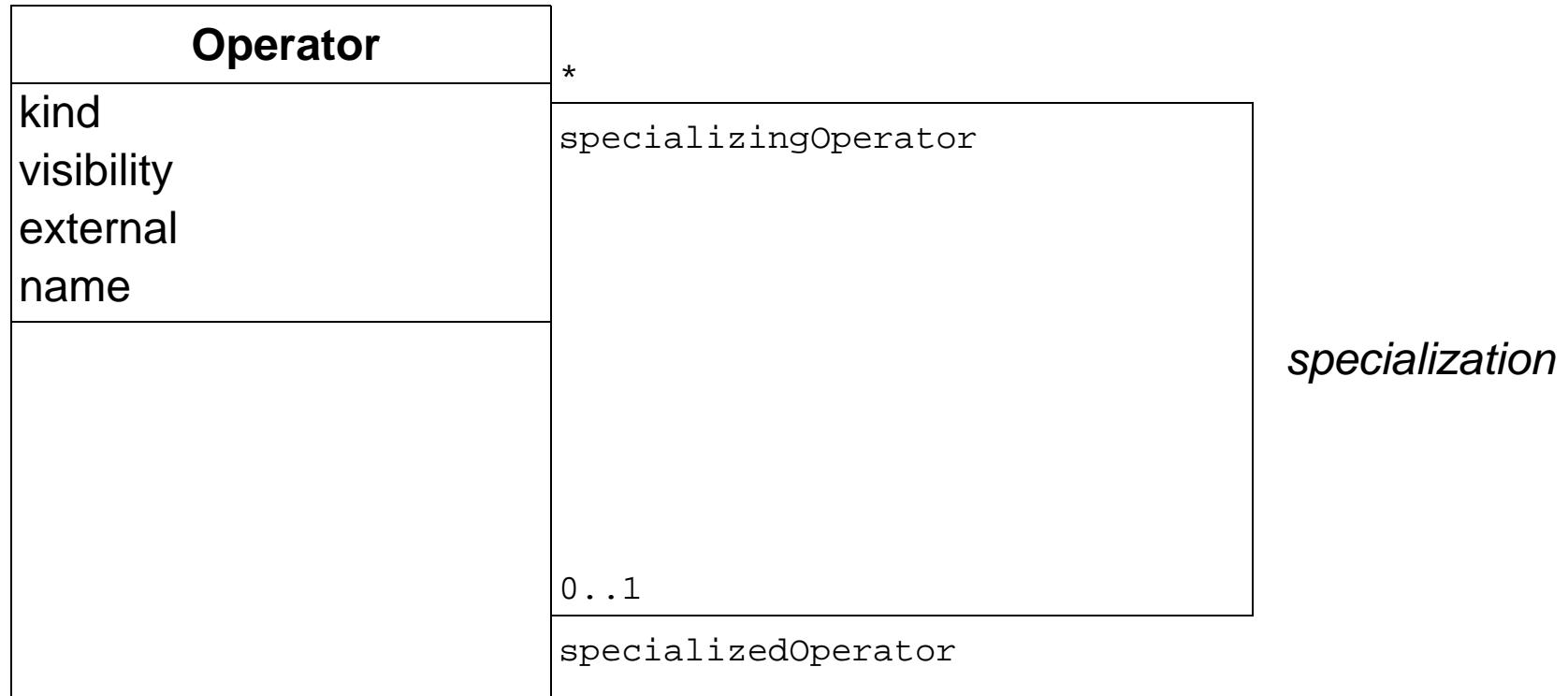
# Roots



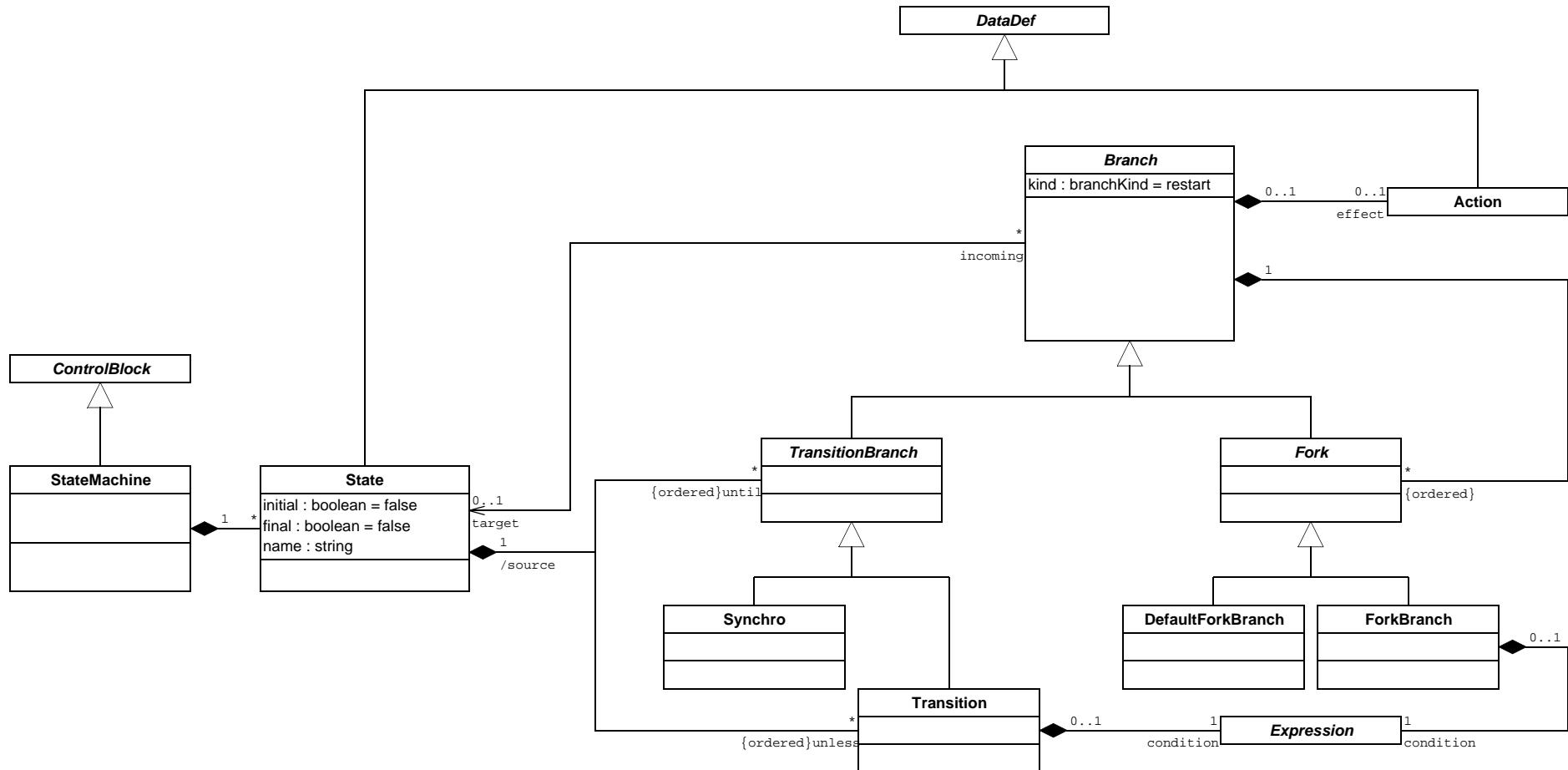
# Simple and Control Operators



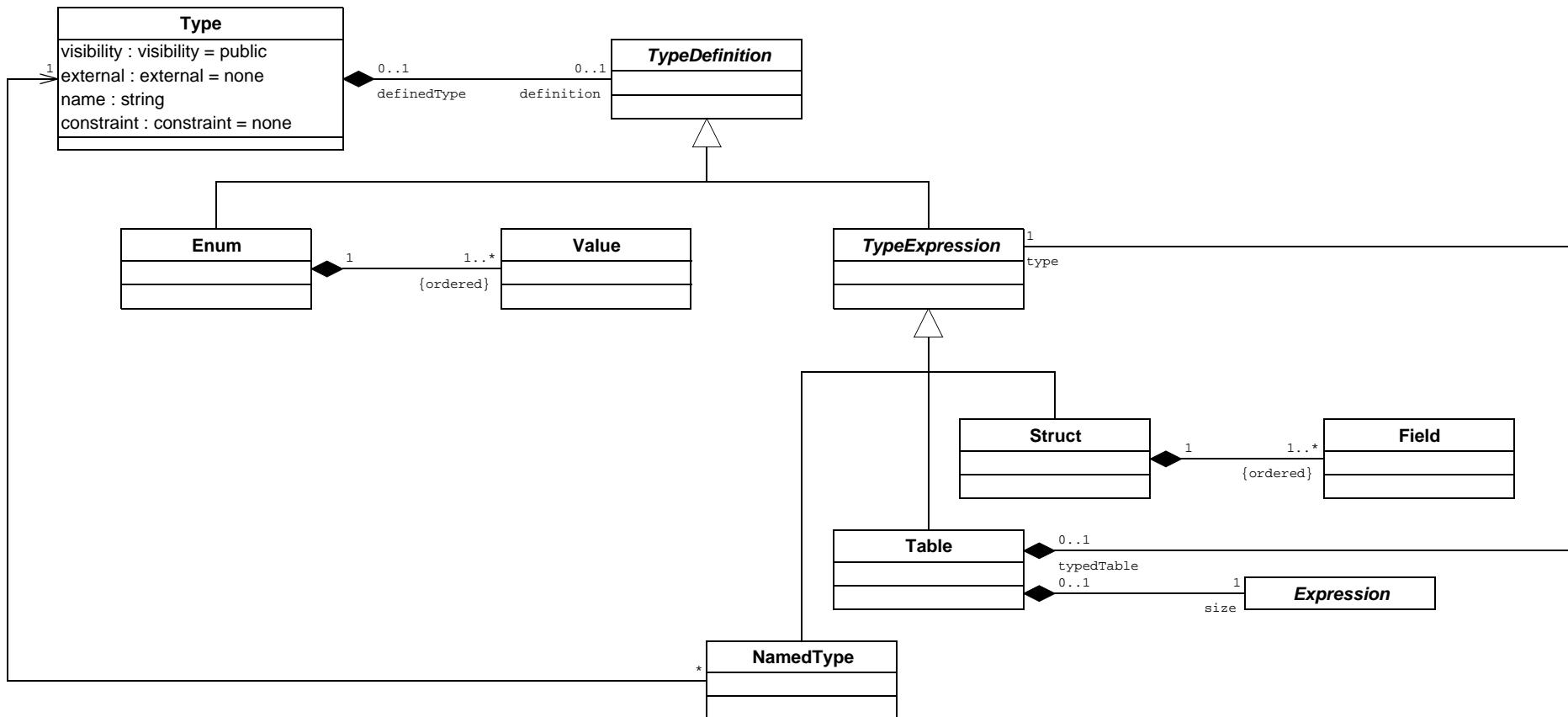
## Specialization



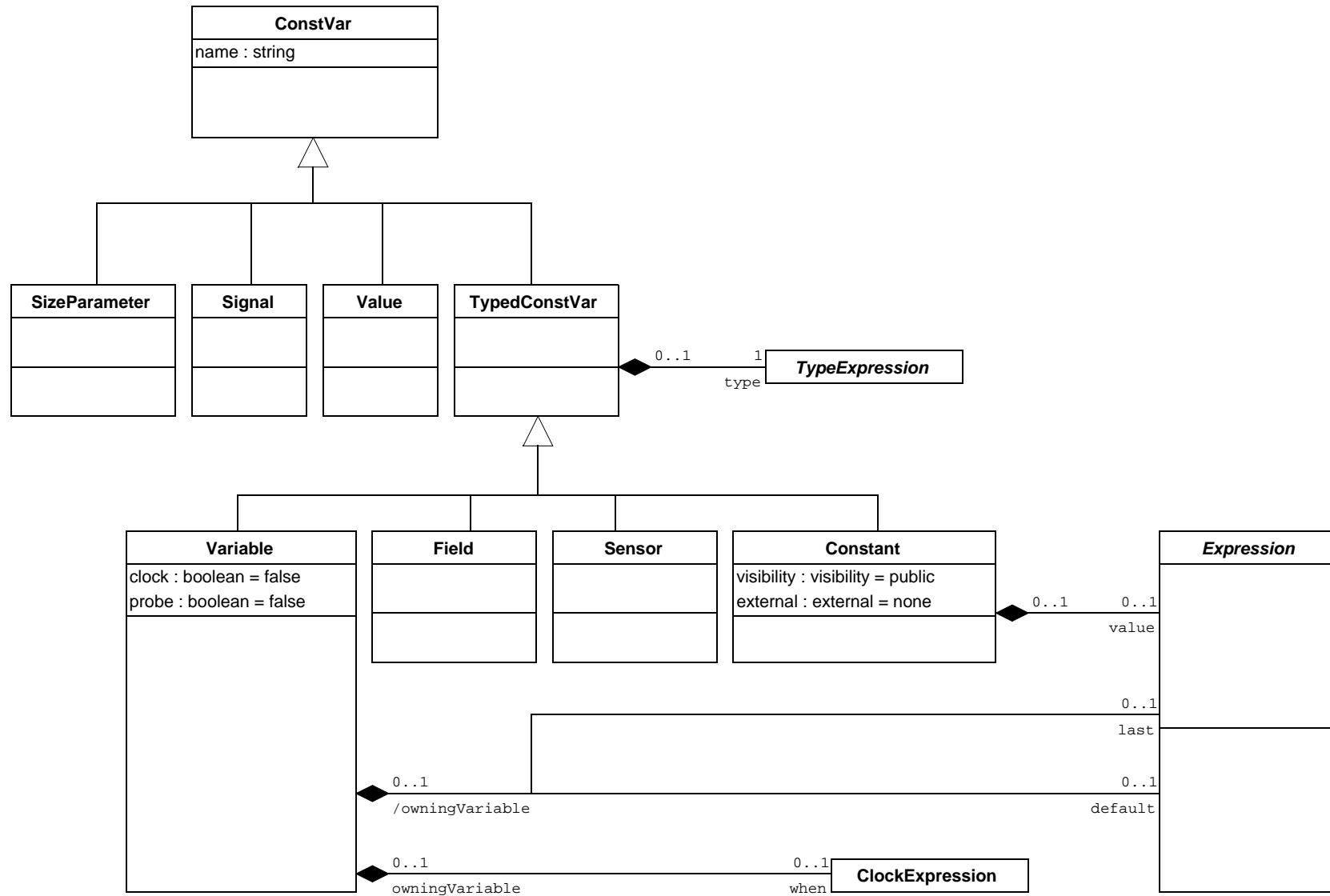
# State Machines



# Type



# Variable

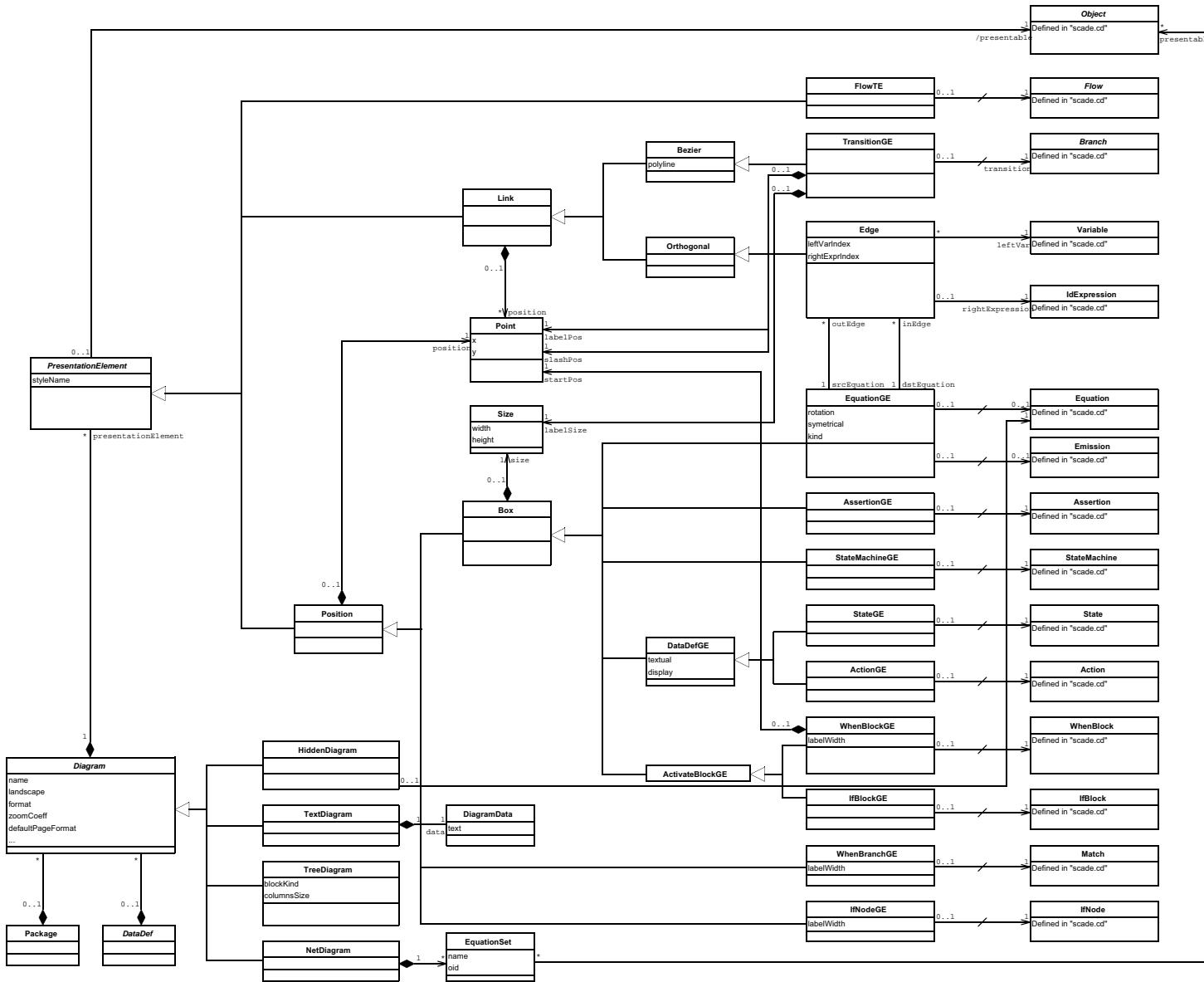


# 19 /Scade Graphics Metamodels

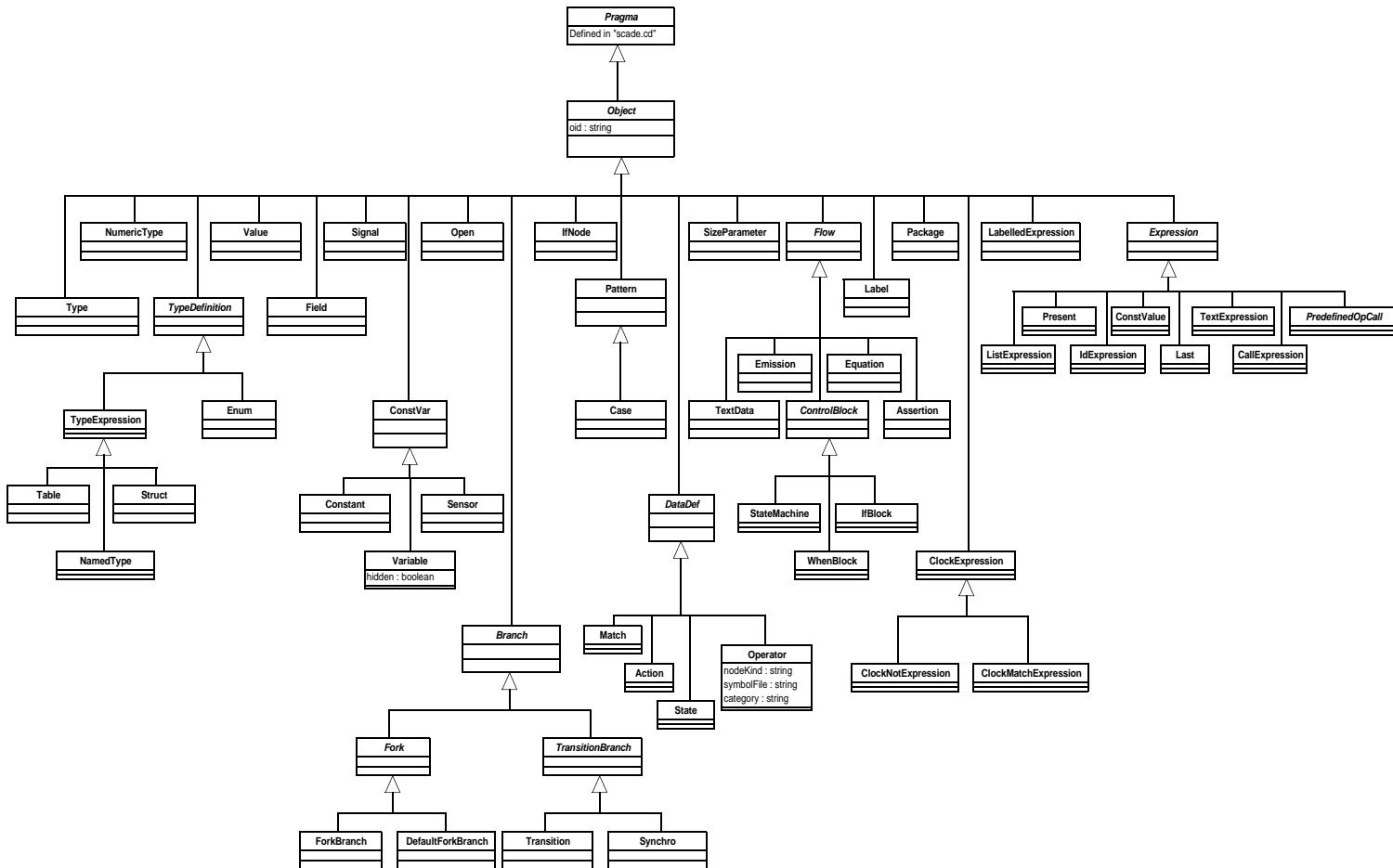
These metamodels present the data structures that give access to the graphic constructs of the Scade language using Java API:

