Ansys

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"

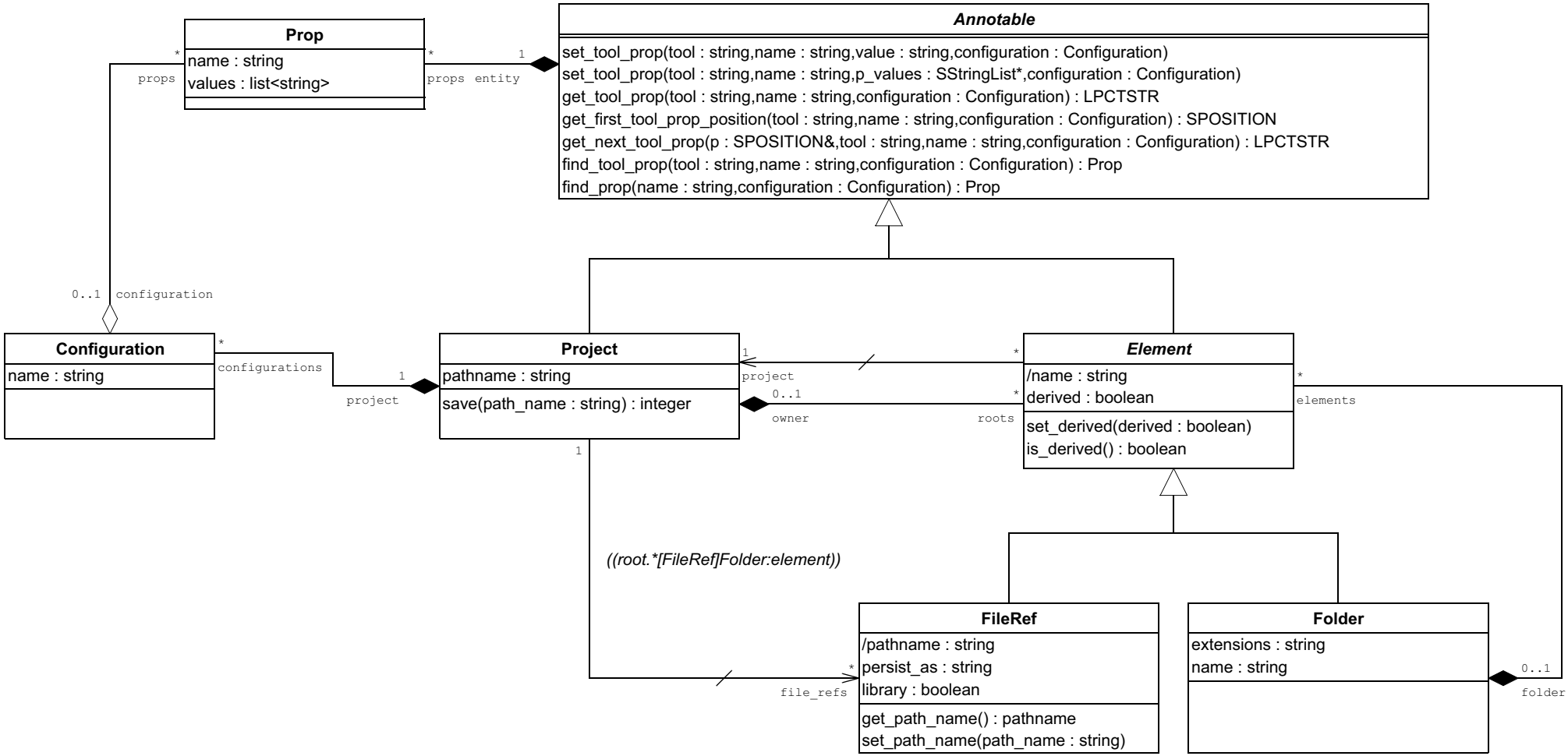
Contents

1 / SCADE Projects Metamodel

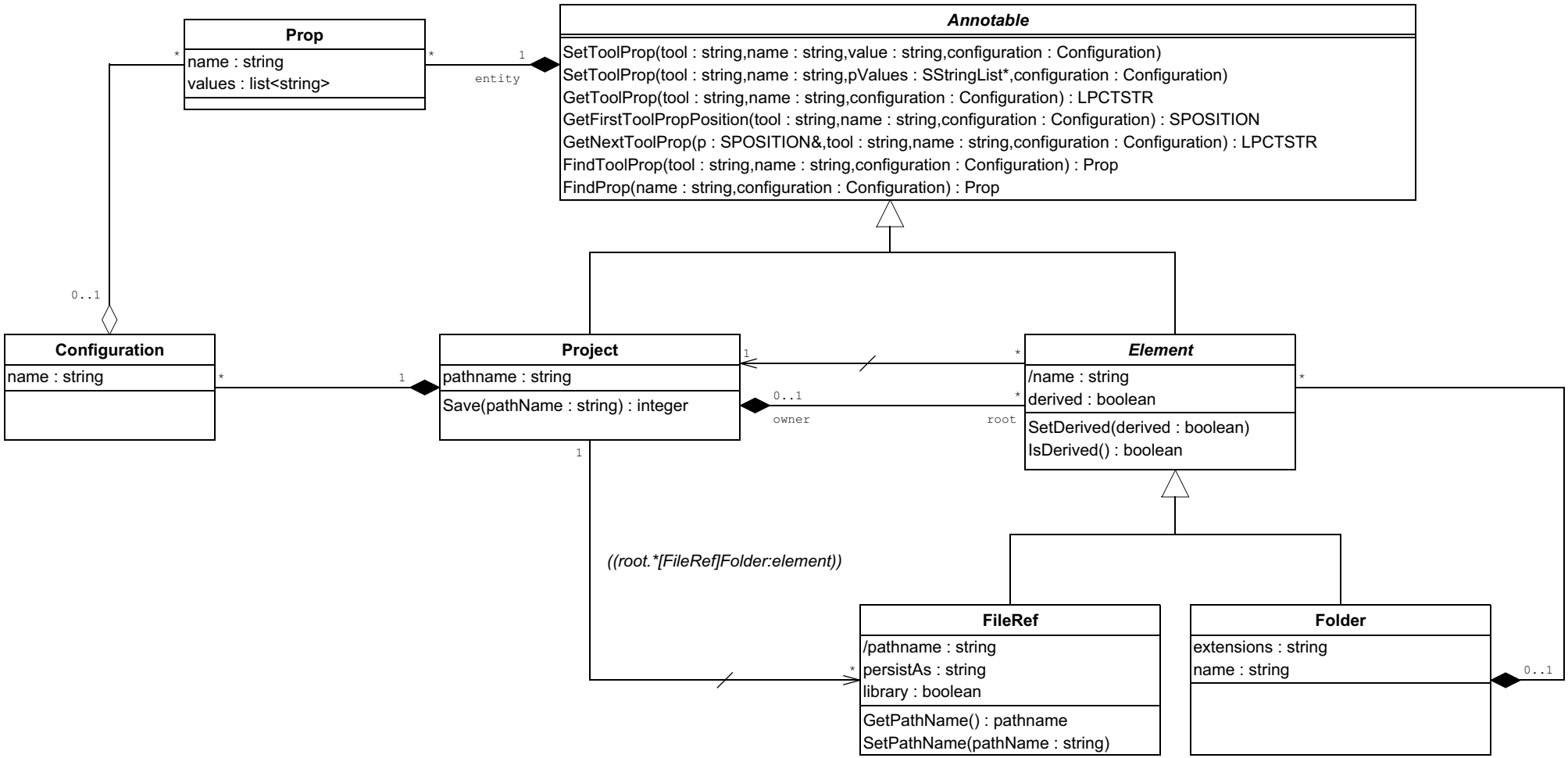
This metamodel presents the data structure that gives access to any SCADE project using SCADE Python API, Tcl API, or Java API:

- "SCADE Projects (Python)"
- "SCADE Projects (Tcl, Java)"

SCADE Projects (Python)



SCADE Projects (Tcl, Java)



Reference Card

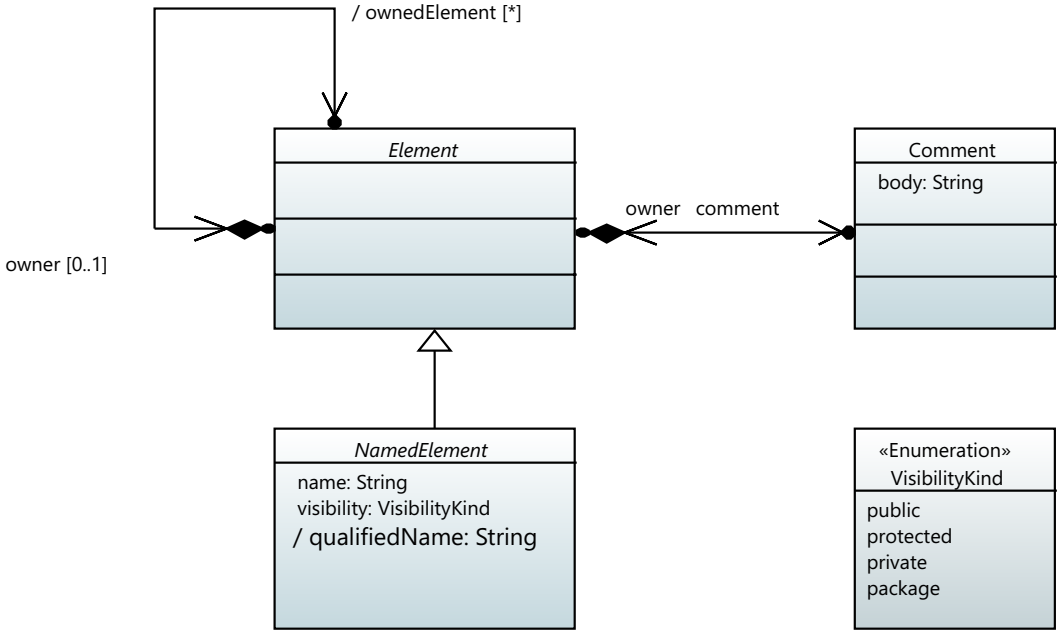
Part 1

SCADE Architect Metamodels

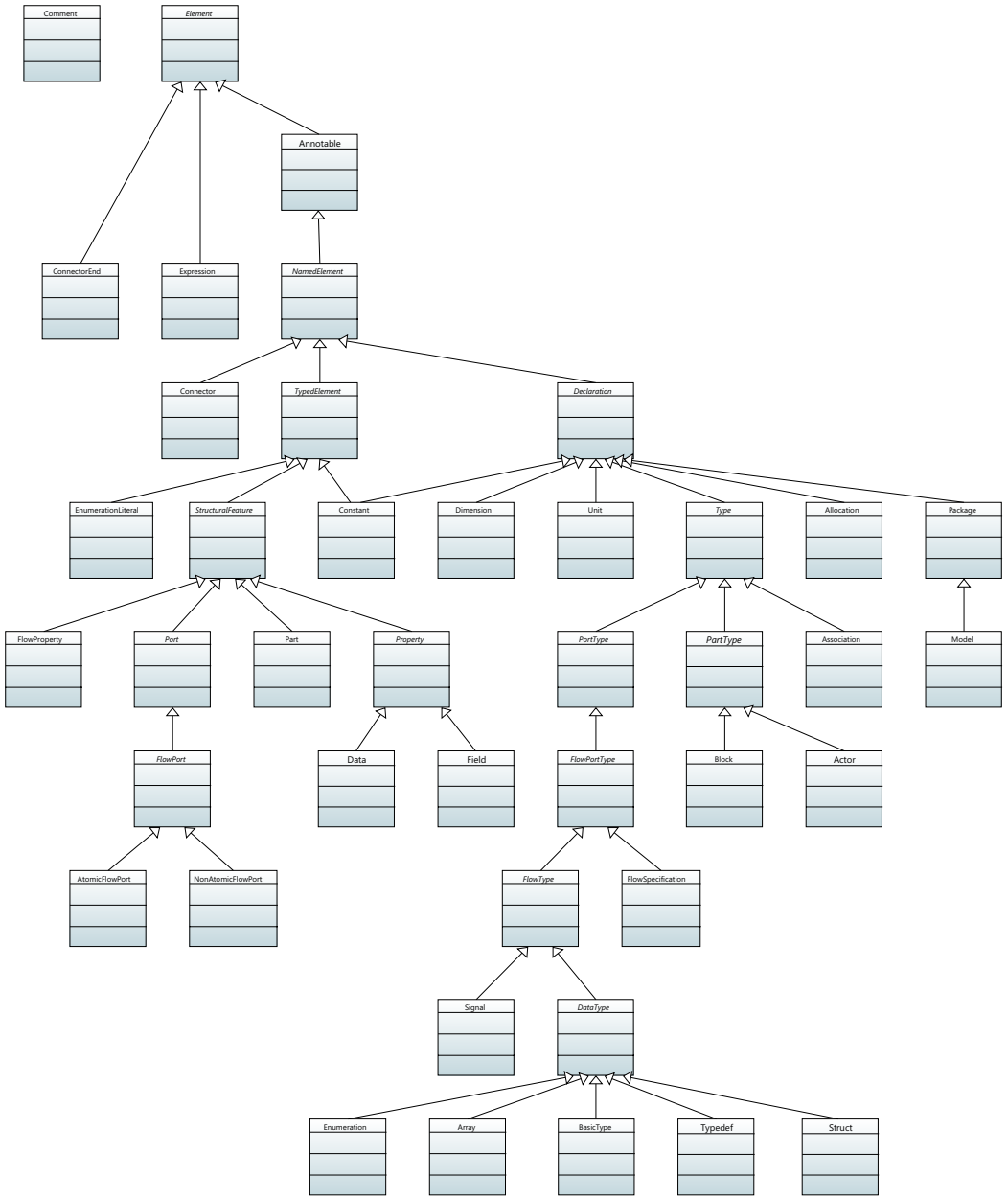
2 / General Construct Metamodels

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

Definition	The root abstract concept in the SCADE Architect Language. All other constructs except Comment inherit from Element.
Inheritance	n/a
Attributes	none

Associations	`comment: Comment`
	Comment applying to the element
	`owner: Element`
	Parent of the element from a lexical point of view
	`/ownedElement: Element {derived union}`
	Set of elements owned by the element (derived union where subclasses introduce concrete compositions that subset the composition and share the owner role)
	`table: Tablt`
	The set of tables owned by this element. Only a Package or a Block can contain table elements.
	`diagram: Diagram`
	The set of diagrams owned by this element. Only a Package or a Block can contain diagram elements.

Comment

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

NamedElement

Definition	The abstract concept for all elements having a name.
Inheritance	NamedElement → Element

Attributes	`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

Associations	`/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.