- [“Diagrams”](#)
- [“Graphical Pragmas”](#)

# Diagrams



# Graphical Pragmas



## Part 5

# SCADE Display Metamodels

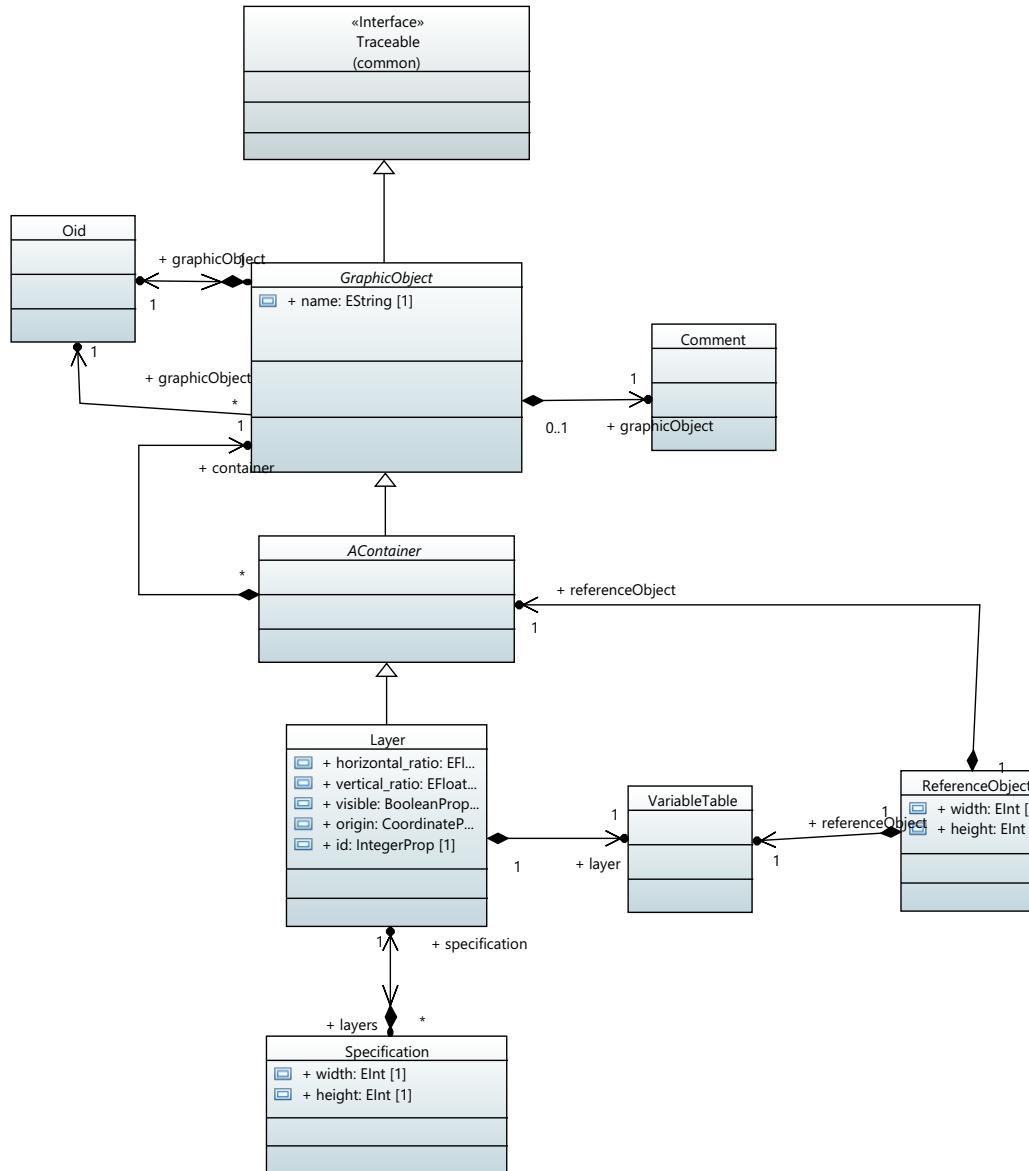
- 20/ [SCADE Display Metamodels](#)
- 21/ [SCADE Display Mapping Files Metamodels](#)

# 20 /SCADE Display Metamodels

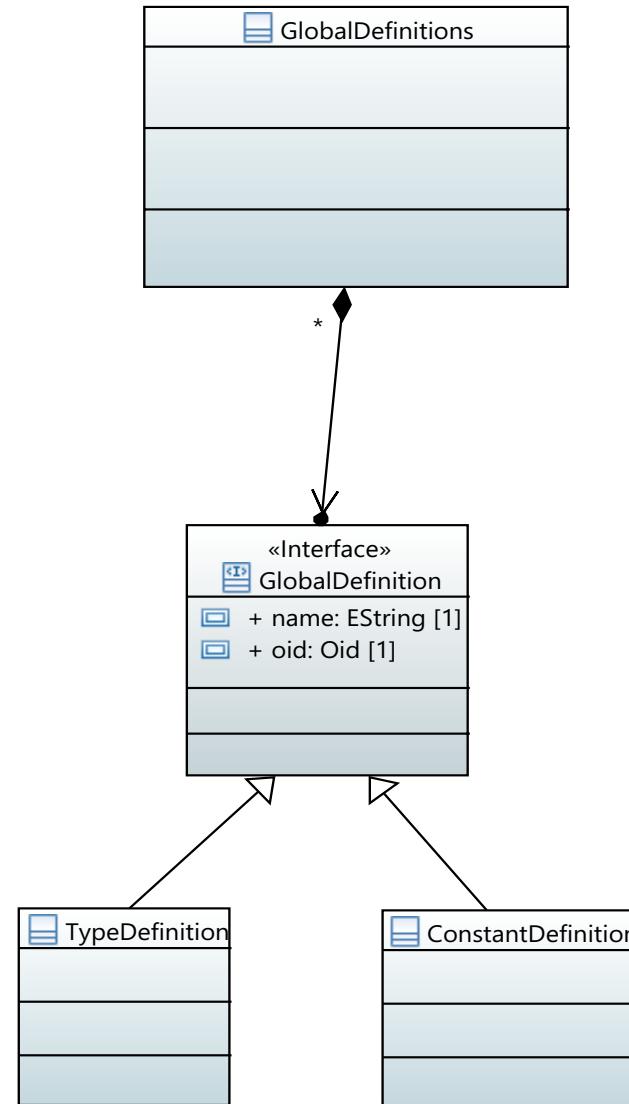
These metamodels present the data structure that give access to SCADE Display models using Python API or Java API:

- [“Specification and Reference Object Model”](#)
- [“Global Definitions”](#)
- [“Constants”](#)
- [“Types”](#)
- [“Variable Dictionary”](#)
- [“Graphic Primitives”](#)
- [“Text Primitives”](#)
- [“Masks”](#)
- [“Containers”](#)
- [“Interactive Primitives”](#)
- [“Reference Primitives”](#)
- [“Path Primitives”](#)
- [“Properties - 1”](#)
- [“Properties - 2”](#)
- [“Resource Tables”](#)

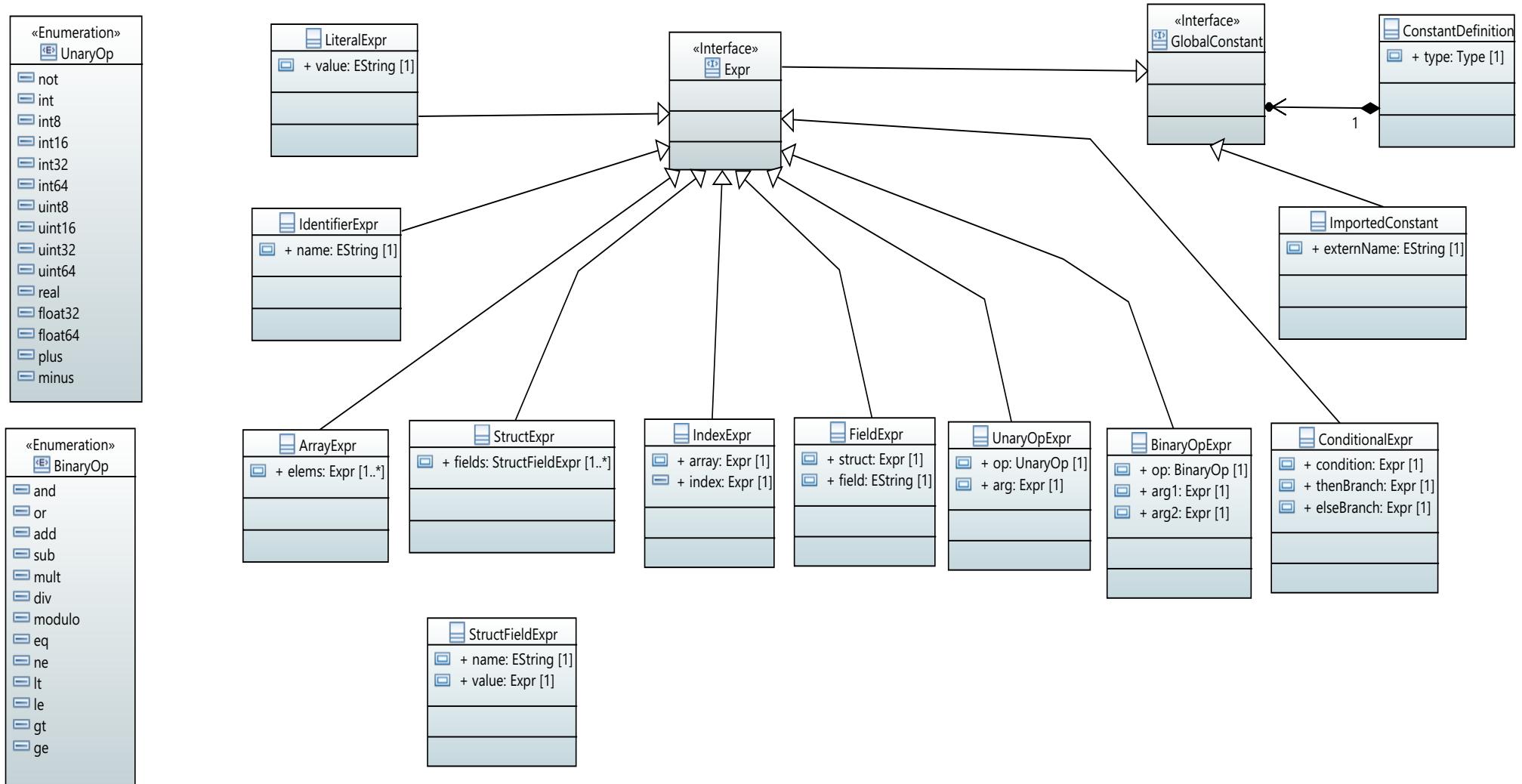
# Specification and Reference Object Model



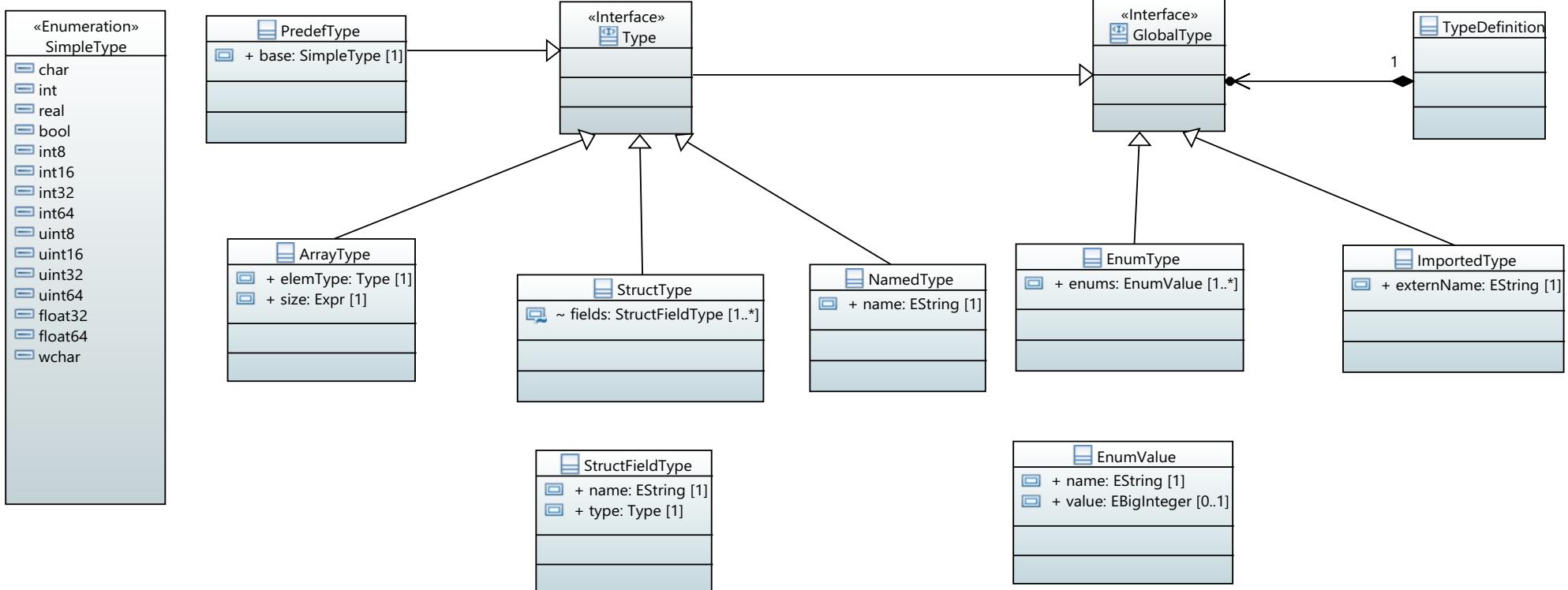
## Global Definitions



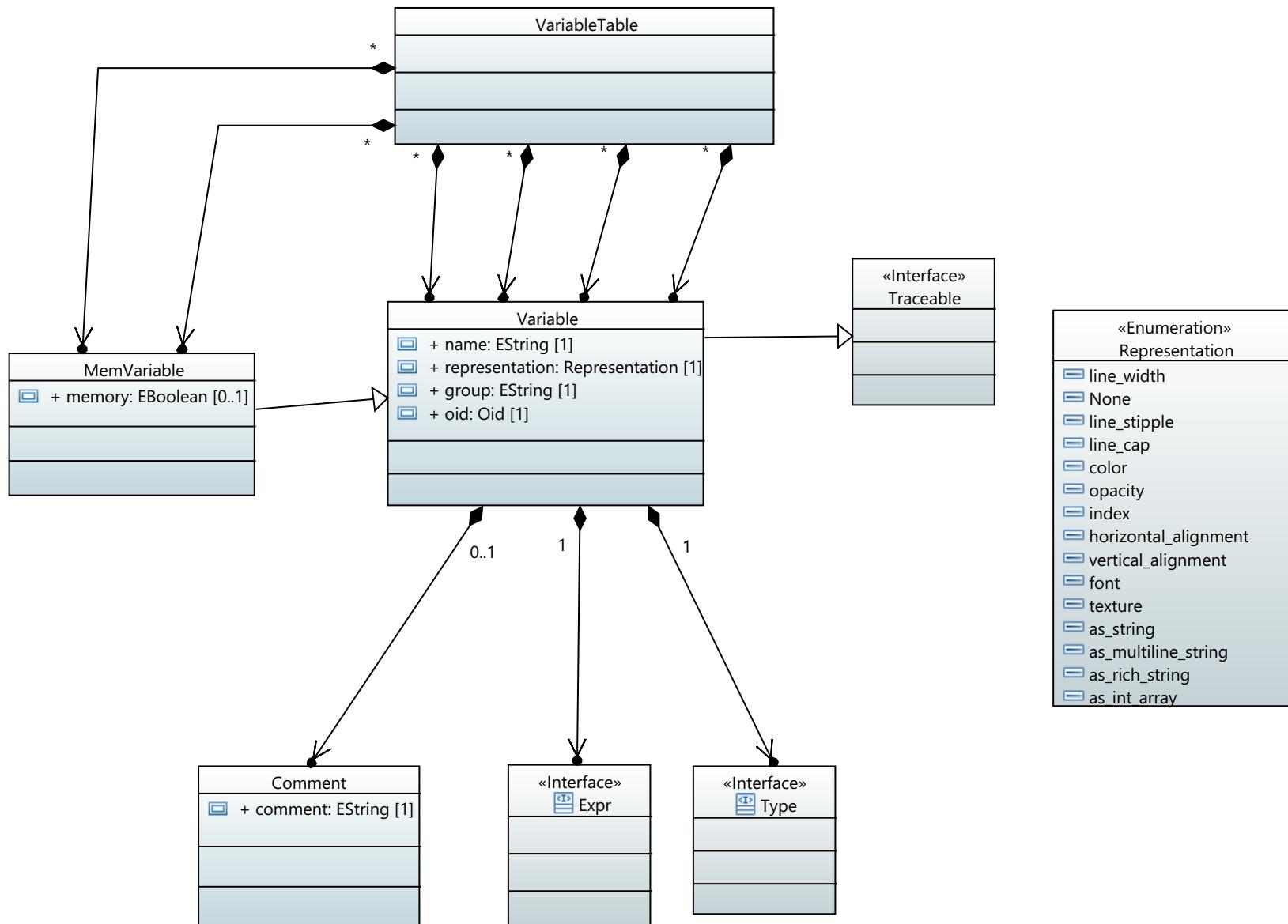
# Constants



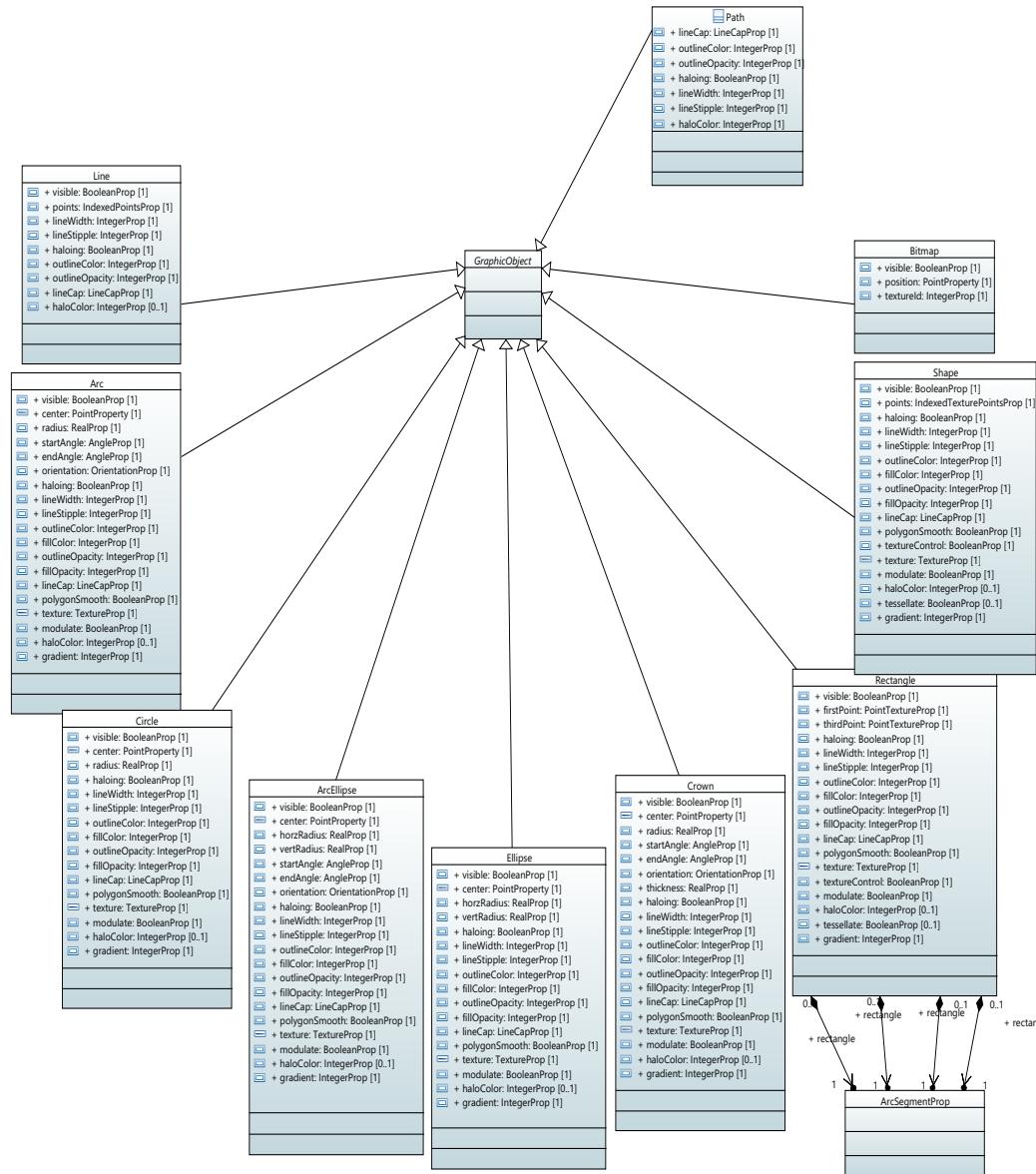
# Types



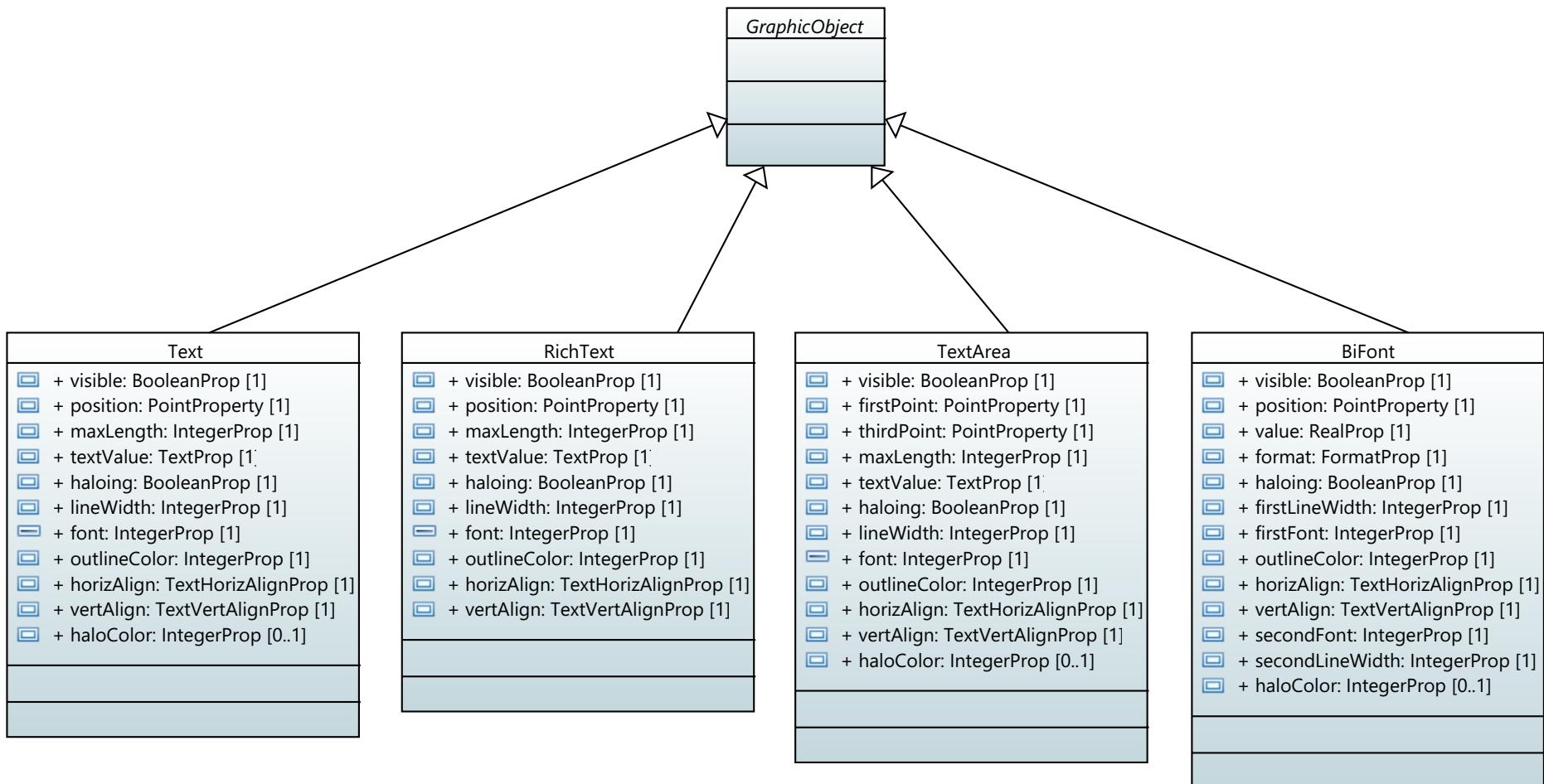
# Variable Dictionary



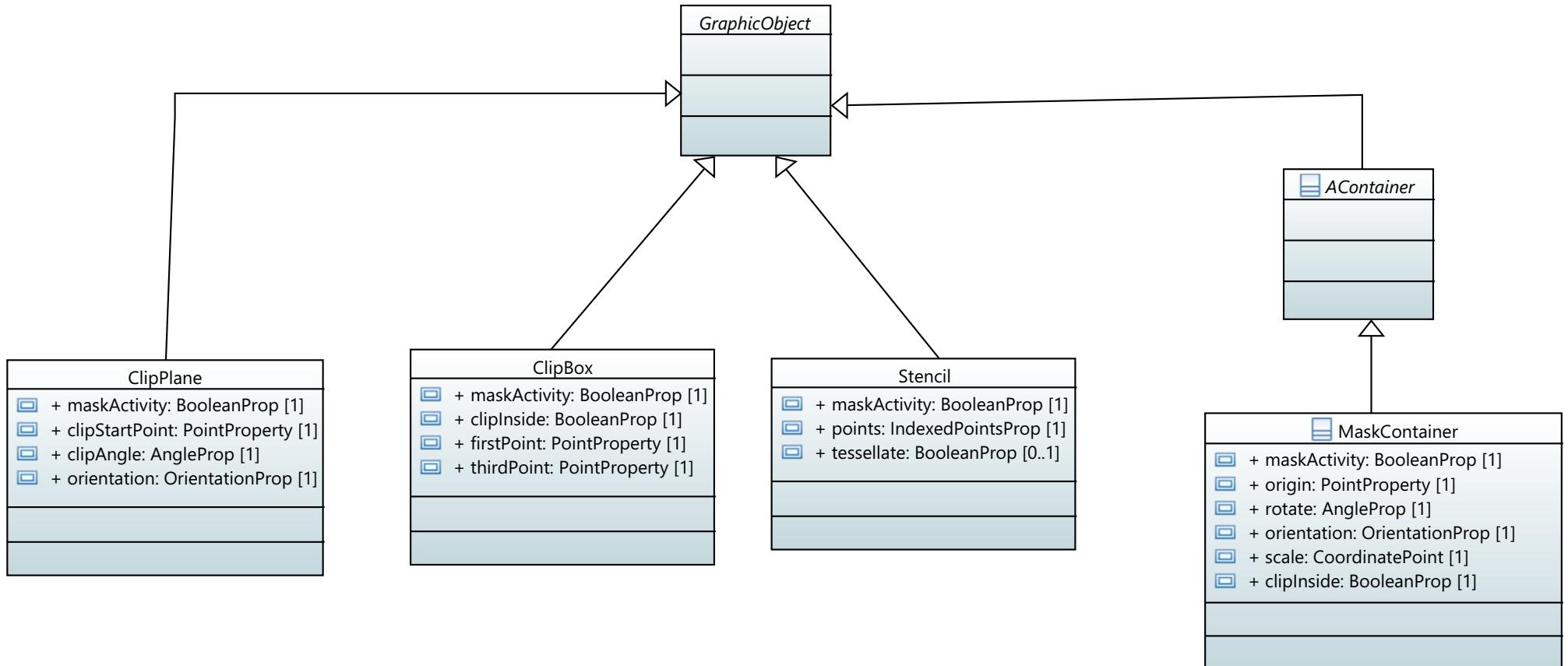
# Graphic Primitives



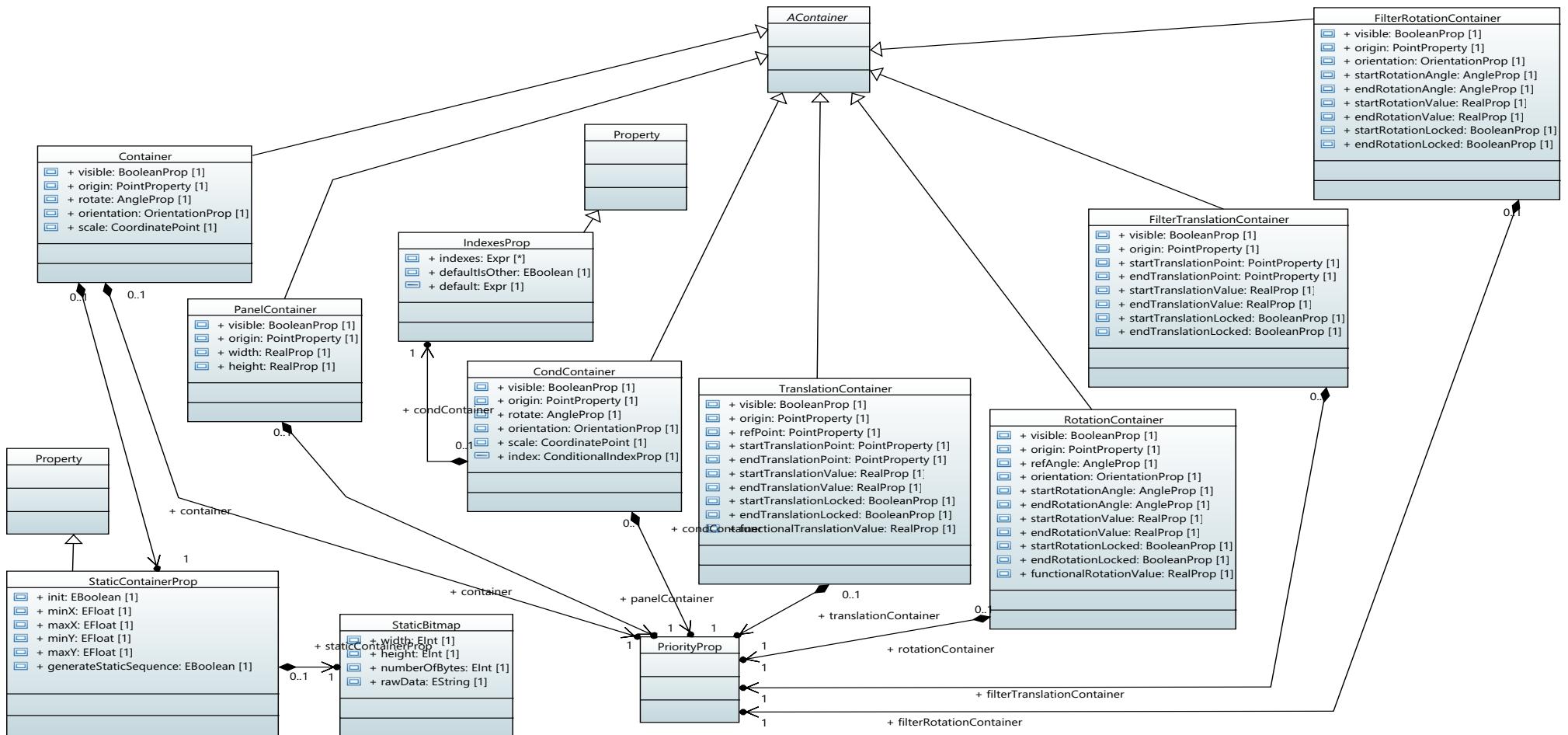
# Text Primitives



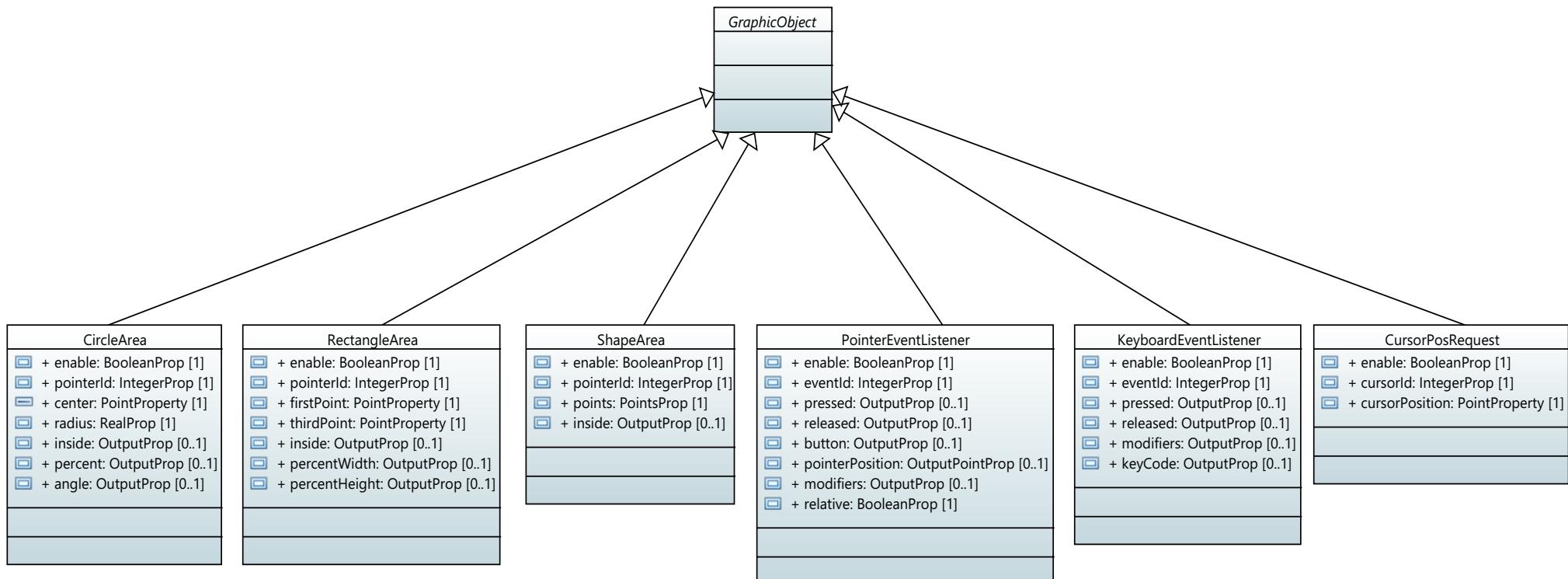
# Masks



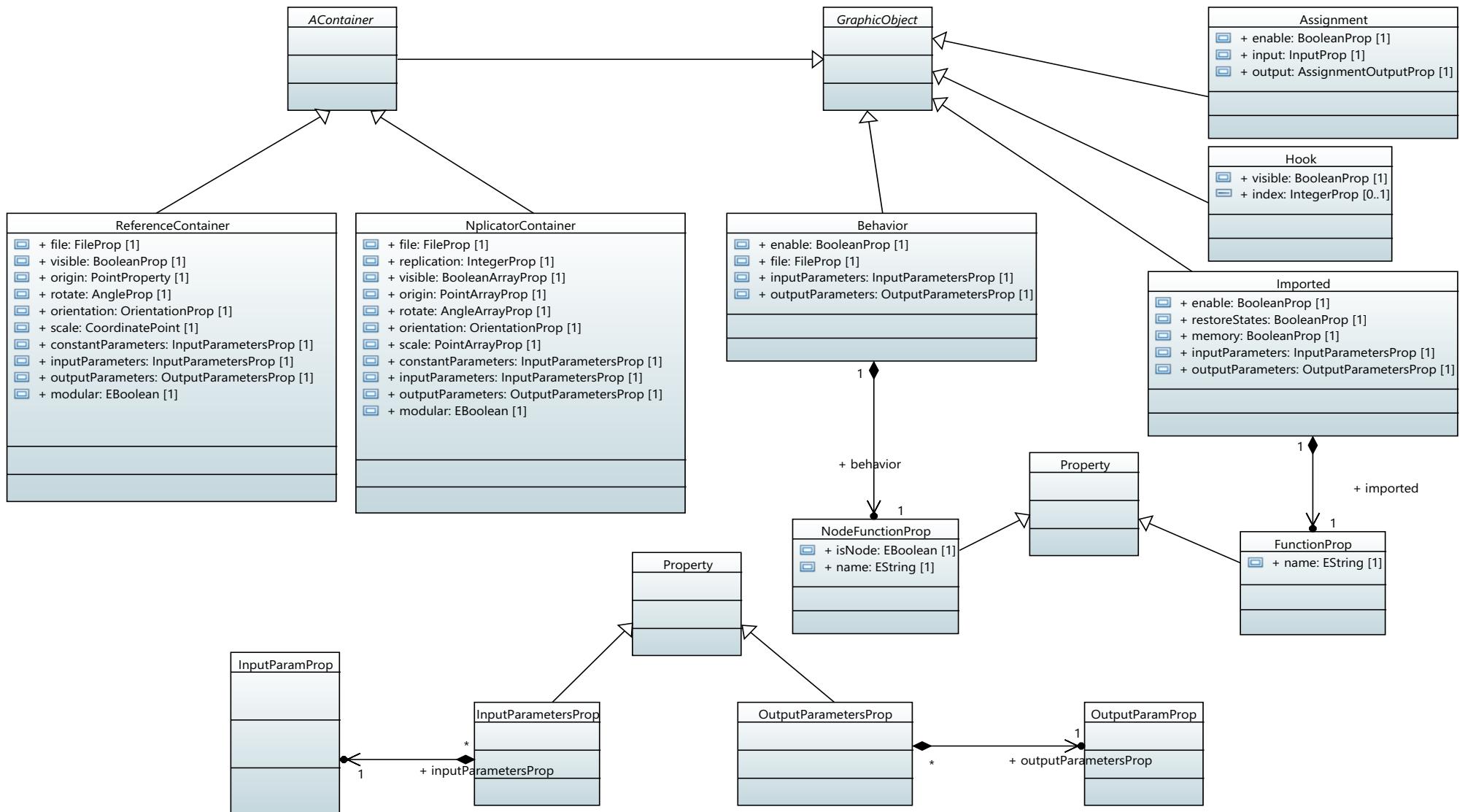
# Containers



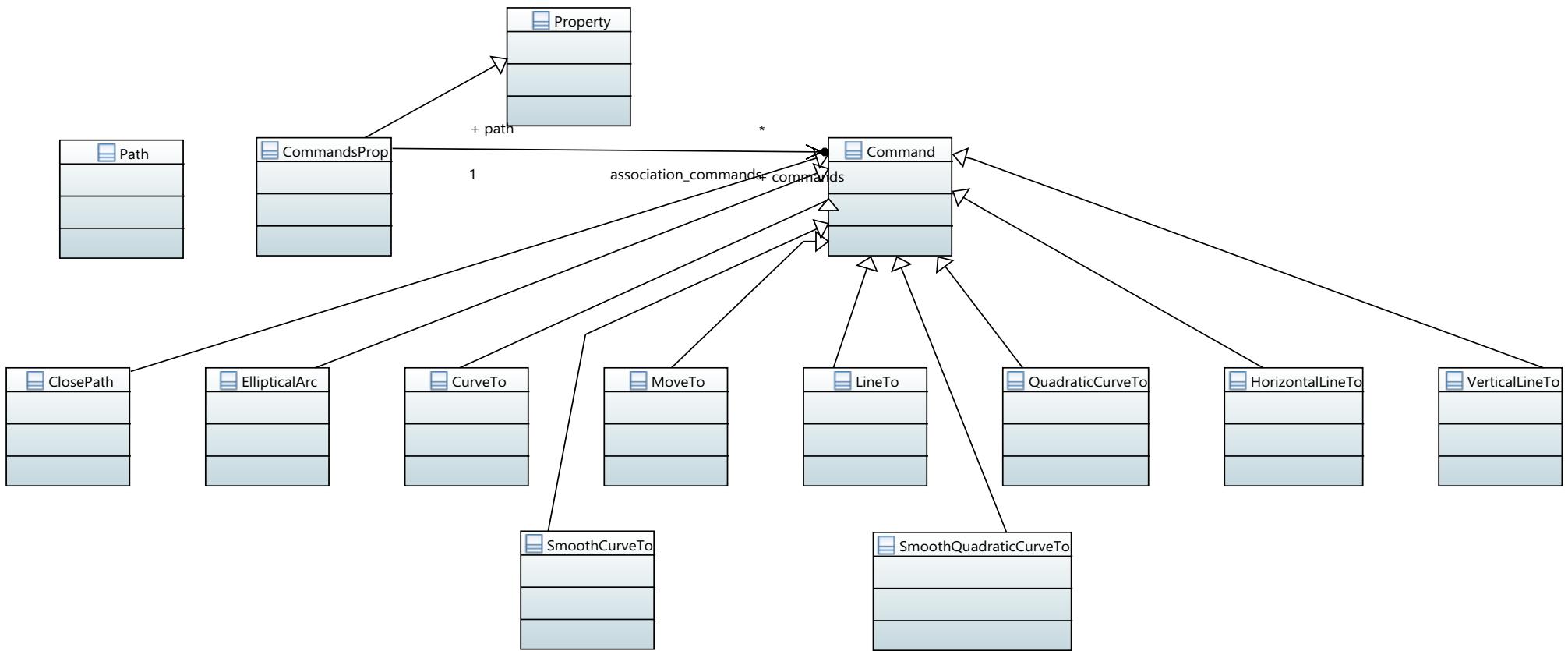
# Interactive Primitives



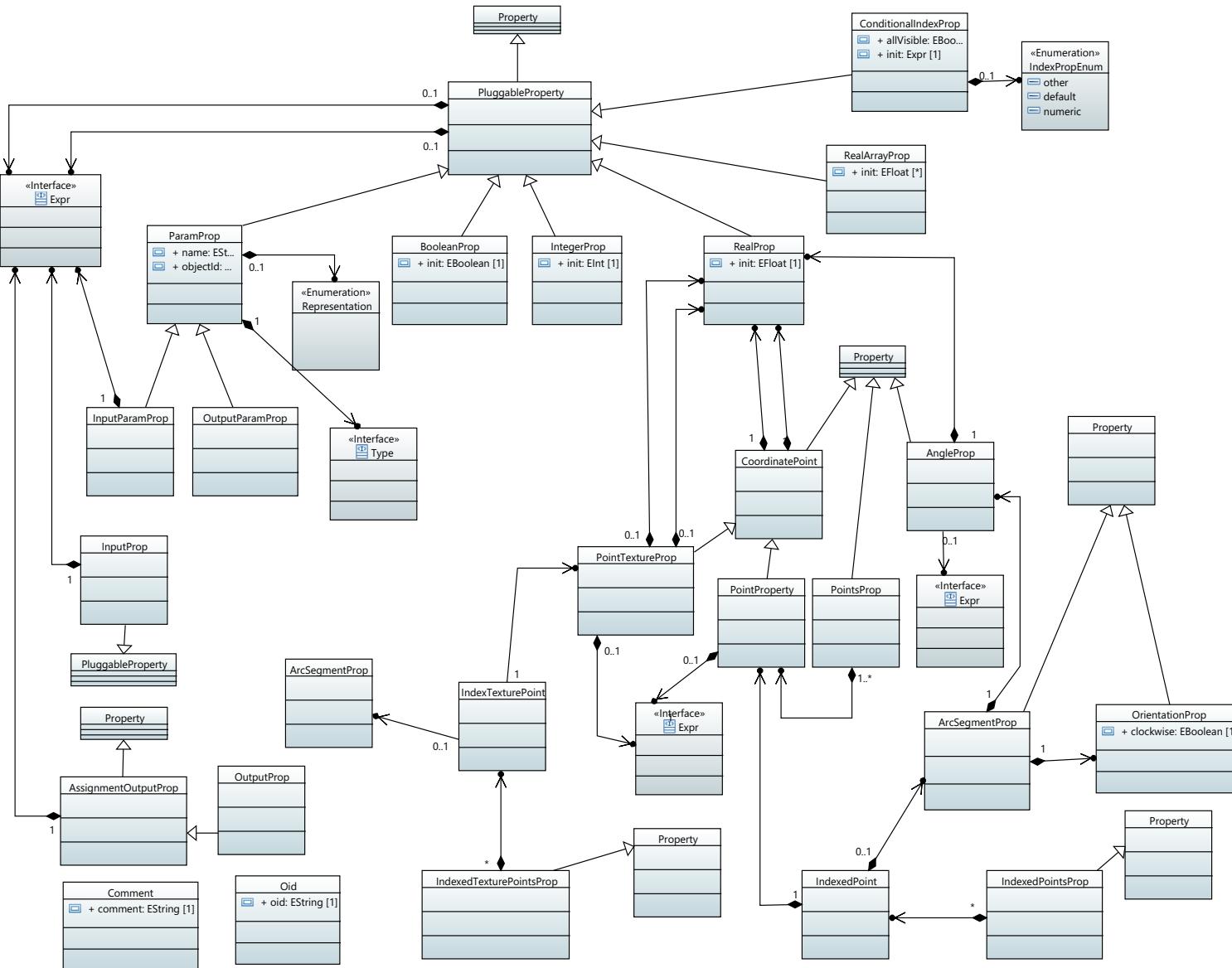