Declaration

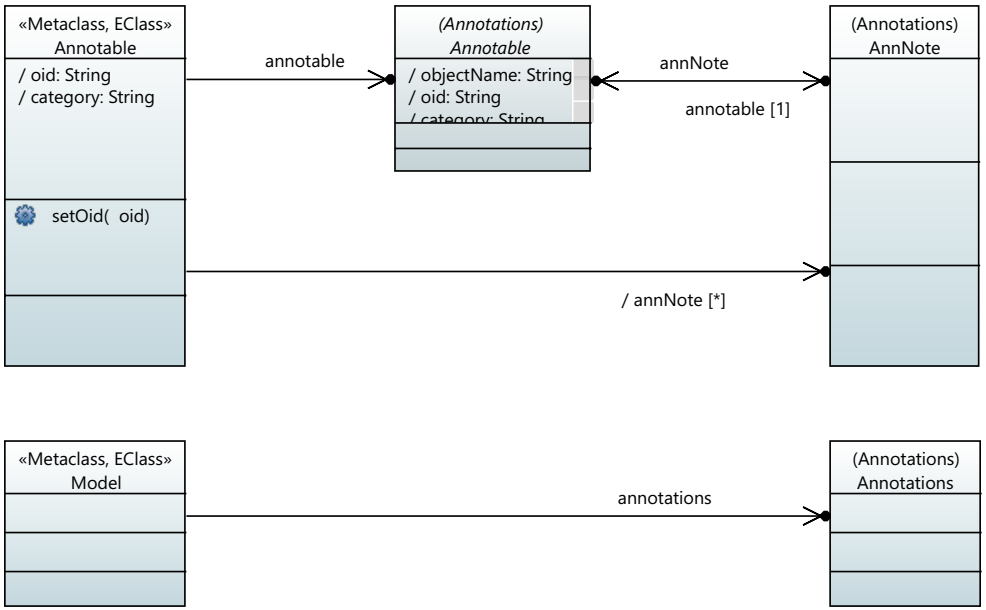
Definition	It represents any element that can be declared in a Package.
Inheritance	Declaration → NamedElement
Attributes	n/a
Associations	n/a

Annotable

Definition	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.
Inheritance	Annotable → Element

Attributes	`/oid: String`
	The oid of the annotable element
	`/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.

Associations	`annNote: AnnNote` The notes attached to Element

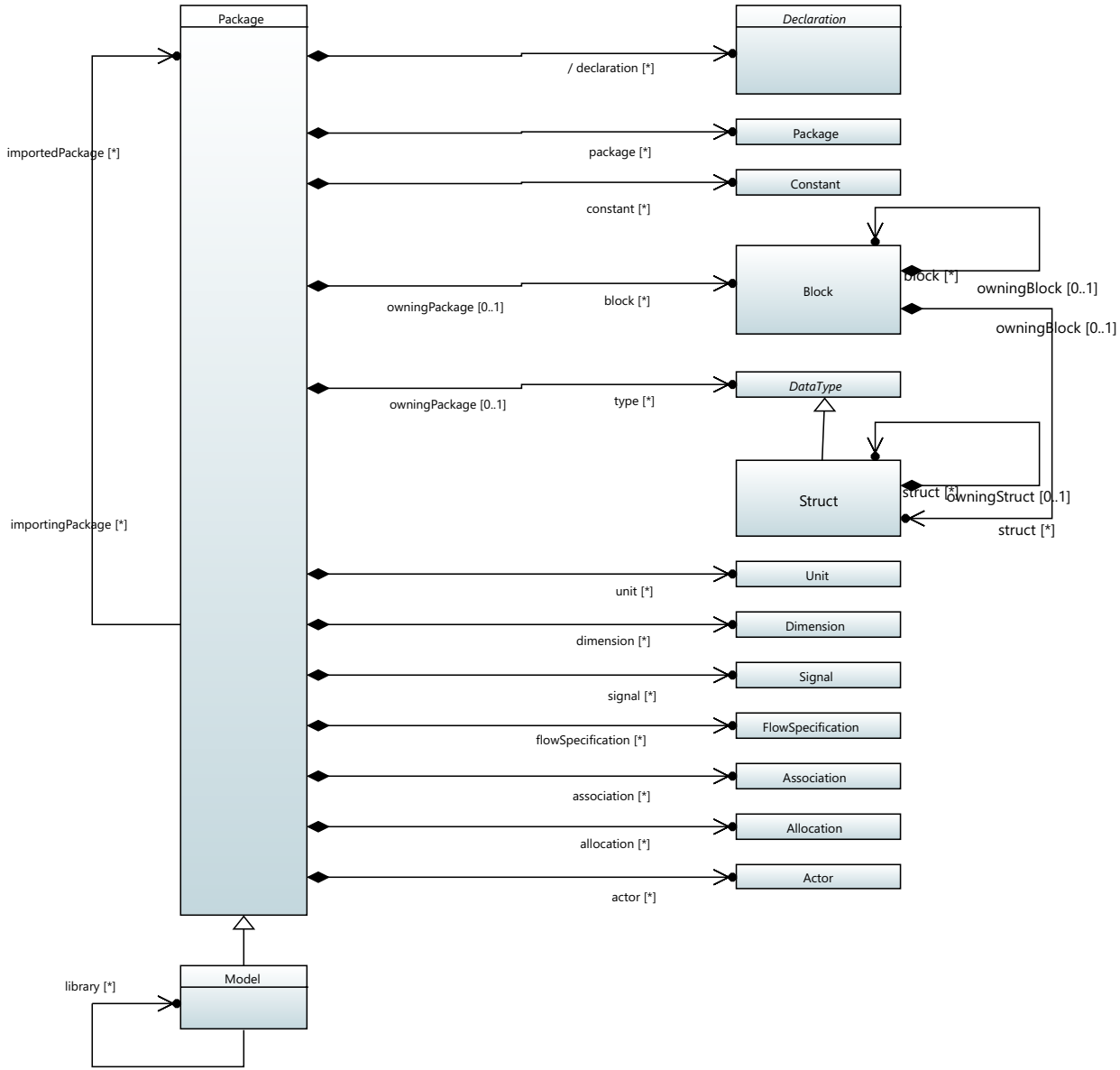


Package

Definition	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.
Inheritance	Package → Declaration
Attributes	n/a

Associations	`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)

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
Associations	`/structuralFeature: StructuralFeature[*]` 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`

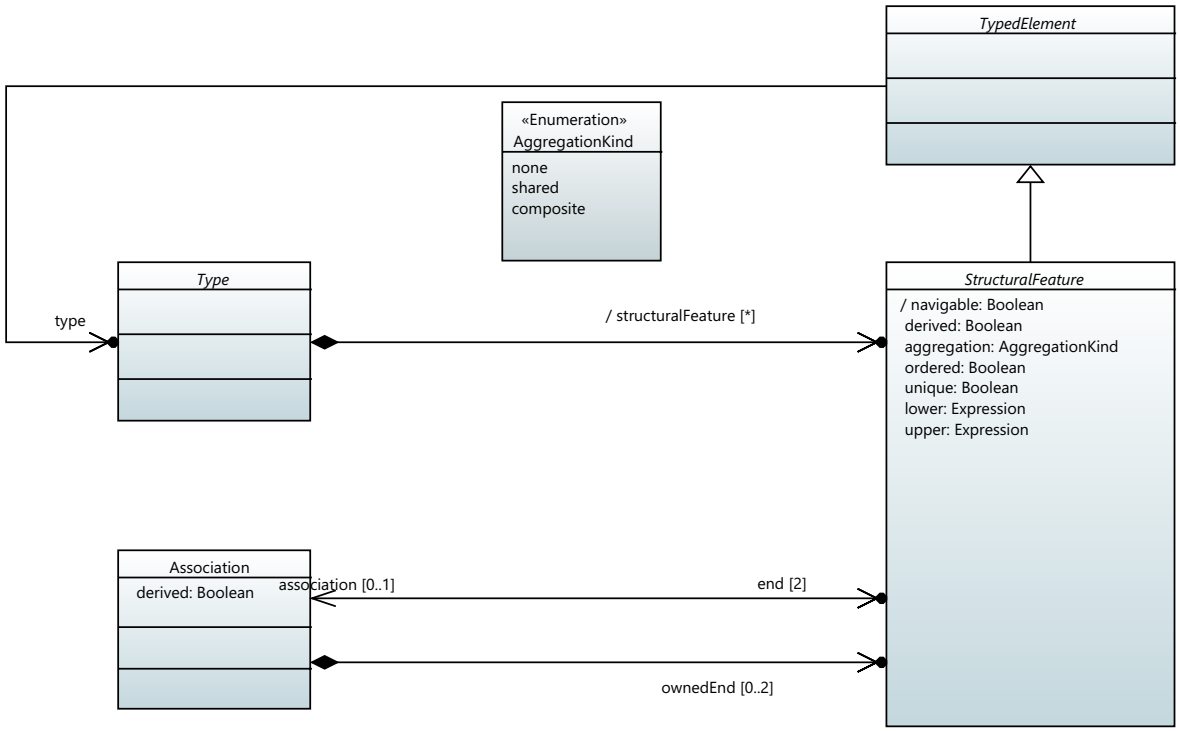
StructuralFeature

Definition	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.
Inheritance	StructuralFeature → TypedElement

Attributes	`lower: Expression`
	Lower bound on the number of sub-elements corresponding to the structural feature at runtime
	`upper: Expression`
	Upper bound on the number of sub-elements corresponding to the structural feature at runtime
	`ordered: Boolean`
	Whether the set of sub-elements at runtime is ordered (if there is more than one)
	`/navigable: Boolean`
	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)
	`aggregation: AggregationKind (none, shared or composite)`
	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.