# Reference Primitives



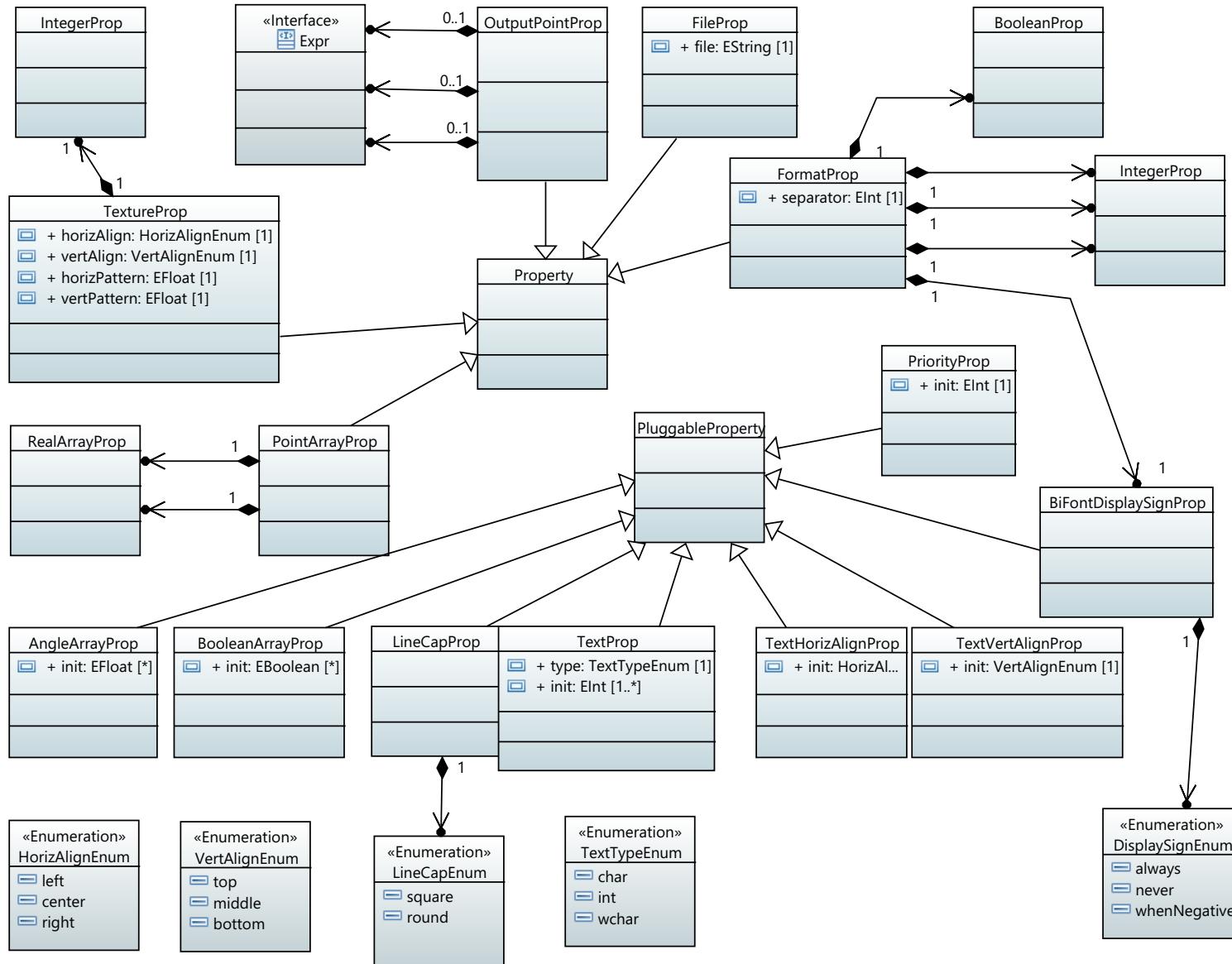
# Path Primitives



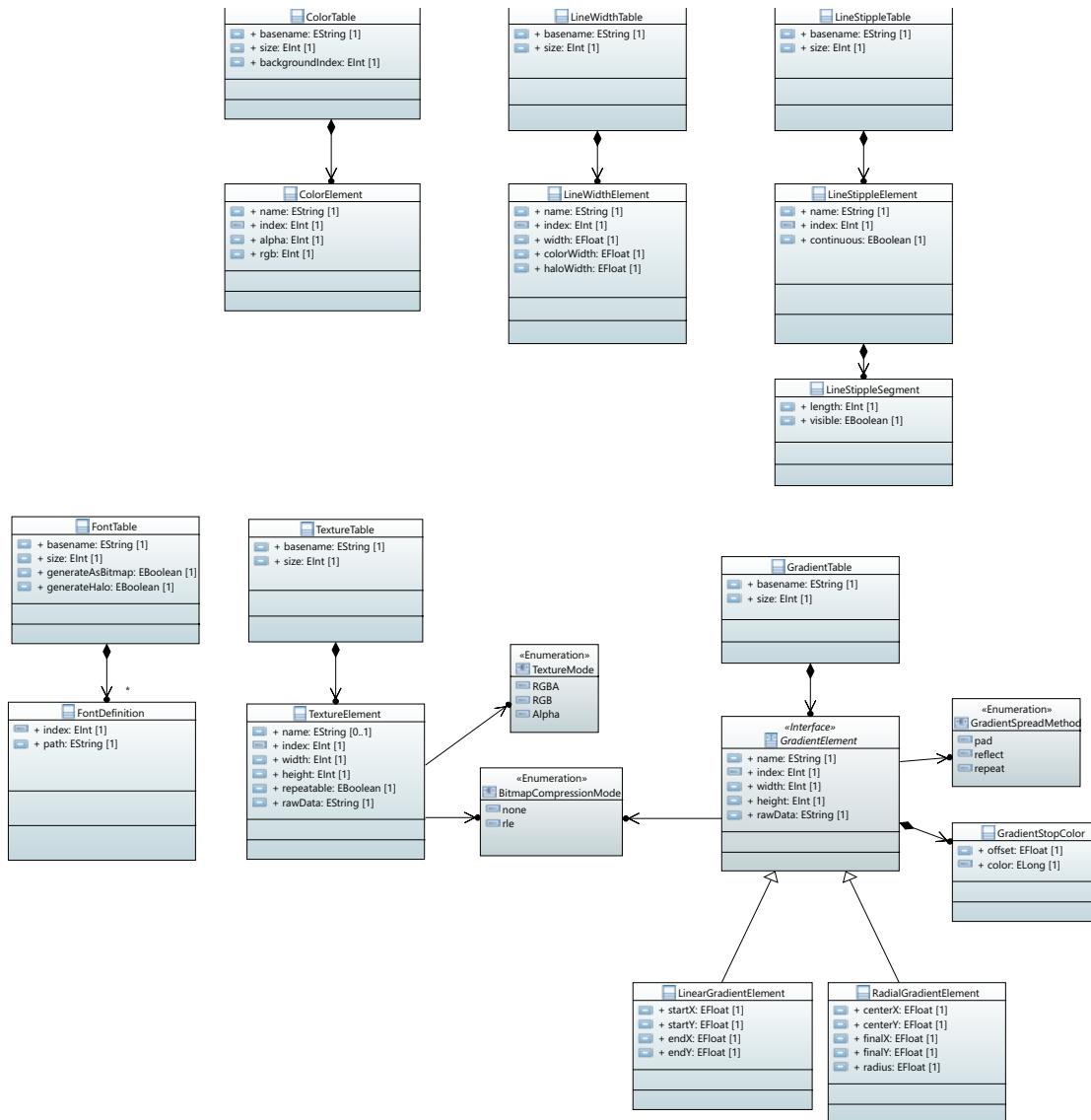
# Properties - 1



## Properties - 2



# Resource Tables

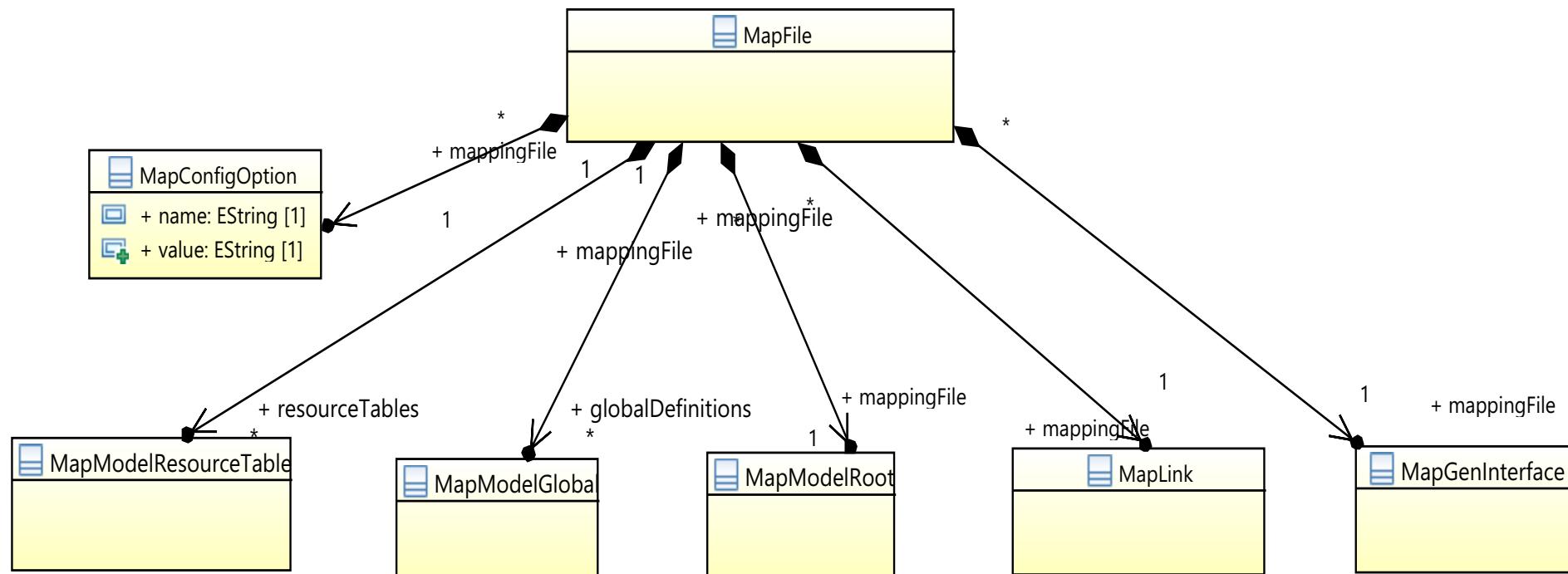


# 21 /SCADE Display Mapping Files Metamodels

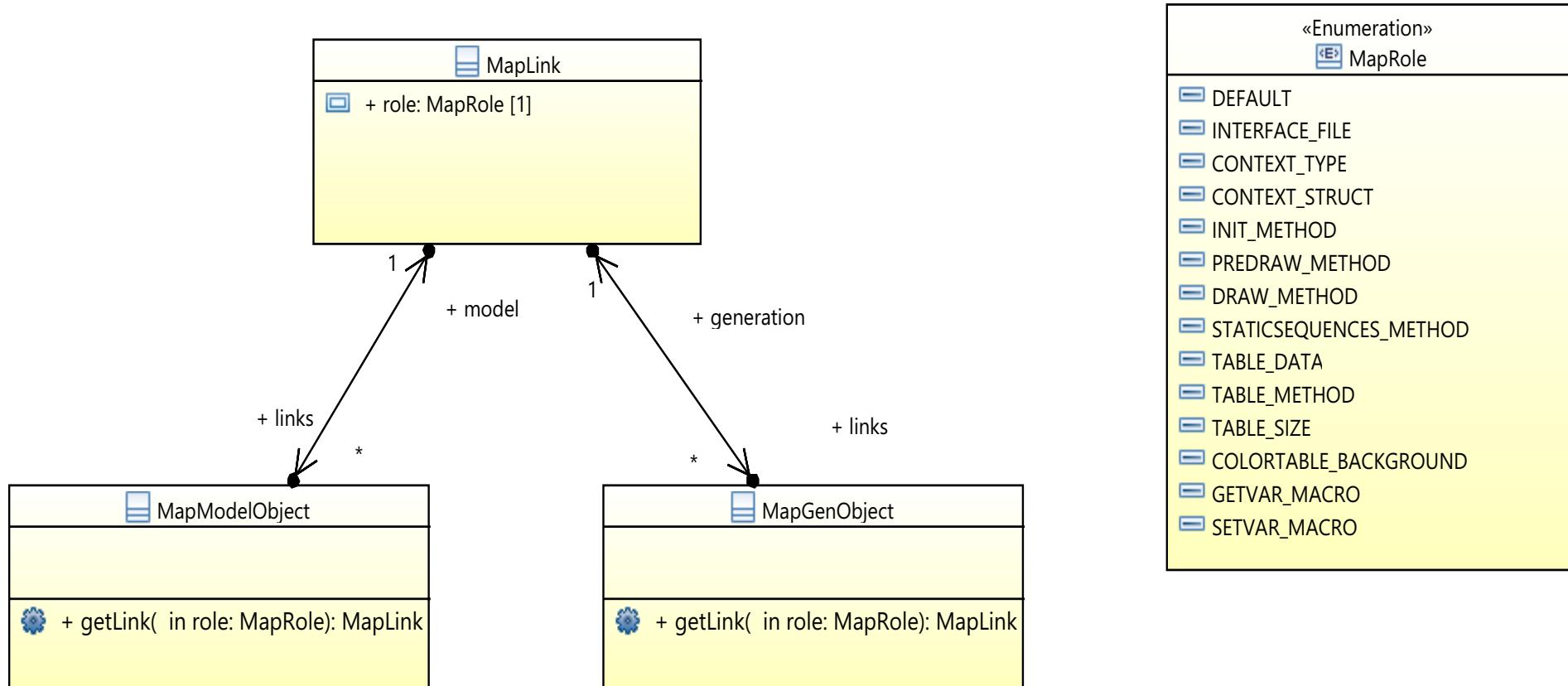
These metamodels present the data structure that give access to generated code through SCADE Display KCG Mapping Files using Python API or Java API:

- [“Mapping File Entry Points”](#)
- [“Mapping Model”](#)
- [“Mapping File Links \(between Generate and Model\)”](#)