Associations	n/a

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



Property

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

Association

Definition	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).
Inheritance	Association → NamedElement
Attributes	n/a

Associations	`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

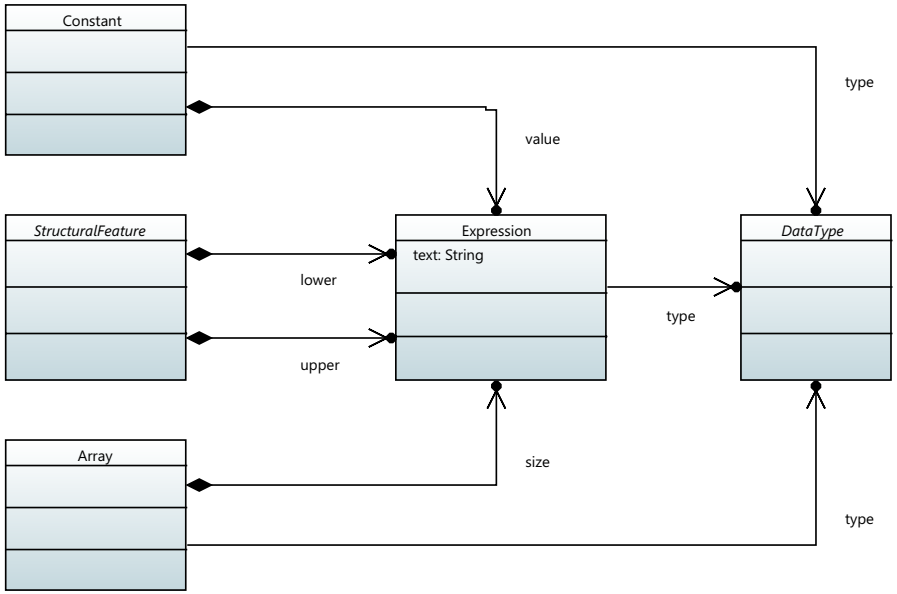
Definition	It represents a named expression that evaluates to a constant value. Constants in a model are typically used in expressions for the lower or upper bound multiplicities of StructuralFeatures, or for the size of Arrays.
Inheritance	Constant → Declaration, TypedElement
Attributes	`value: Expression` 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).
Associations	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

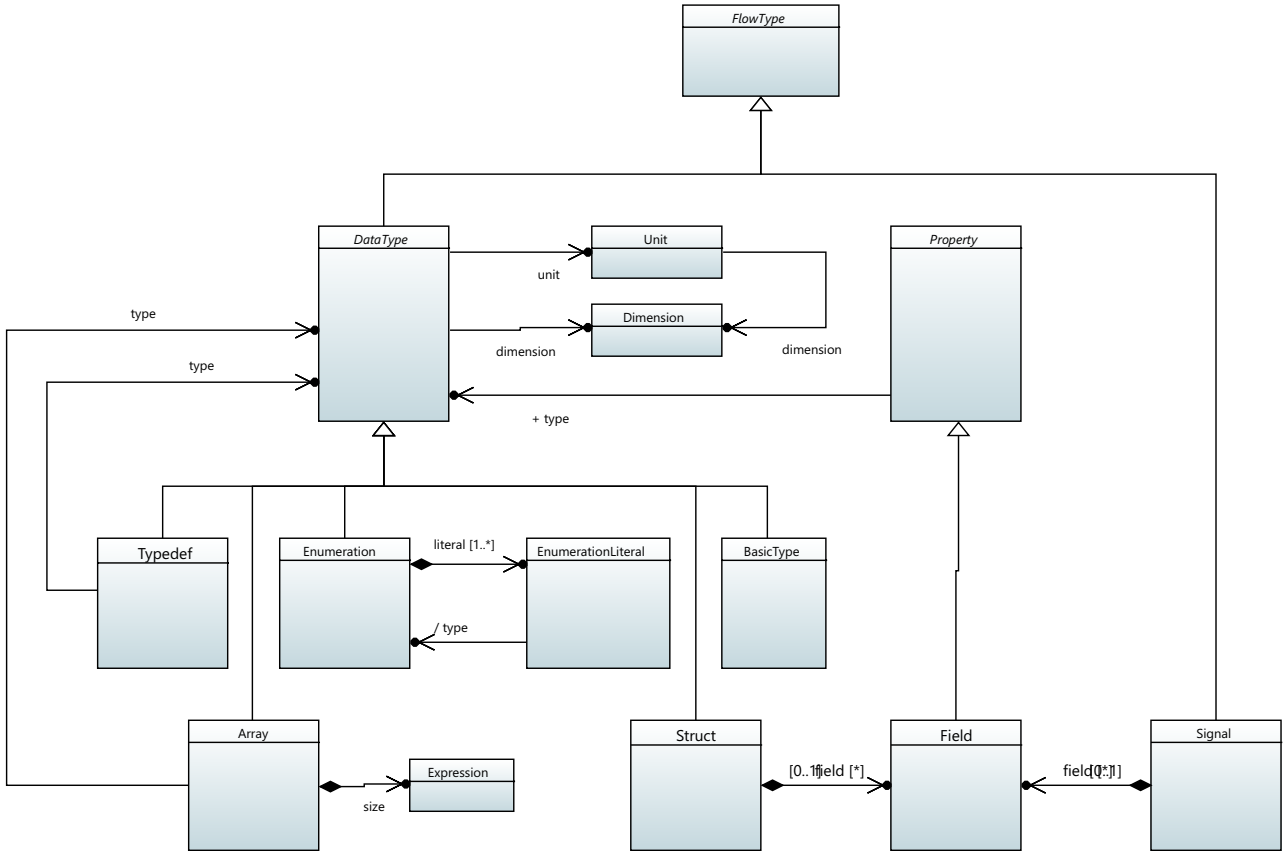
SCADE Architect only supports "opaque" expressions given in the form of non-interpreted text.



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

Definition	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).
Inheritance	DataType → FlowType
Attributes	n/a

Associations	`unit: Unit[0..1]`
	Optional unit of a DataType
	`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

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

EnumerationLiteral

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

Array

Definition	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).
Inheritance	Array → DataType
Attributes	n/a

Associations	`type: DataType`
	Type of elements present in the array
	`size: Expression`
	Size of the Array (size expression must be statically computable)

Signal

Definition	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).
Inheritance	Signal → FlowType
Attributes	n/a
Associations	`fields: Field[*] {ordered}` The fields of the signal.

Unit

Definition	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”).
Inheritance	Unit → Declaration
Attributes	n/a
Associations	`dimension: Dimension` Dimension of the Unit

Dimension

Definition	It represents the Dimension concept of the SI system. Examples of Dimension are “distance” or “weight”.
Inheritance	Dimension → Declaration
Attributes	n/a
Associations	n/a

Typedef

Definition	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.
Inheritance	Typedef → DataType
Attributes	n/a
Associations	`type: DataType` Data type of which this data type is an alias

BasicType

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

Struct

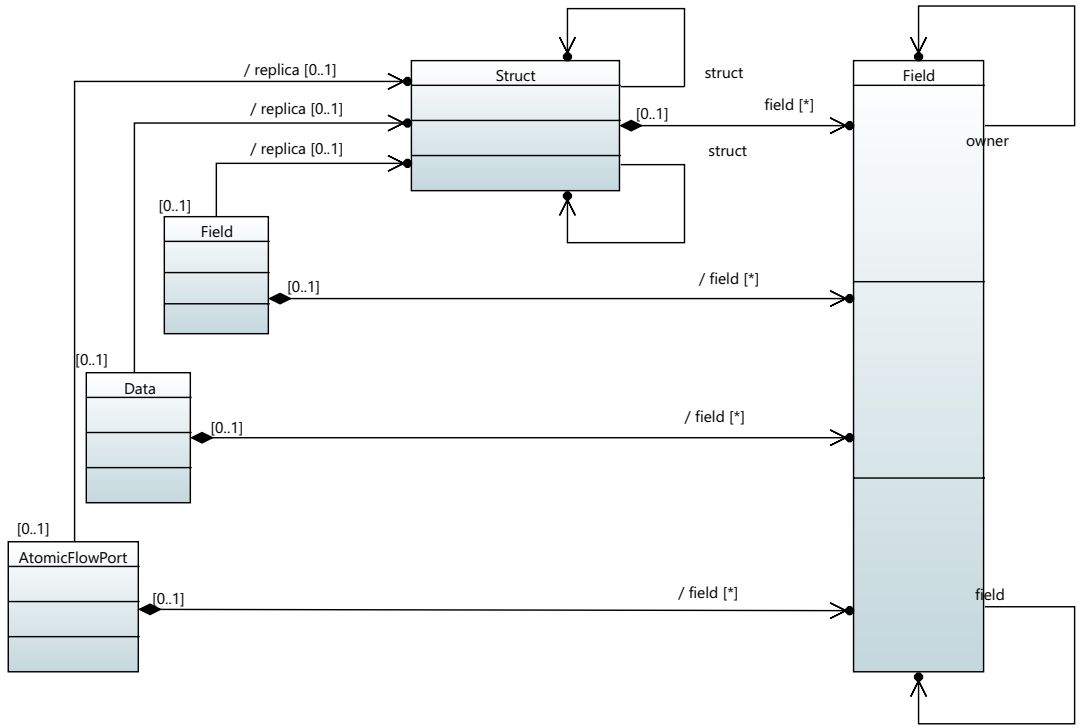
Definition	It represents a data type that can be further structured with fields. When used to type atomic flow ports, data or fields of other structs, the struct is replicated (using the same mechanism as for block replicas), which means it becomes possible to denote a field within a specific port, data or field (that is, a field within the struct replica that types the owning port, data or field).
Inheritance	Struct → DataType
Attributes	n/a

Associations	`n/a fields: Field[*] {ordered}`
	The fields of the data type.
	`prototype: Struct[1]`
	The prototype for a struct replica.
	declaration: Struct[0..1]
	The block declaration for a struct replica.

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

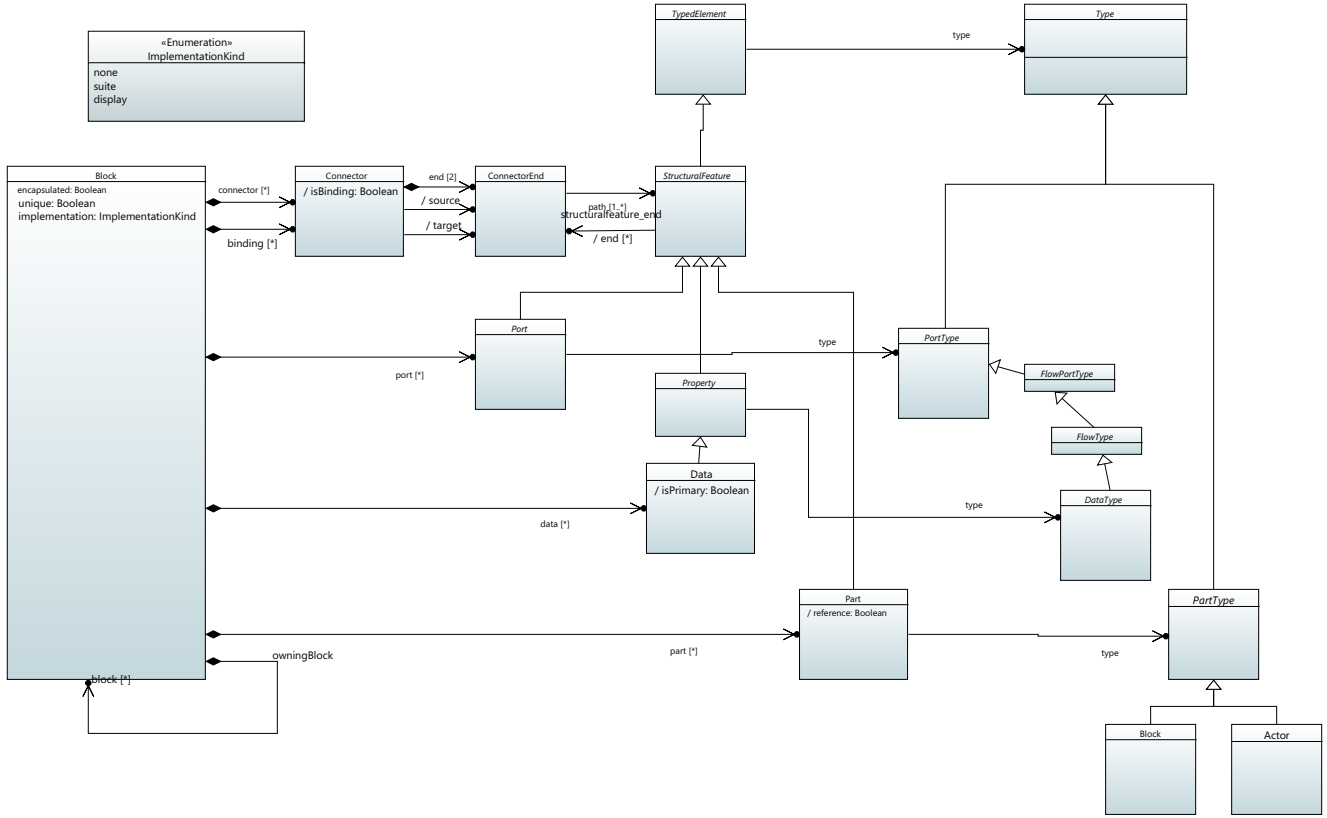
Definition	It represents a sub-value within of a Struct or Signal type.
Inheritance	Field → Property
Attributes	n/a

Associations	`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

Definition	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”.
Inheritance	Actor → PartType
Attributes	n/a
Associations	n/a

Block

Definition	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 is defined.
Inheritance	Block → PartType