- [“Mapping File Generate Model”](#)

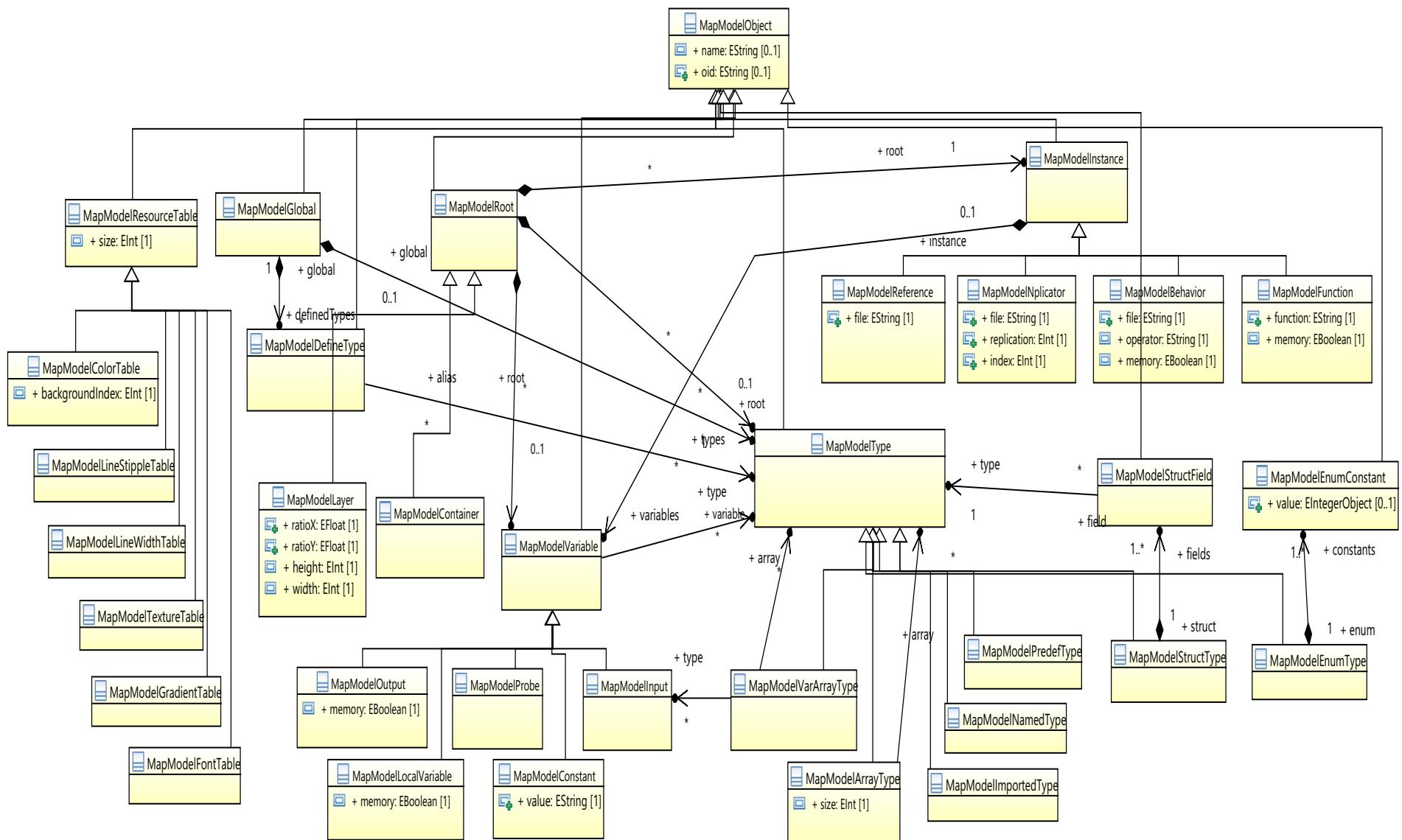
## Mapping File Entry Points



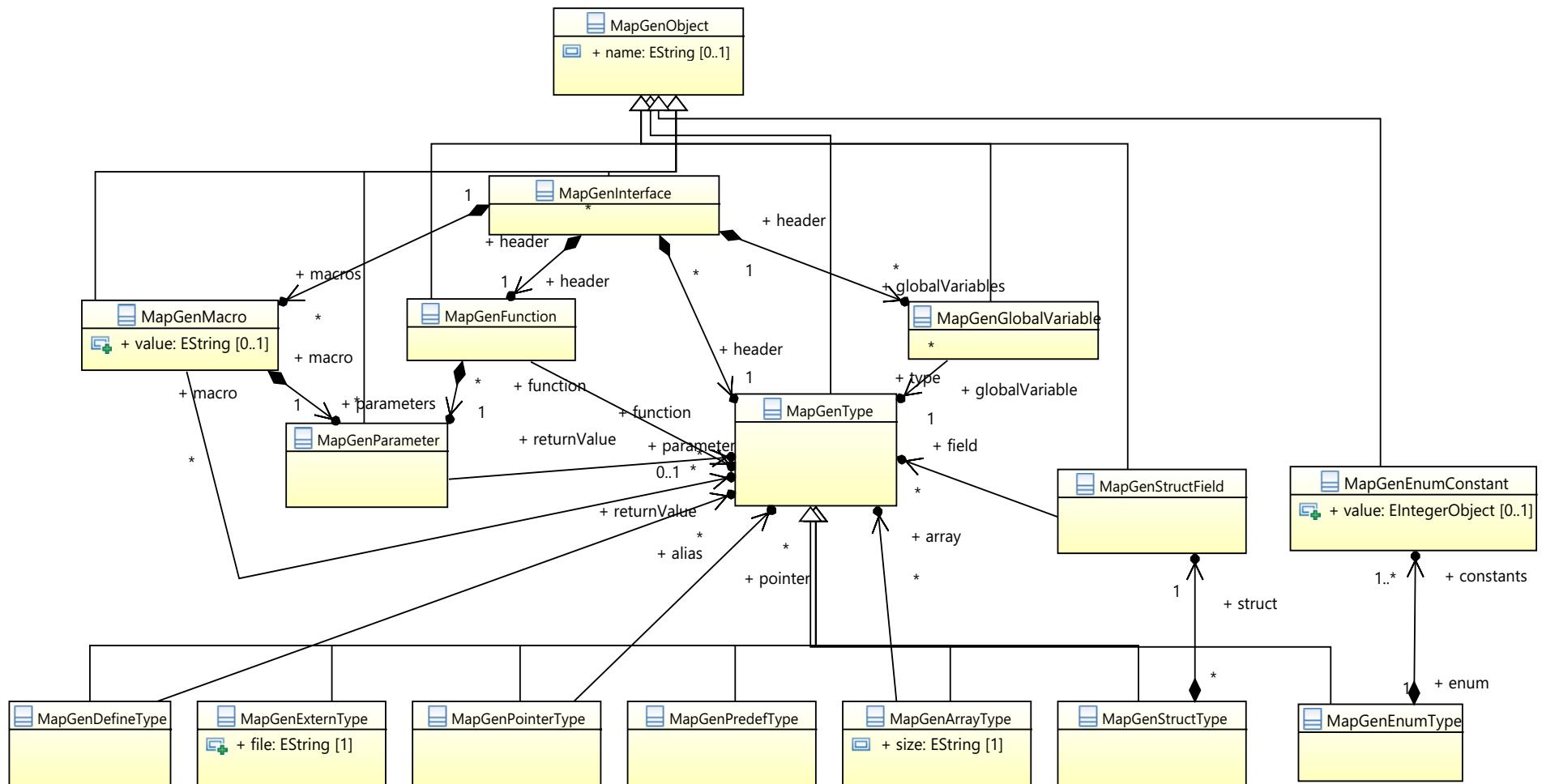
## Mapping File Links (between Generate and Model)



# Mapping Model



# Mapping File Generate Model



## Part 6

# SCADE UA Page Creator Metamodels

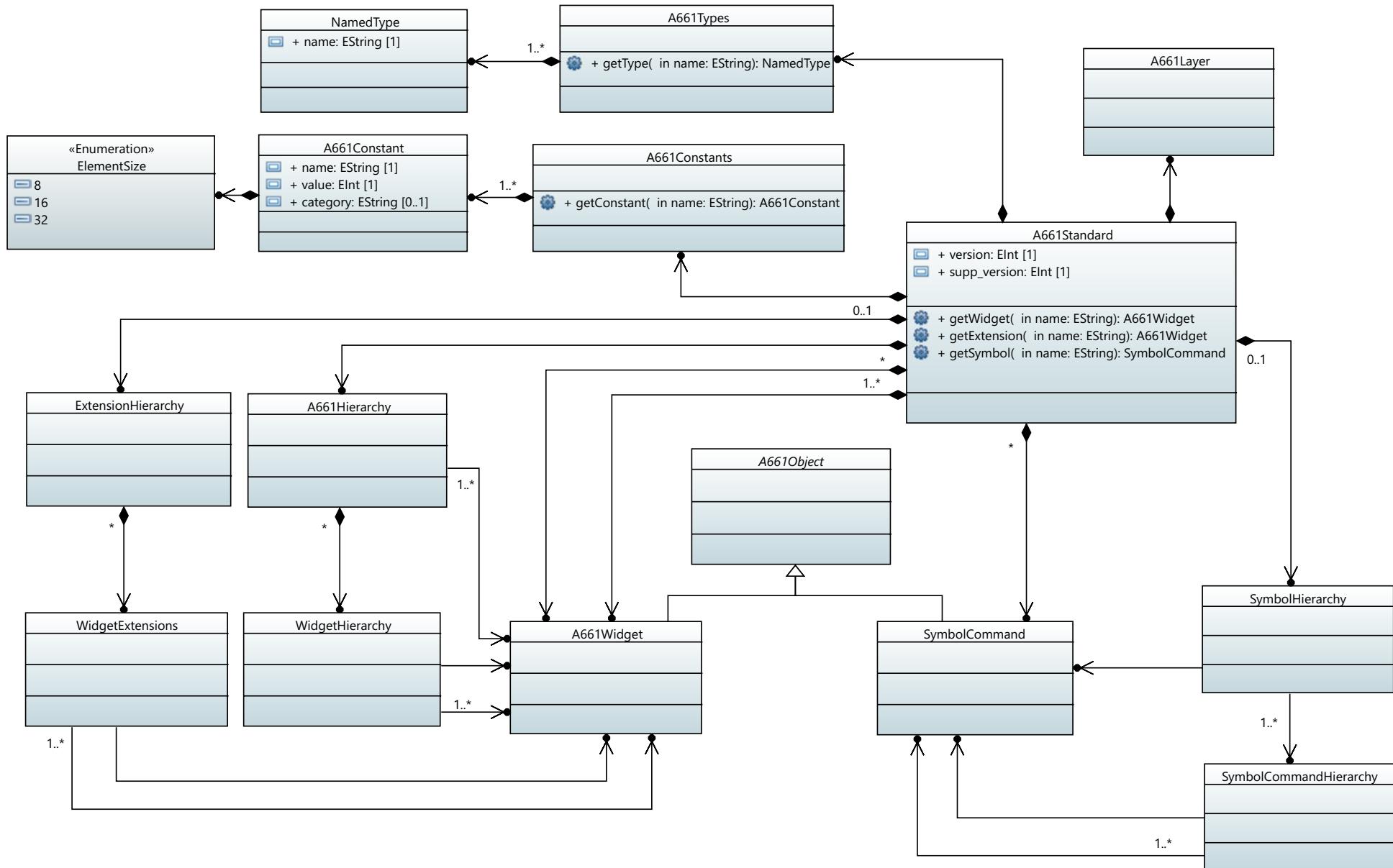
- 22/ [SCADE UA Page Creator Metamodels](#)

## 22 /SCADE UA Page Creator Metamodels

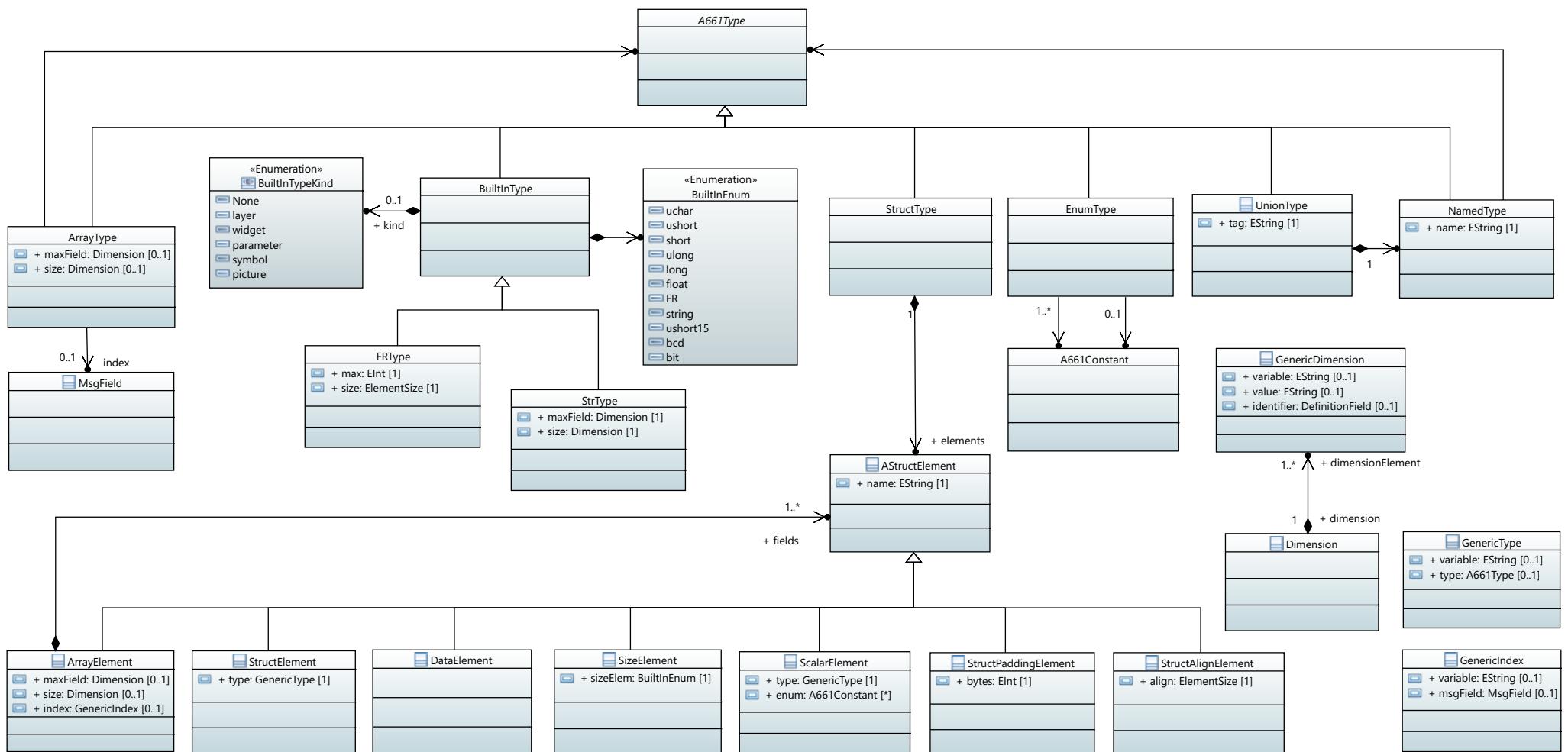
These metamodels present the data structure that give access to SCADE UA Page Creator models using Python API or Java API:

- [“ARINC 661 Standard Model”](#)
- [“A661 Types”](#)
- [“A661 Model”](#)
- [“Widgets”](#)
- [“Widget Set”](#)
- [“Widget Instance”](#)
- [“A661 Properties”](#)
- [“Component and Widget Look Capacities”](#)
- [“Component and Widget Look Definitions”](#)
- [“Picture and Symbol Tables”](#)

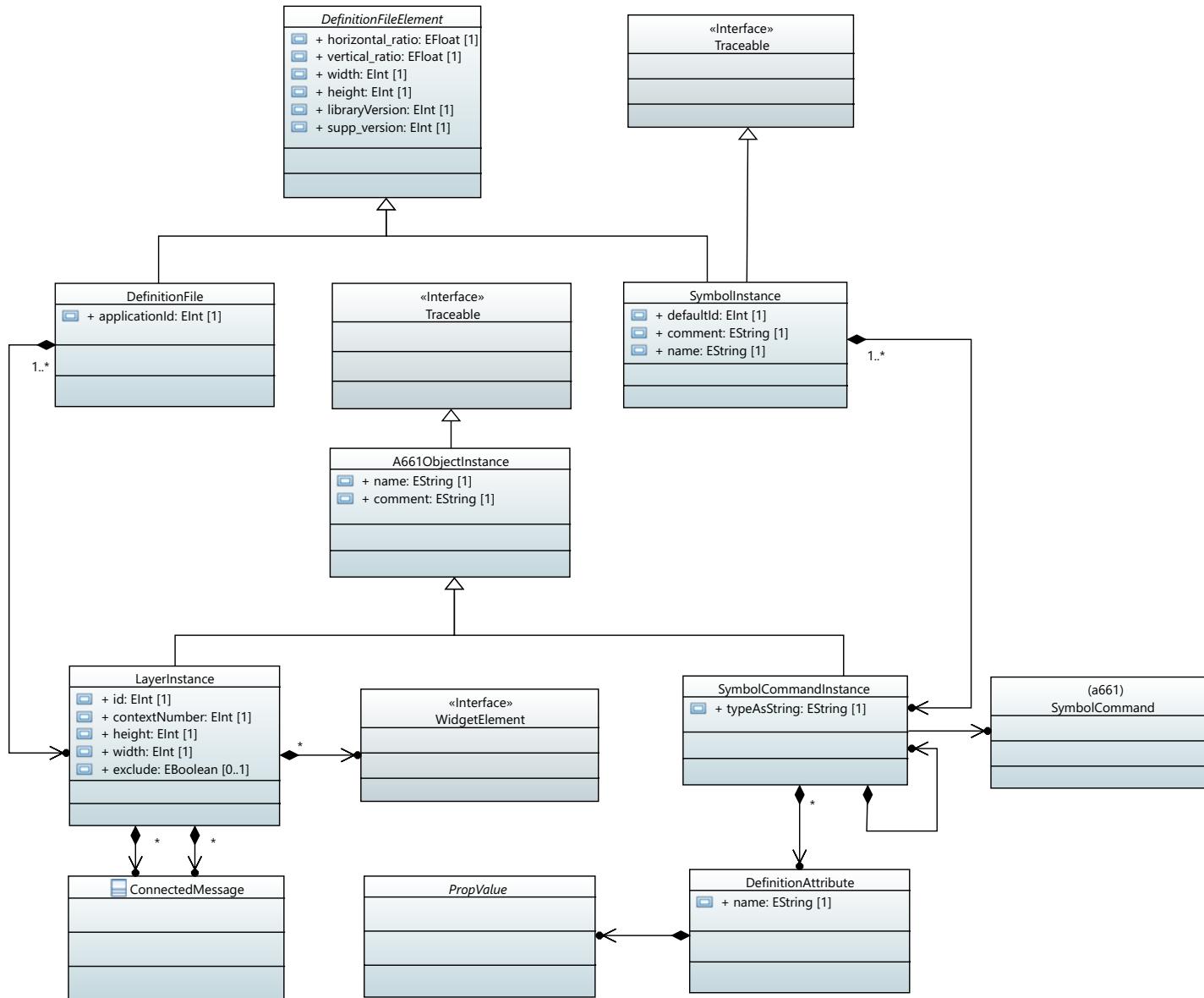
# ARINC 661 Standard Model



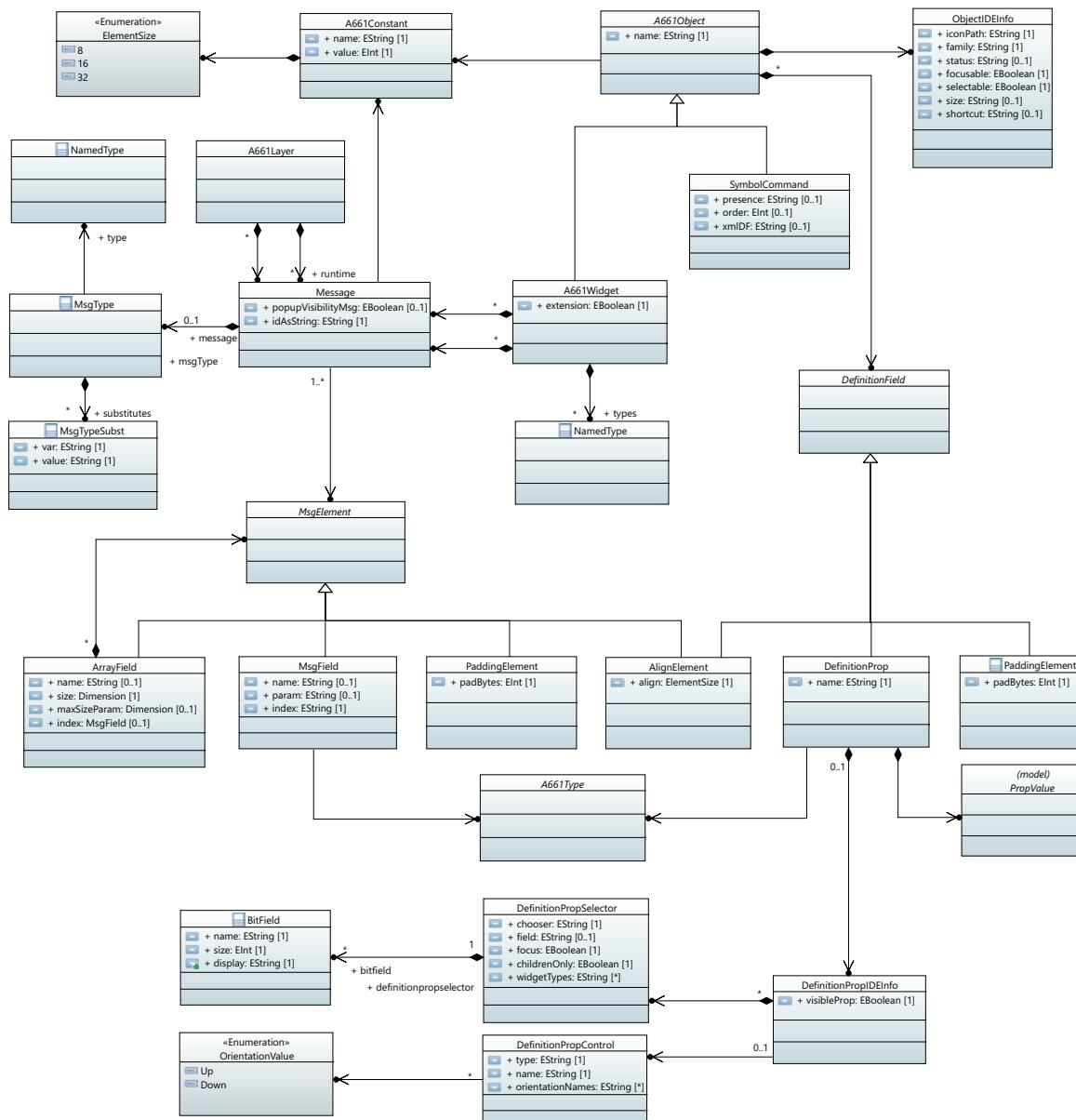
# A661 Types



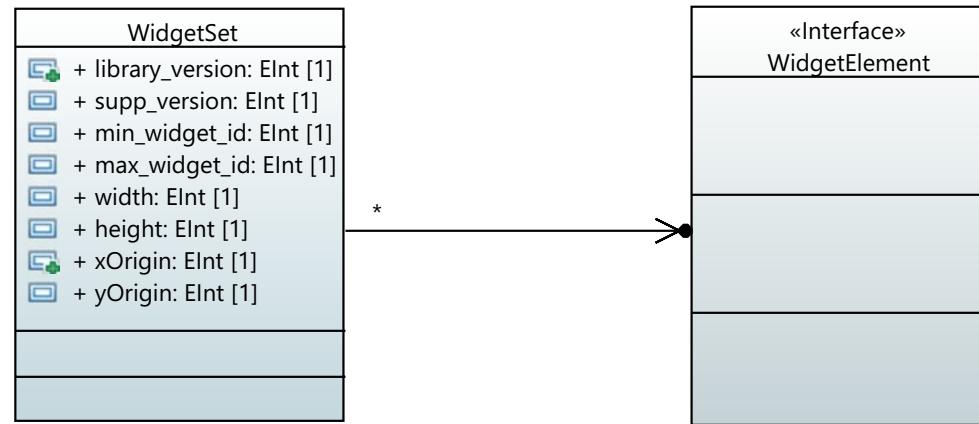
# A661 Model



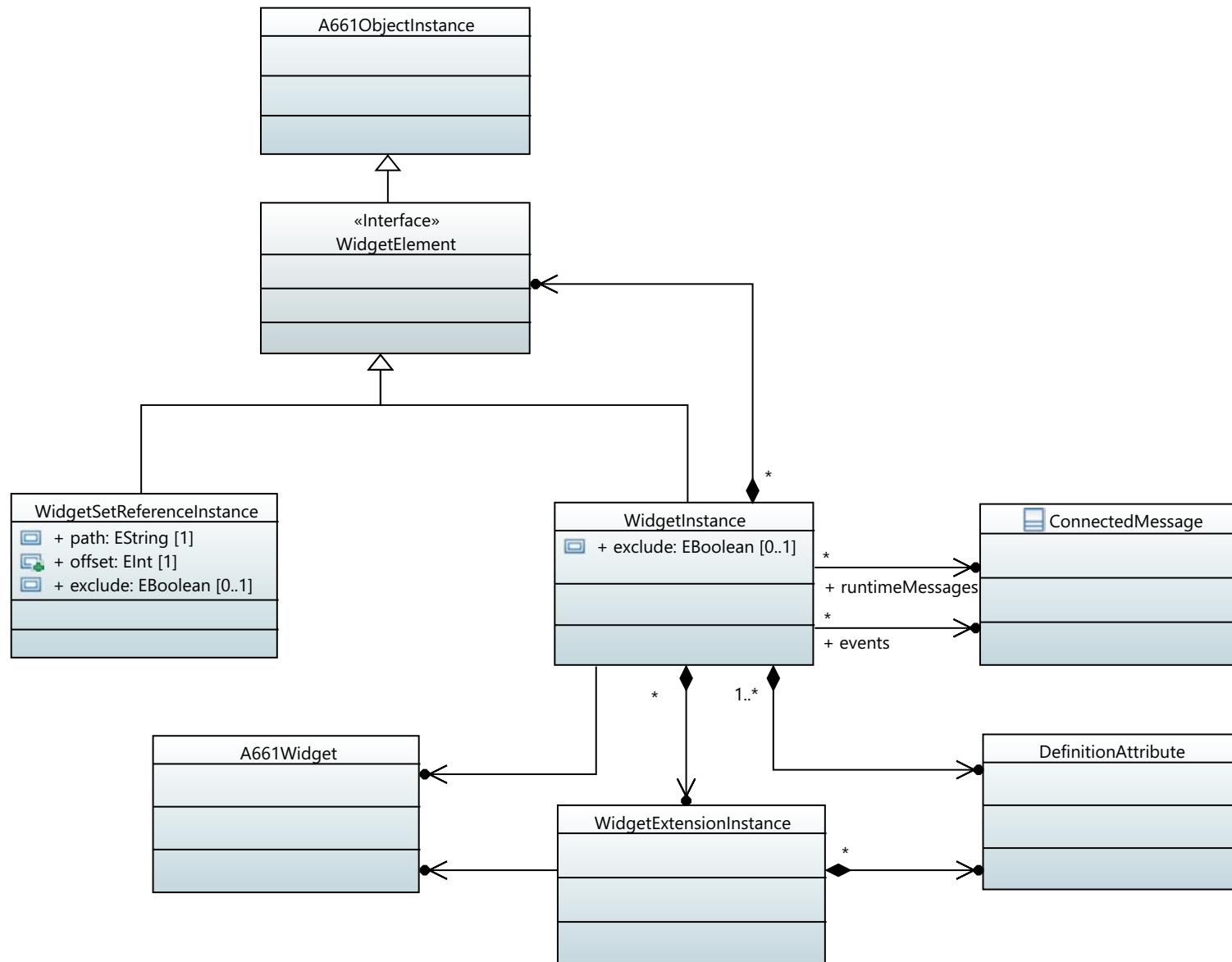
# Widgets



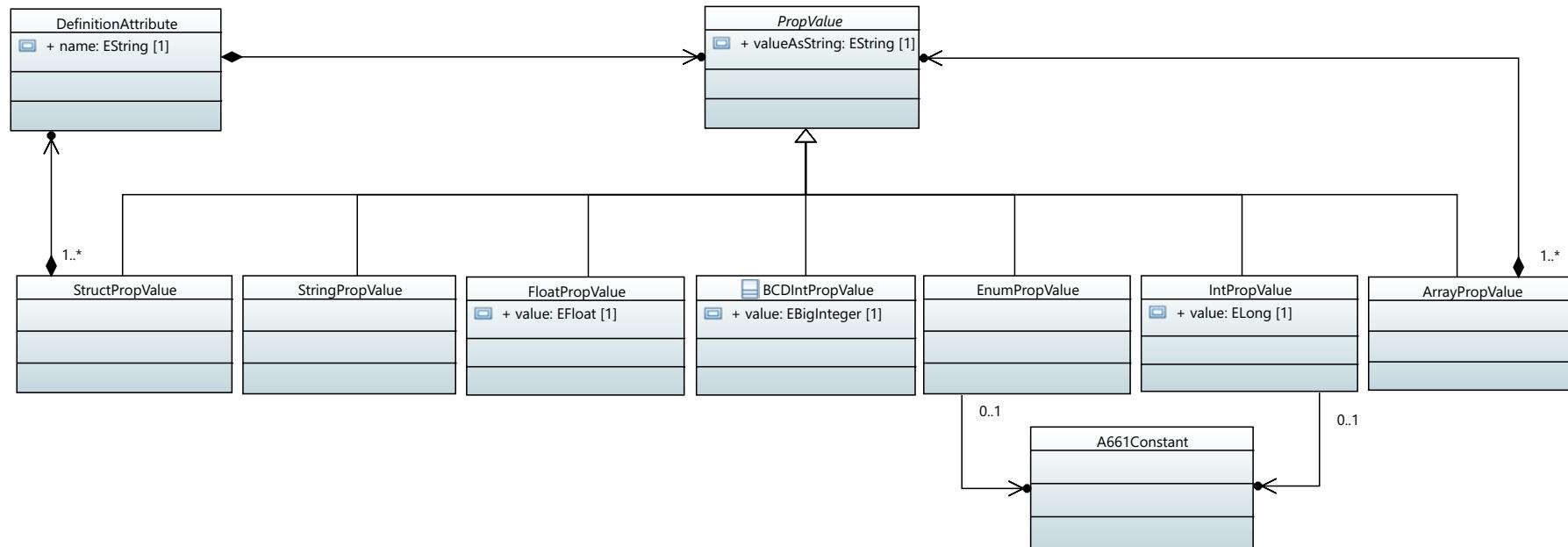
# Widget Set



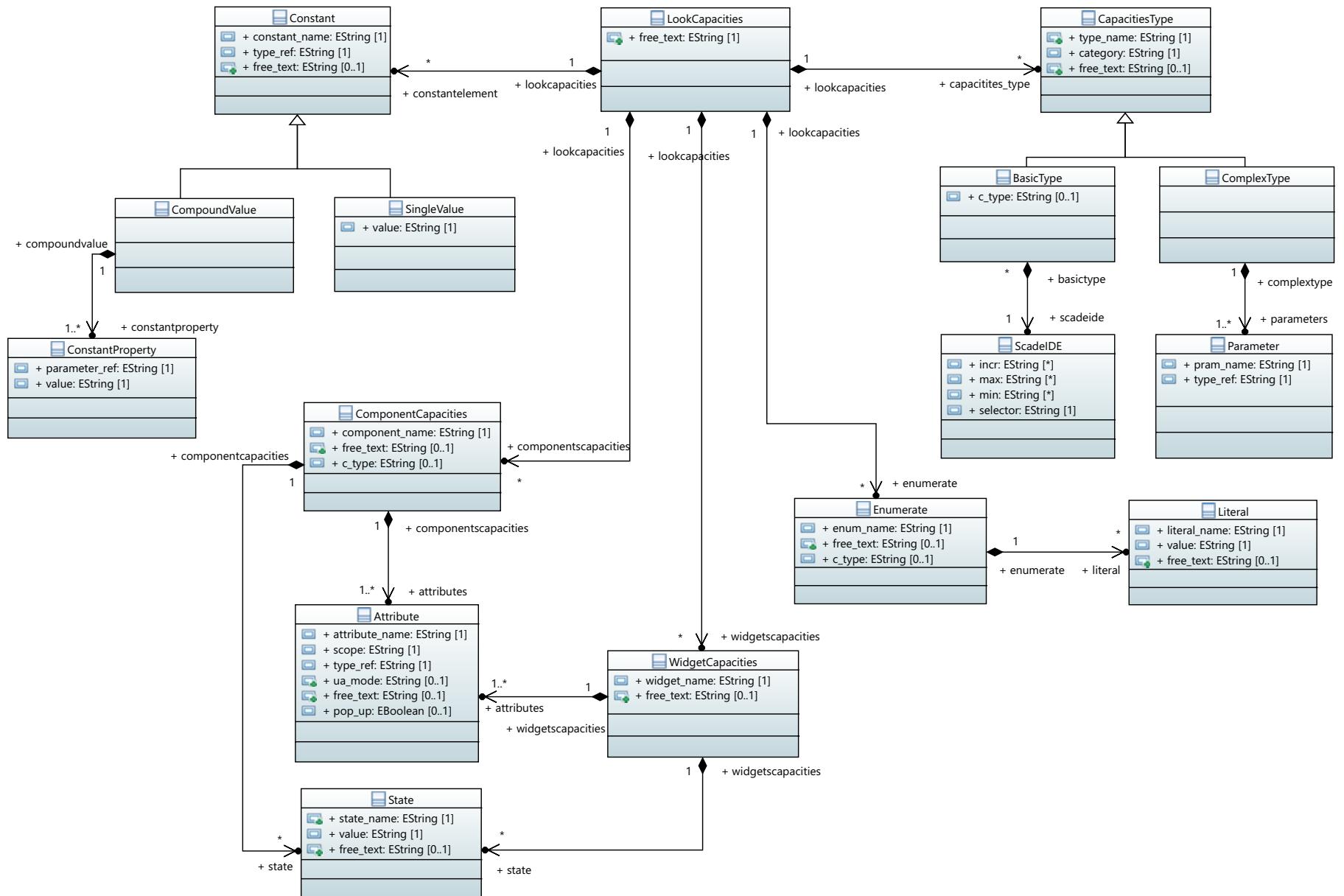
# Widget Instance



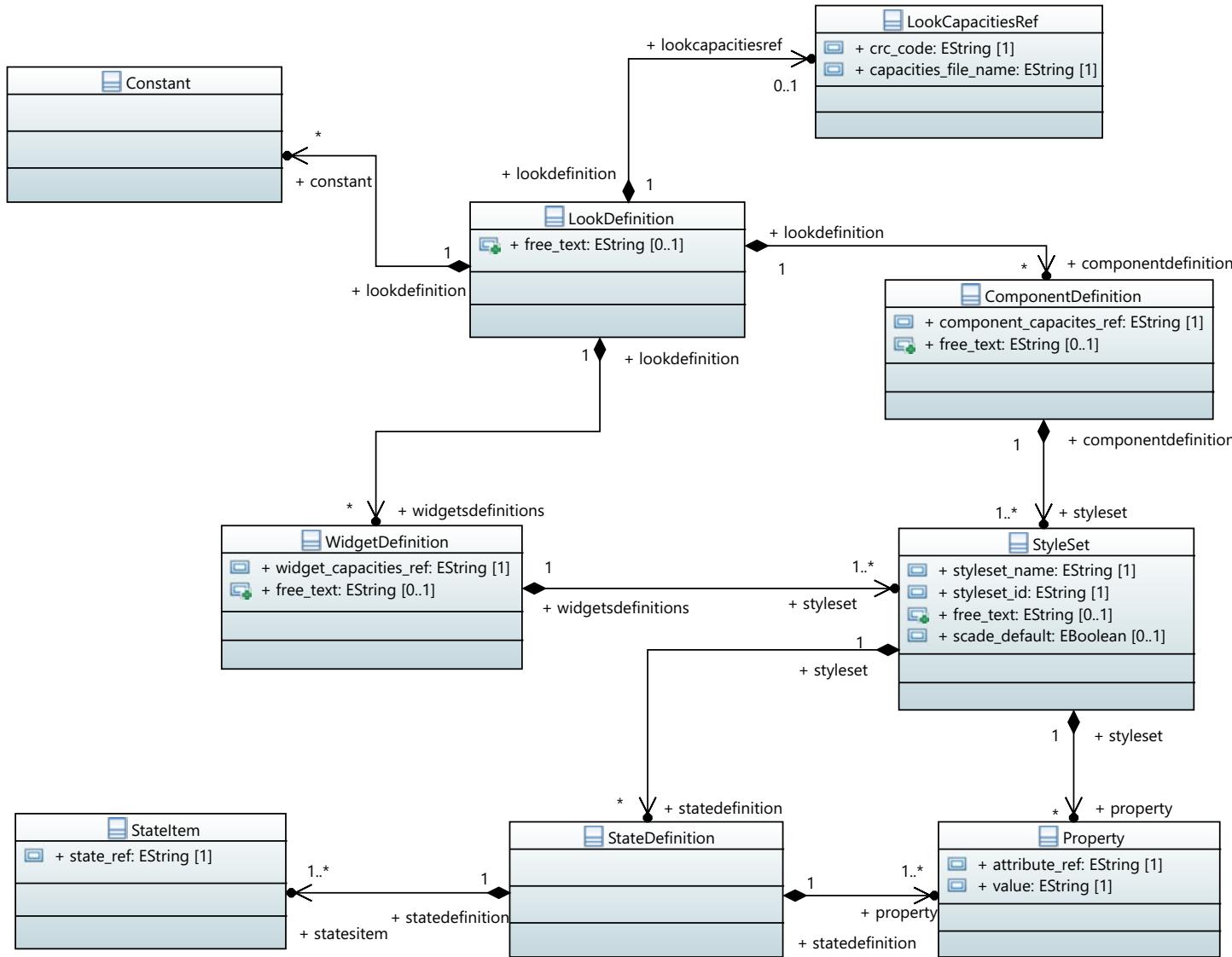
# A661 Properties



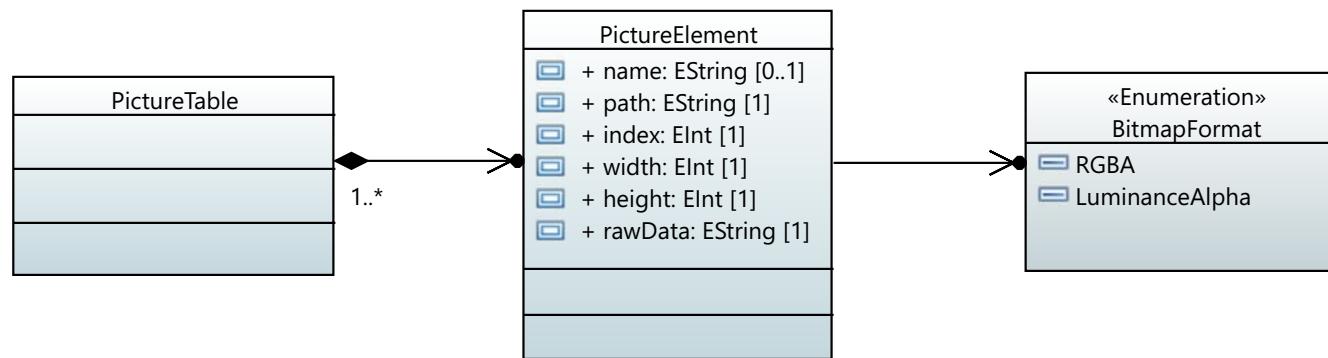
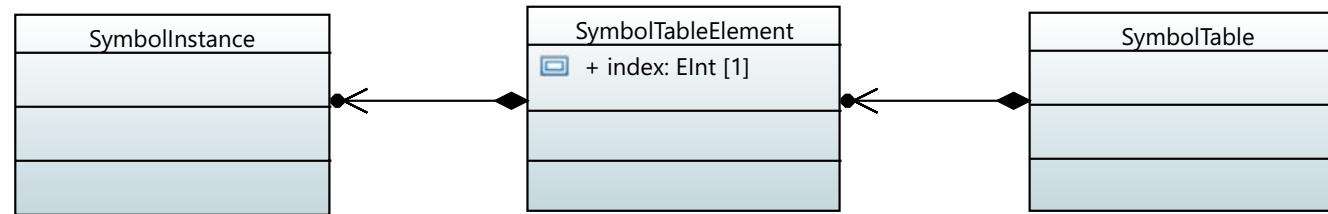
# Component and Widget Look Capacities



# Component and Widget Look Definitions



## Picture and Symbol Tables



## Part 7

# SCADE Test Metamodels

- 23/ ["Test Environment for Host Metamodels \(Tcl\)"](#)