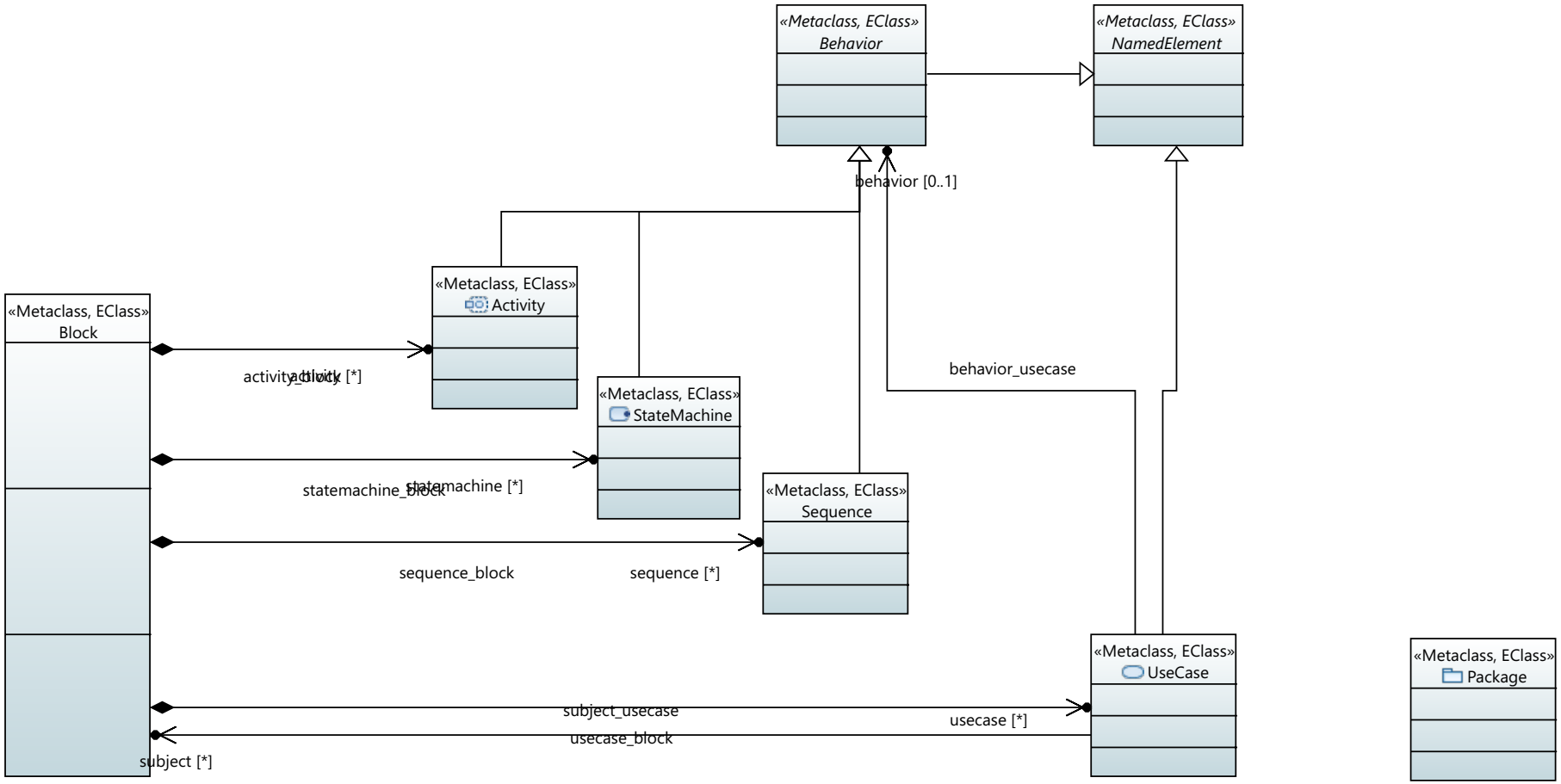
Attributes	`encapsulated: Boolean`
	Whether there can be inbound connections to inner parts/references of the block or whether the block is “opaque” with respect to the outside
	`unique: Boolean`
	Whether the block can be replicated several times for several parts, or whether the block is a unique block (meant to type only a single part).

Associations	`port: Port[*] {ordered, subsets structuralFeature}`
	Ports of the block
	`part: Part[*] {subsets structuralFeature}`
	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.
	`data: Data[*] {ordered, subsets structuralFeature}`
	Data of the block
	`connector: Connector[*]`
	Connectors of the block. Binding connectors are excluded.
	`binding: Connector[*]`
	Data-propagation binding connectors of the block
	`block: Block[*]`
	Unique blocks defined and used as parts of this block.
	`prototype: Block[1]`
	Prototype for a block replica
	`declaration: Block[0..1]`
	Block declaration for a block replica

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).

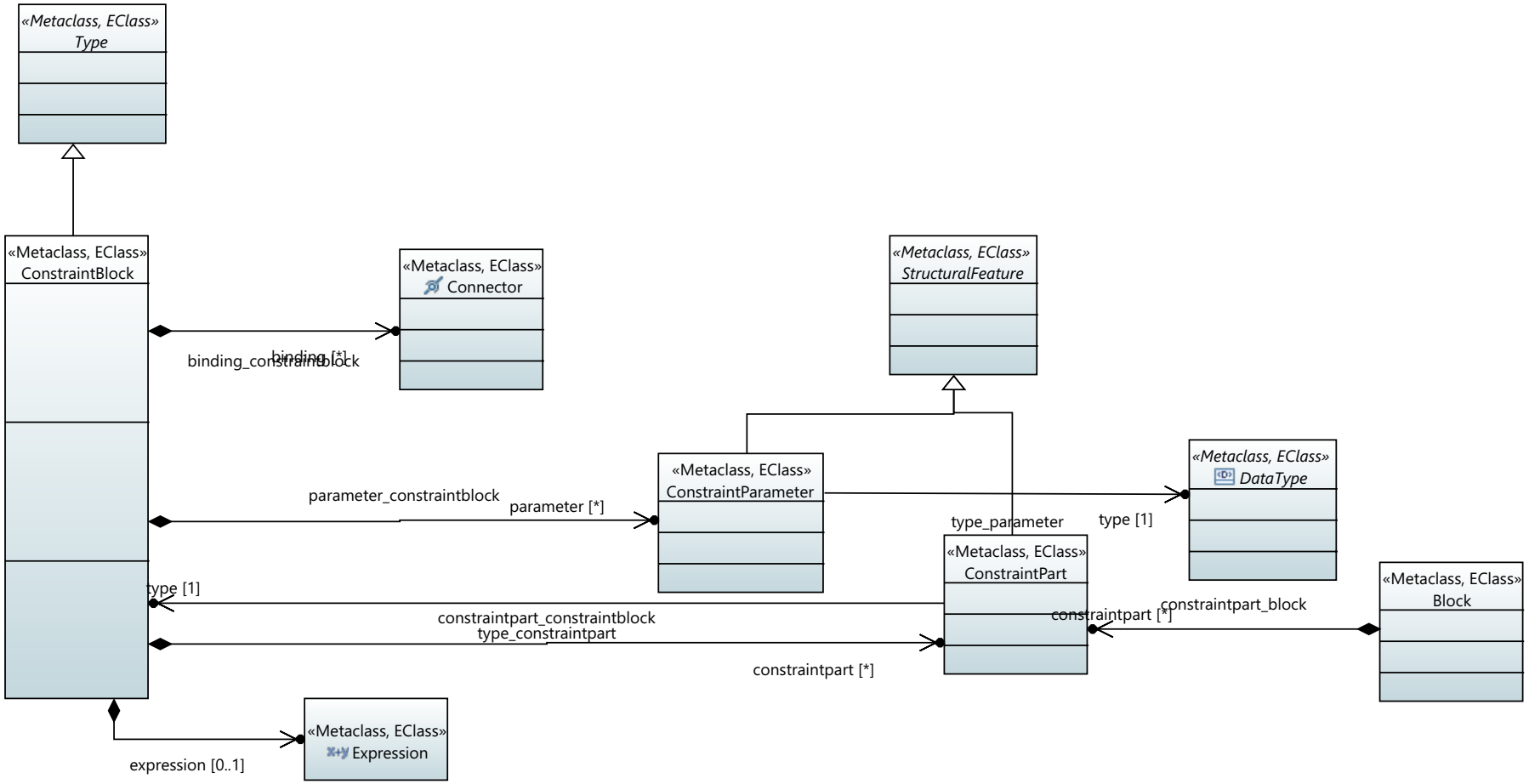
Behavior

The following metamodel shows the data structure for behavior definitions:



Constraint Blocks

The following metamodels shows the data structure for constraint blocks:



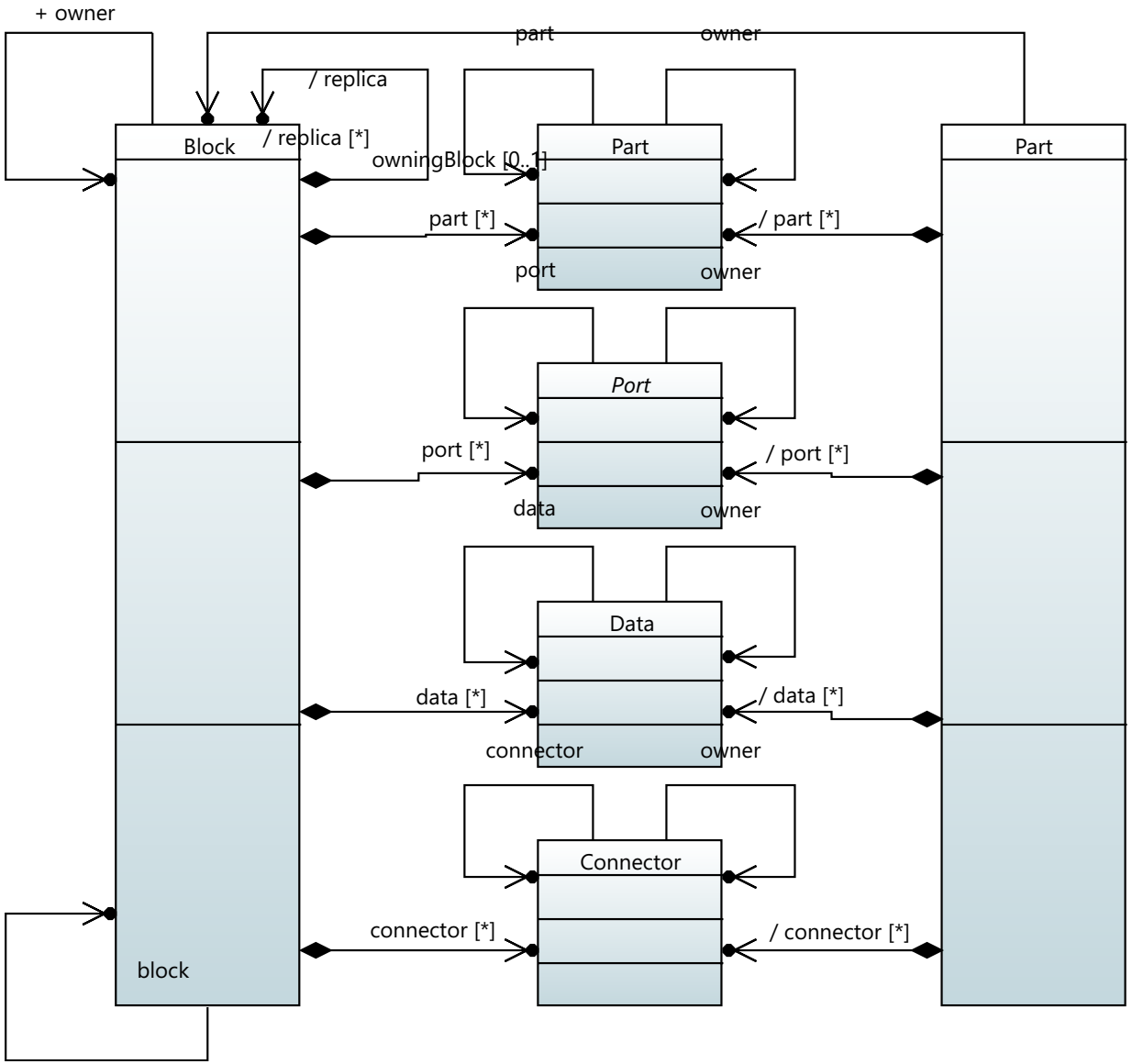
Part & Reference

Definition	*A part of a block* 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 *part by reference within a block* 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.
Inheritance	Part → StructuralFeature
Attributes	`/reference: Boolean` Whether the part is actually only a reference (not a full-fledged part) derived from `StructuralFeature.aggregation` (reference == true if aggregation != composite)

Associations	`type: PartType {redefines TypedElement.type}`
	Type of the Part (inherited from TypedElement) redefined to be restrained to Block or Actor
	`association: Association`
	Association of which this part is an end
	`/replica: Block[0..1]`
	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.
	`port: Port[*] {ordered, subsets structuralFeature}`
	Ports of the part, derived from the ports of the part's associated replica
	`part: Part[*] {subsets structuralFeature}`
	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
	`data: Property[*] {ordered, subsets structuralFeature}`
	Data of the part, derived from the data of the part's associated replica
	`connector: Connector[*]`
	Connectors of the part, derived from the connectors of the part's associated replica

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

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

Data

Definition	It represents some information carried by a block.
Inheritance	Data → Property
Attributes	`/isPrimary: Boolean` Whether the data is a primary data or a data proxy resulting from data propagation.

Associations	`replica: Struct[0..1]`
	Data-specific struct replica if this data is typed by a struct
	`fields: Field[*] {ordered}`
	Fields of this data, if the data is typed by a struct (that is, the fields of its replica)

Port

Definition	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: • Atomic flow ports: simple input/output flow ports • Non-atomic flow ports: grouped flow ports
Inheritance	Port → StructuralFeature
Attributes	n/a
Associations	`type: PortType {redefines TypedElement.type}` Type of the port (inherited from TypedElement) redefined to be restrained to PortType

PortType

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

Connector

Definition	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).
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

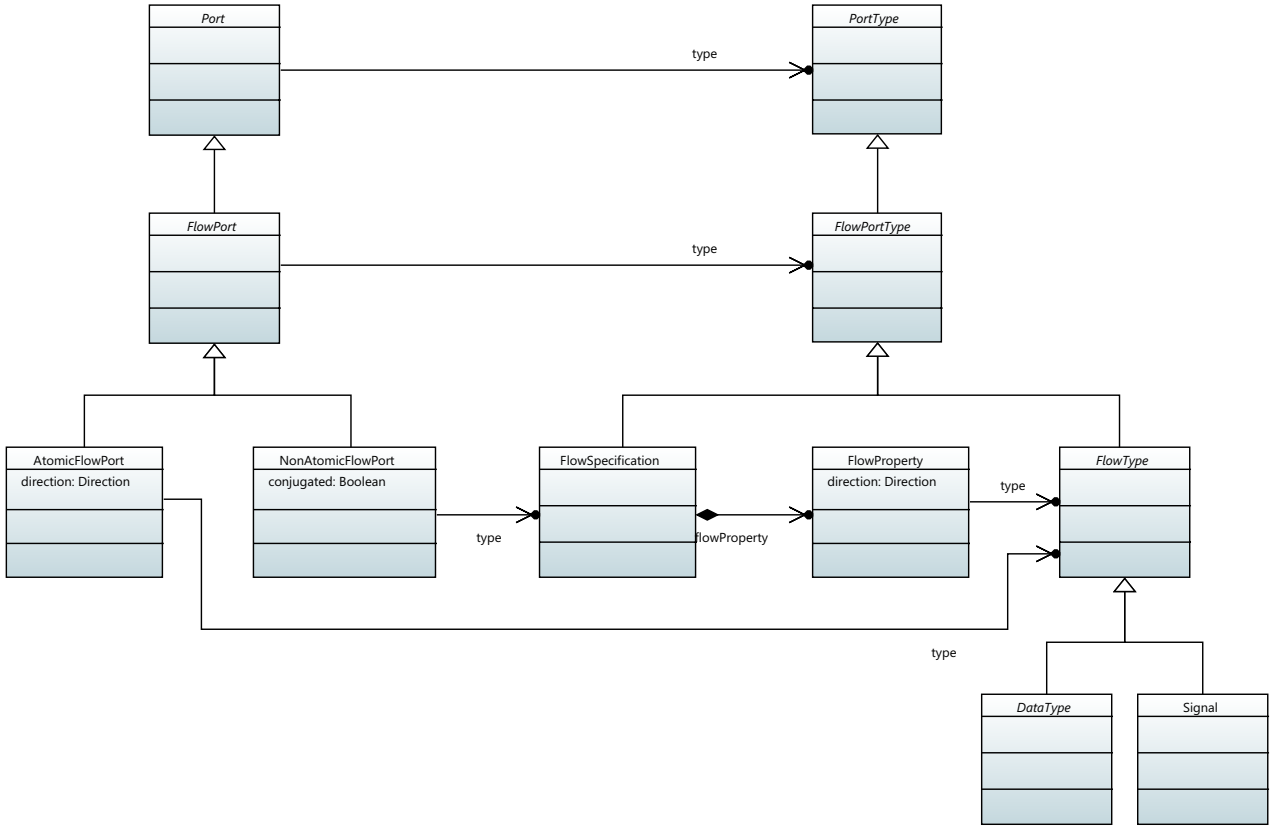
ConnectorEnd

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
Associations	`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 trough 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