- 24/ ["Test Environment for Host Metamodels \(Python\)"](#)

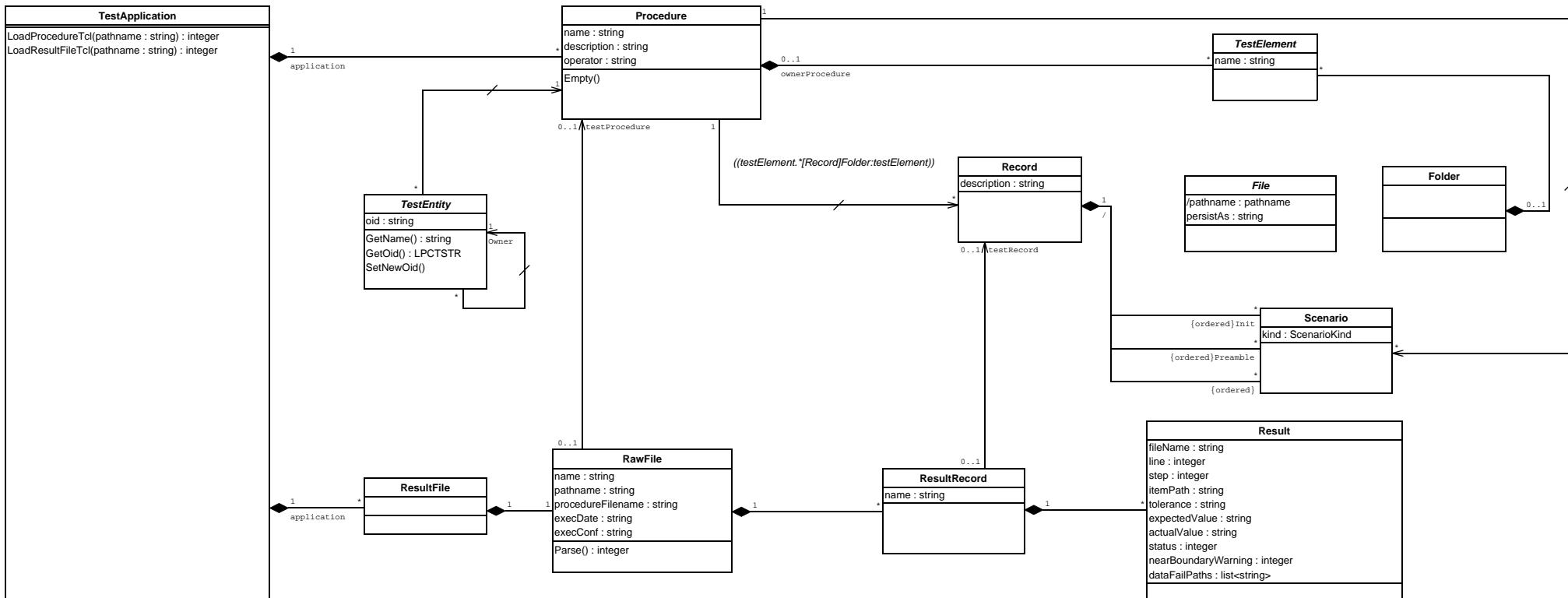
## 23 /Test Environment for Host Metamodels (Tcl)

These metamodels present the data structures that give access to SCADE Test Environment for Host using Tcl API or Java API:

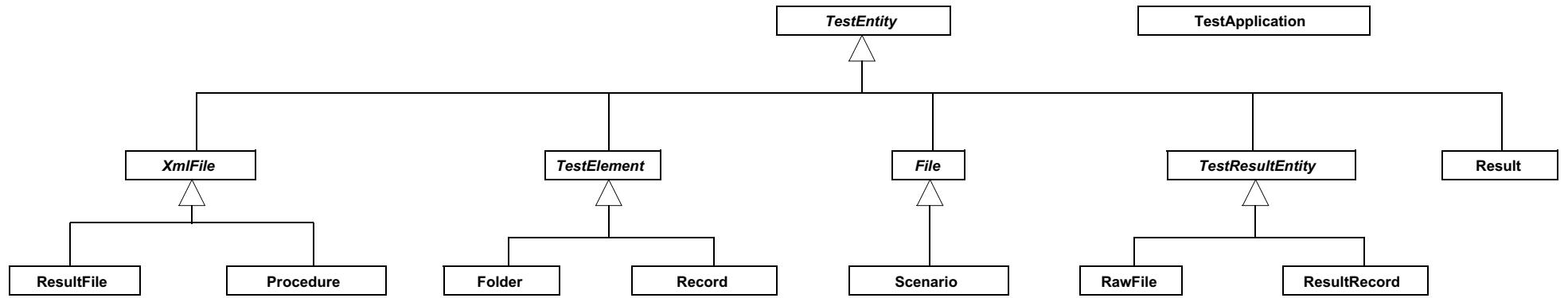
For SCADE Suite model testing:

- [“Testing Environment Core \(Tcl\)”](#)
- [“Testing Environment Inheritance \(Tcl\)”](#)
- [“Testing Environment Procedure File \(Tcl, Java\)”](#)
- [“Testing Environment Result File \(Tcl\)”](#)

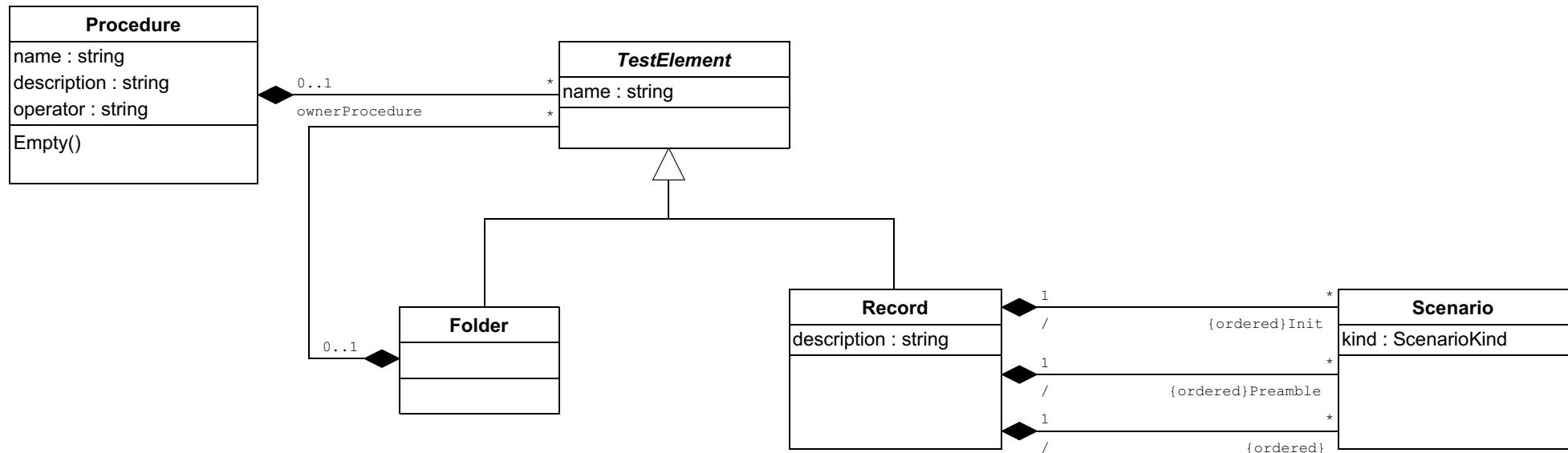
# Testing Environment Core (Tcl)



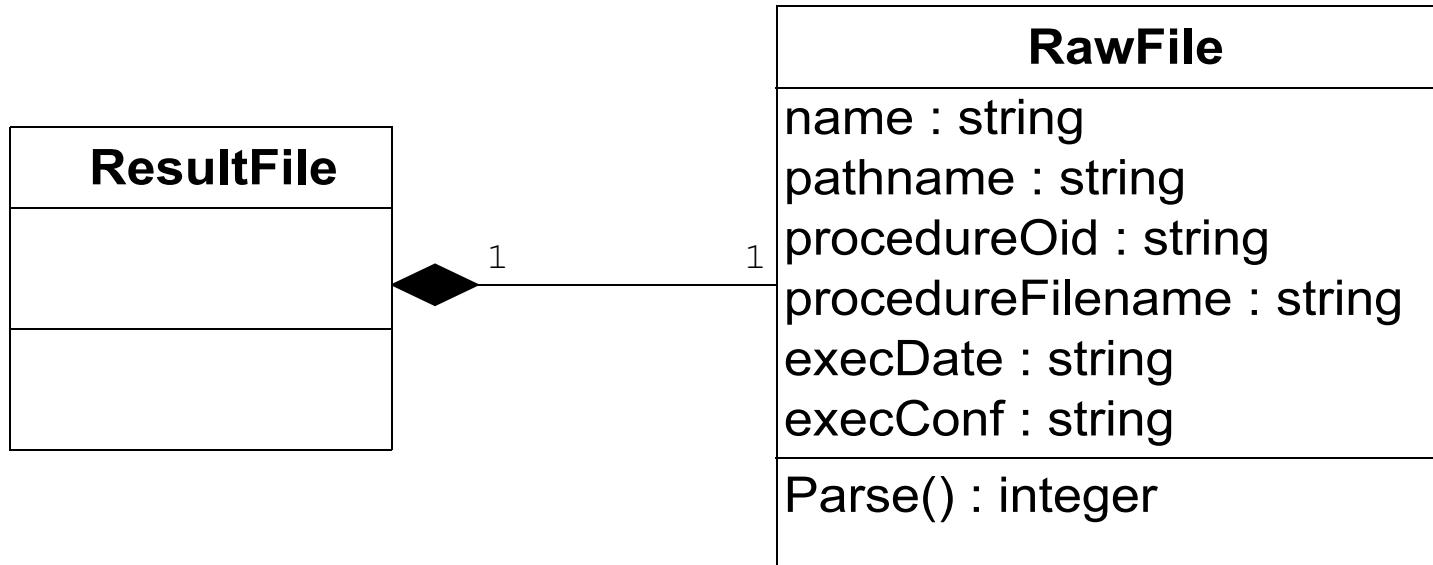
# Testing Environment Inheritance (Tcl)



# Testing Environment Procedure File (Tcl, Java)



## Testing Environment Result File (Tcl)



## 24 /Test Environment for Host Metamodels (Python)

These metamodels present the data structures that give access to SCADE Test Environment for Host using Python API:

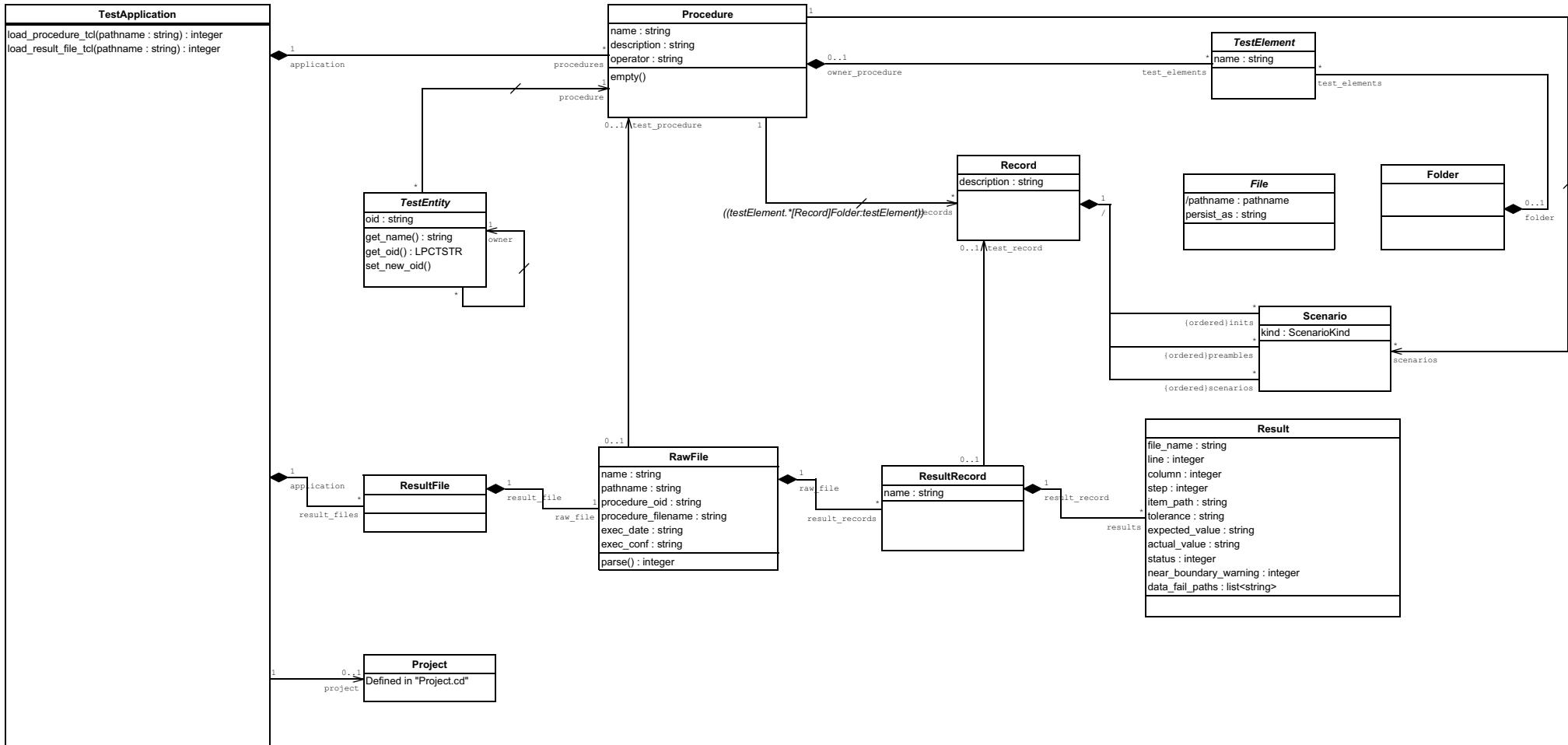
For SCADE Suite model testing:

- [“Testing Environment Core \(Python\)”](#)
- [“Testing Environment Inheritance \(Python\)”](#)
- [“Testing Environment Procedure File \(Python\)”](#)
- [“Testing Environment Result File \(Python\)”](#)

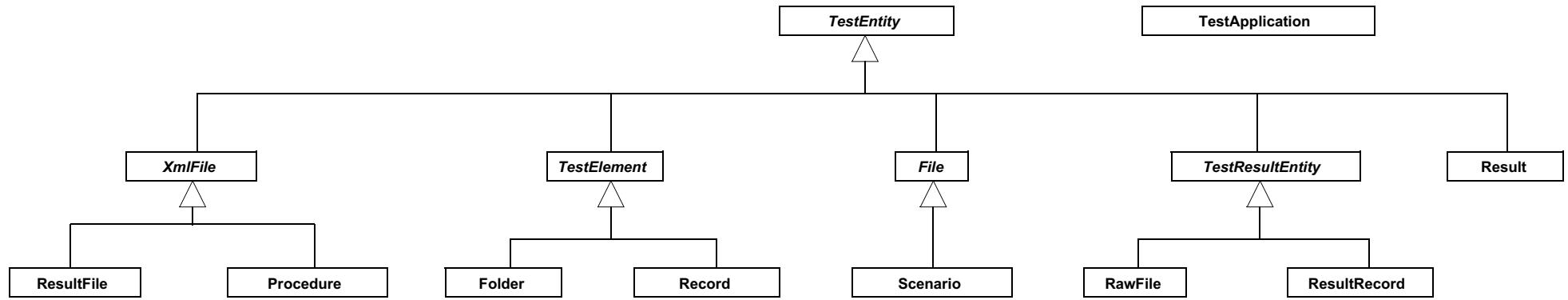
For SCADE Display model testing:

- [“Scenario Classes \(Python, Java\)”](#)

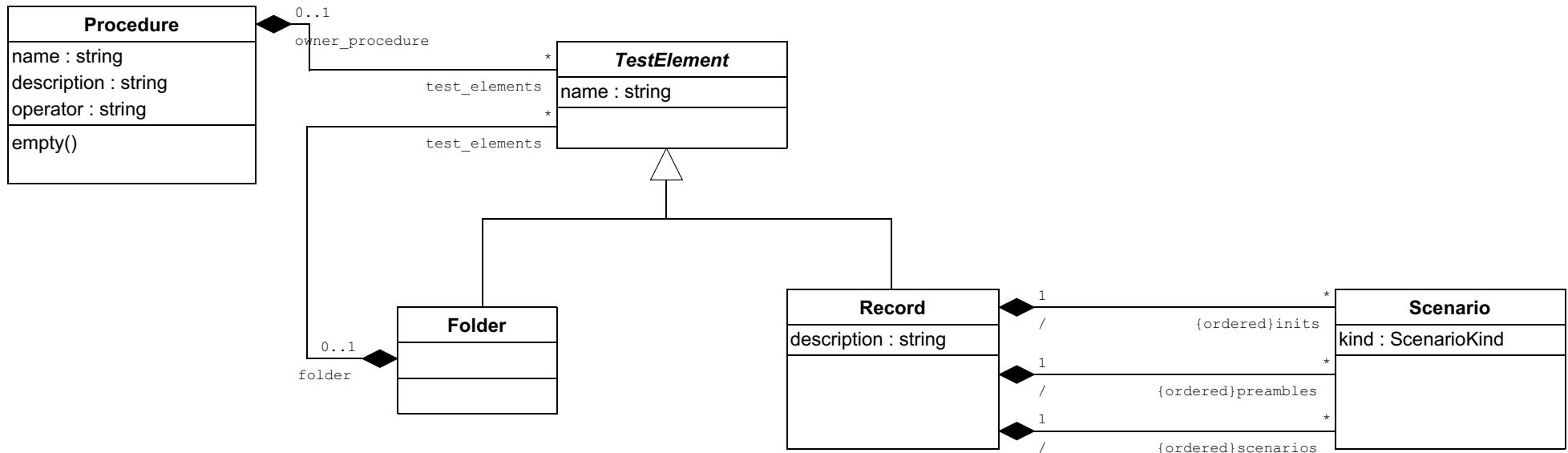
# Testing Environment Core (Python)



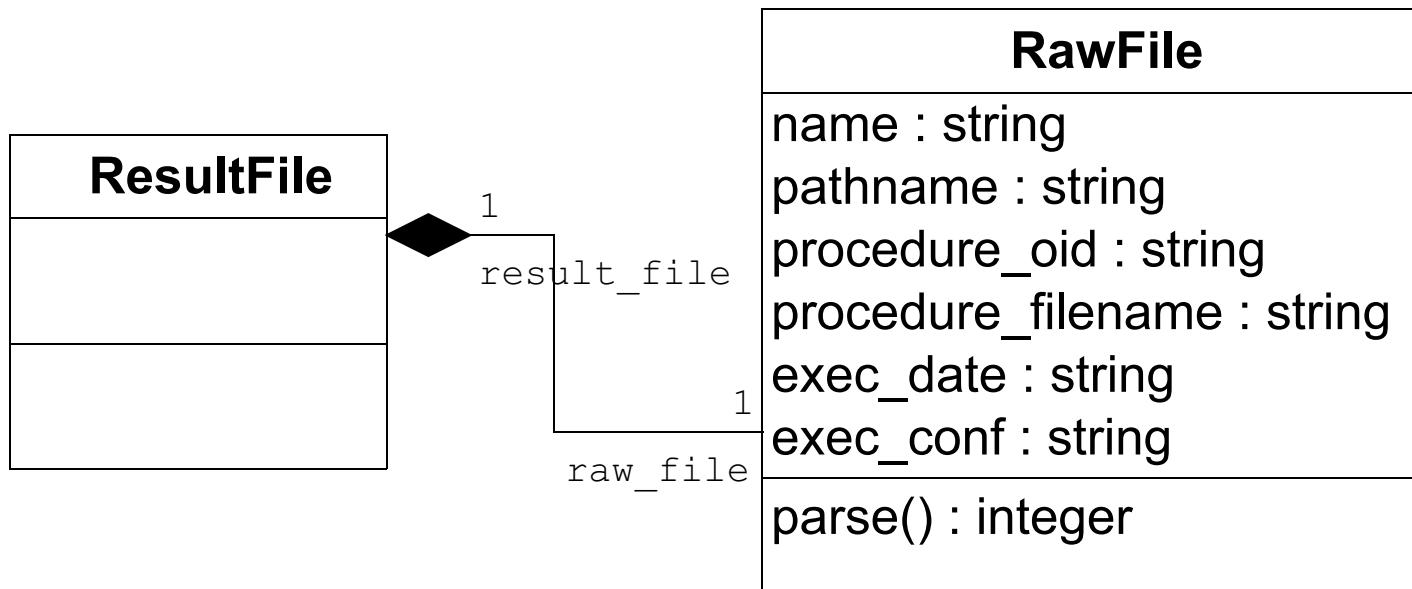
# Testing Environment Inheritance (Python)



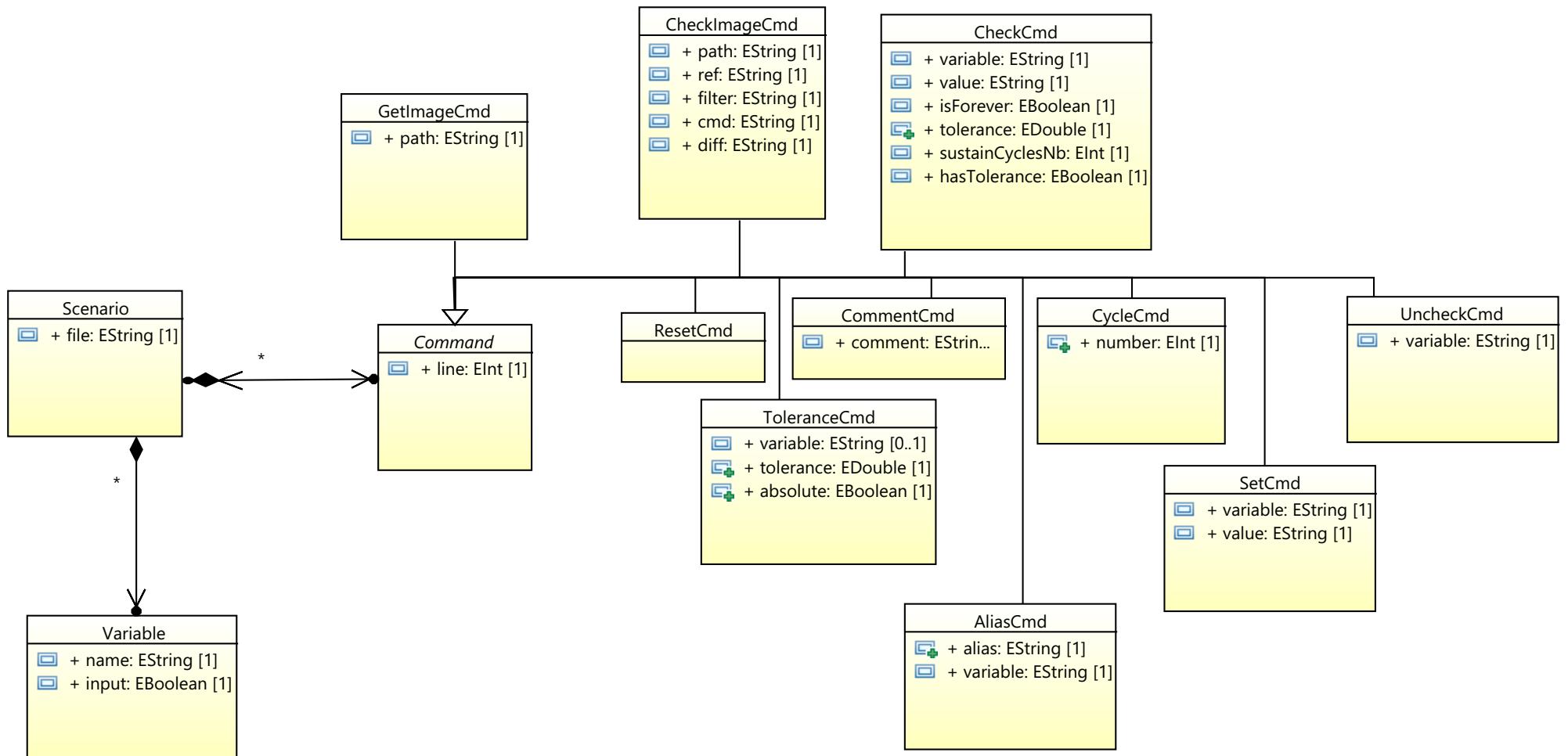
# Testing Environment Procedure File (Python)



## Testing Environment Result File (Python)



## Scenario Classes (Python, Java)



## Part 8

# SCADE Traceability Metamodels for Python API

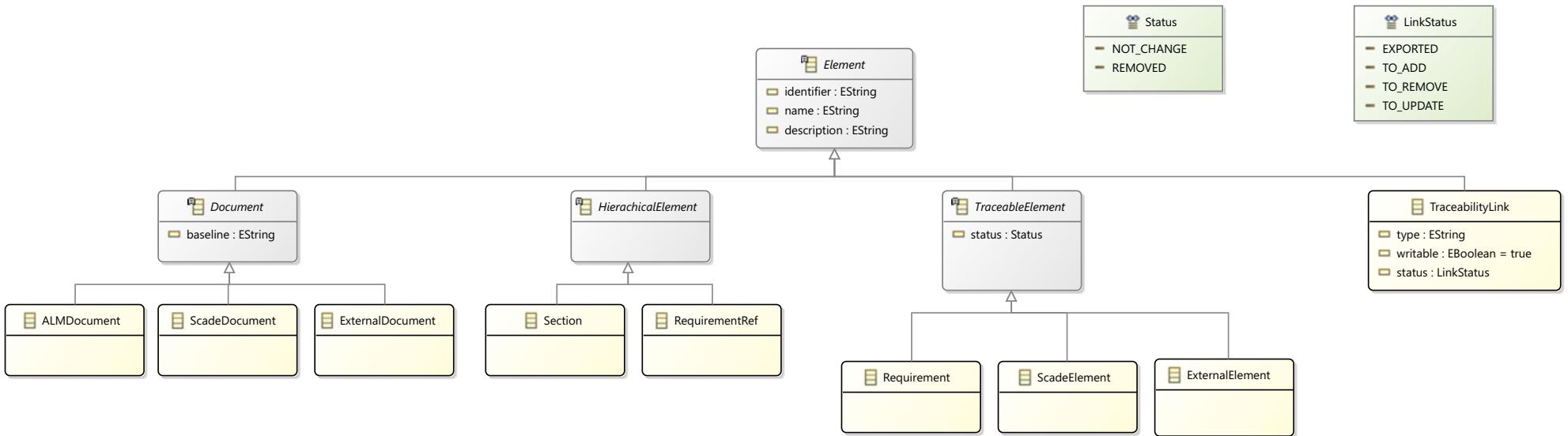
- 25/ ["SCADE Traceability Metamodels"](#)

## 25 /SCADE Traceability Metamodels

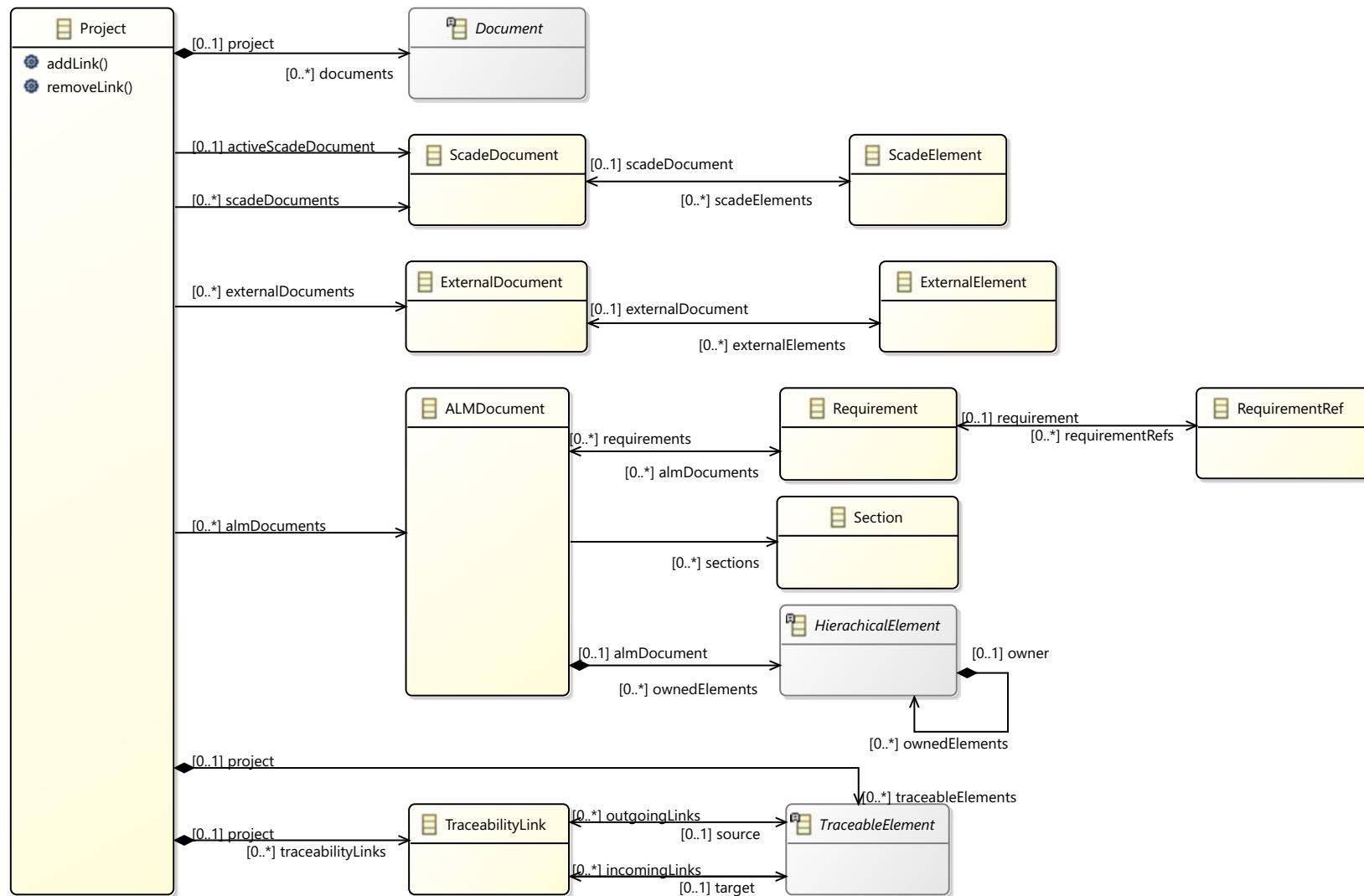
These metamodels present the data structure that give access to traceability data using Python API:

- [“Traceability Project Model \(inheritance\)”](#)
- [“Traceability Project Model \(composition\)”](#)

# Traceability Project Model (inheritance)



# Traceability Project Model (composition)





## Contact Information

*Submit questions to Technical Support at*  
[scade-support@ansys.com](mailto:scade-support@ansys.com)

*Contact one of our Sales representatives at*  
[scade-sales@ansys.com](mailto:scade-sales@ansys.com)

*Direct general questions about SCADE products to*  
[scade-info@ansys.com](mailto:scade-info@ansys.com)

*Discover the latest news on our products at*  
[www.ansys.com/products/embedded-software](http://www.ansys.com/products/embedded-software)

*Copyrights © 2024 ANSYS, Inc. All rights reserved. Ansys, SCADE, SCADE Suite, SCADE Display, SCADE Architect, SCADE LifeCycle, SCADE Test, and Twin Builder are trademarks or registered trademarks of ANSYS, Inc. or its subsidiaries in the U.S. or other countries. All other trademarks and tradenames contained herein are the property of their respective owners.*  
Revision: SCF-API-TRC-25 - 04/10/24