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

FlowPort

Definition	An abstract concept for dataflow-oriented communication ports. It can take the following forms: • *Atomic flow ports* representing a single dataflow • *Non-atomic flow ports* representing a group of dataflows (group type is given through a so-called FlowSpecification that itself contains the individual dataflows)
Inheritance	FlowPort → Port
Attributes	`/atomic: Boolean` Whether the port is atomic (representing a single dataflow) or not

Associations	`type: FlowPortType {redefines Port.type}`
	Type of the FlowPort (inherited from Port) covariantly redefined to be restrained to FlowPortType (*i.e.,* FlowSpecification, DataType, or Signal)
	`replica: Struct[0..1]`
	Port-specific struct replica if this port is typed by a struct
	`fields: Field[*] {ordered}`
	Fields of this port, if the port is typed by a struct (that is, the fields of its replica)

NonAtomicFlowPort

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

FlowType

Definition	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).
Inheritance	FlowType → FlowPortType
Attributes	n/a
Associations	`field: Field[0..*] {ordered, subsets Type.structuralFeature}` Optional fields of the DataType or (valued) Signal

FlowSpecification

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

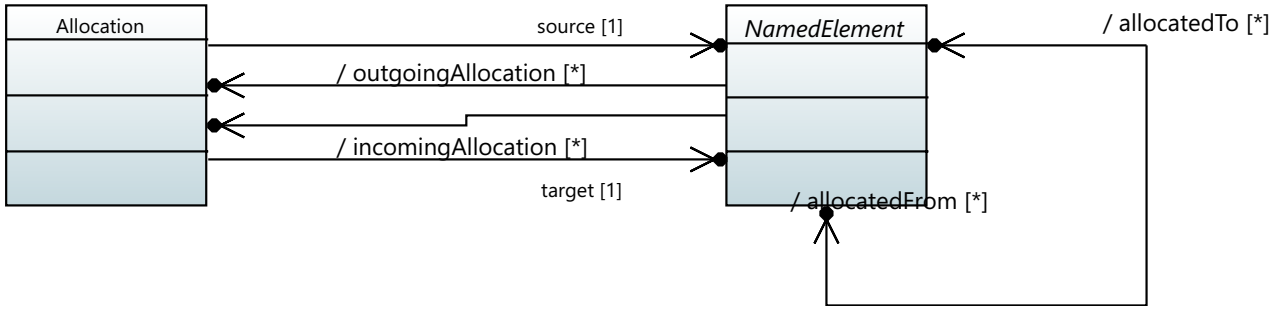
FlowProperty

Definition	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.
Inheritance	FlowProperty → StructuralFeature
Attributes	`direction: DirectionKind (enumeration in, out, inout)` Direction of the dataflow
Associations	`type: FlowType {redefines TypedElement.type}` 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

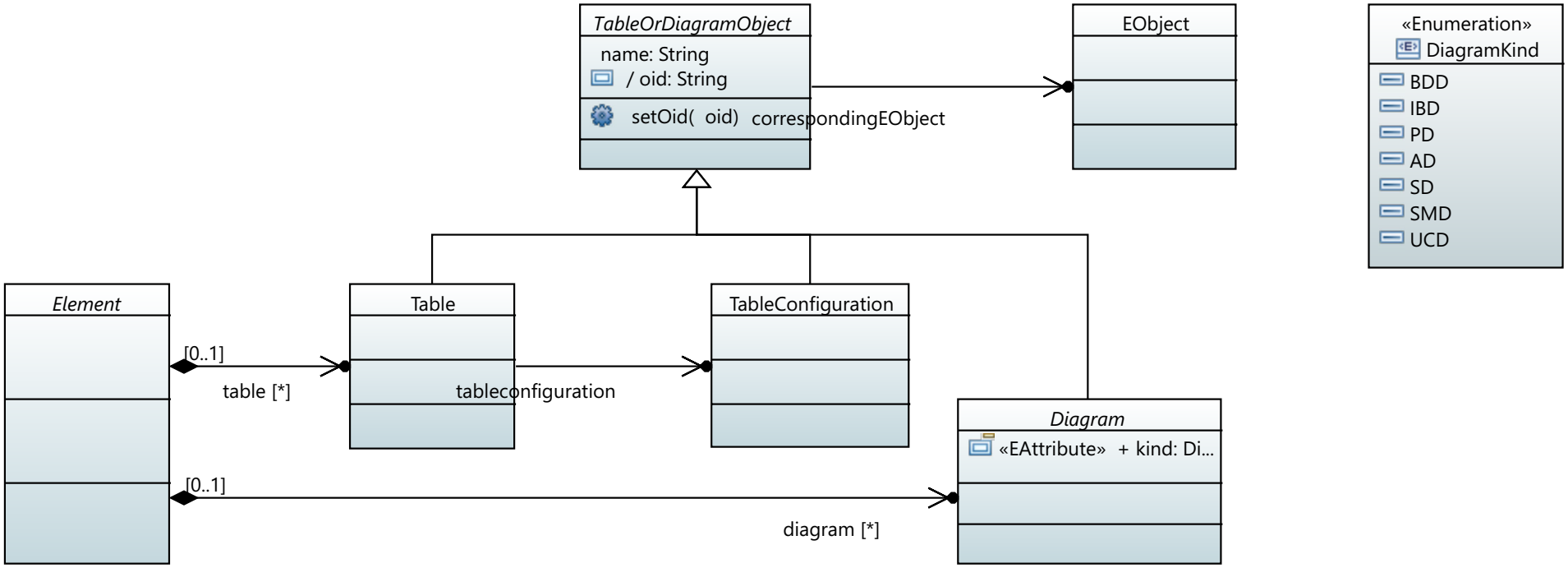
Definition	It is a directed relationship between two arbitrary elements that specifies that one element must be allocated onto another element. Usually, the source element represents a logical element (typically a logical function in system engineering) while the target element represents a physical element (typically an "organ" or a piece of hardware resource that supports the realization of the function).
Inheritance	Allocation → Declaration
Attributes	n/a

Associations	`source: NamedElement`
	Source element of the allocation (the element being allocated)
	`target: NamedElement`
	Target element of the allocation (the element onto which something is allocated)

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

Definition	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.
Inheritance	n/a
Attributes	`name: String` The name of the element.
Associations	`correspondingEObject: EObject` The corresponding EObject from the notation model (for diagram concepts) or nattable model (for table concepts).

Table

Definition	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).
Inheritance	Table → TableOrDiagramObject
Attributes	n/a
Associations	`tableconfiguration: TableConfiguration` The tableconfiguration of a table defines the table configuration (or style) applied to the table.

TableConfiguration

Definition	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, ...).
Inheritance	Table → TableOrDiagramObject
Attributes	n/a
Associations	n/a

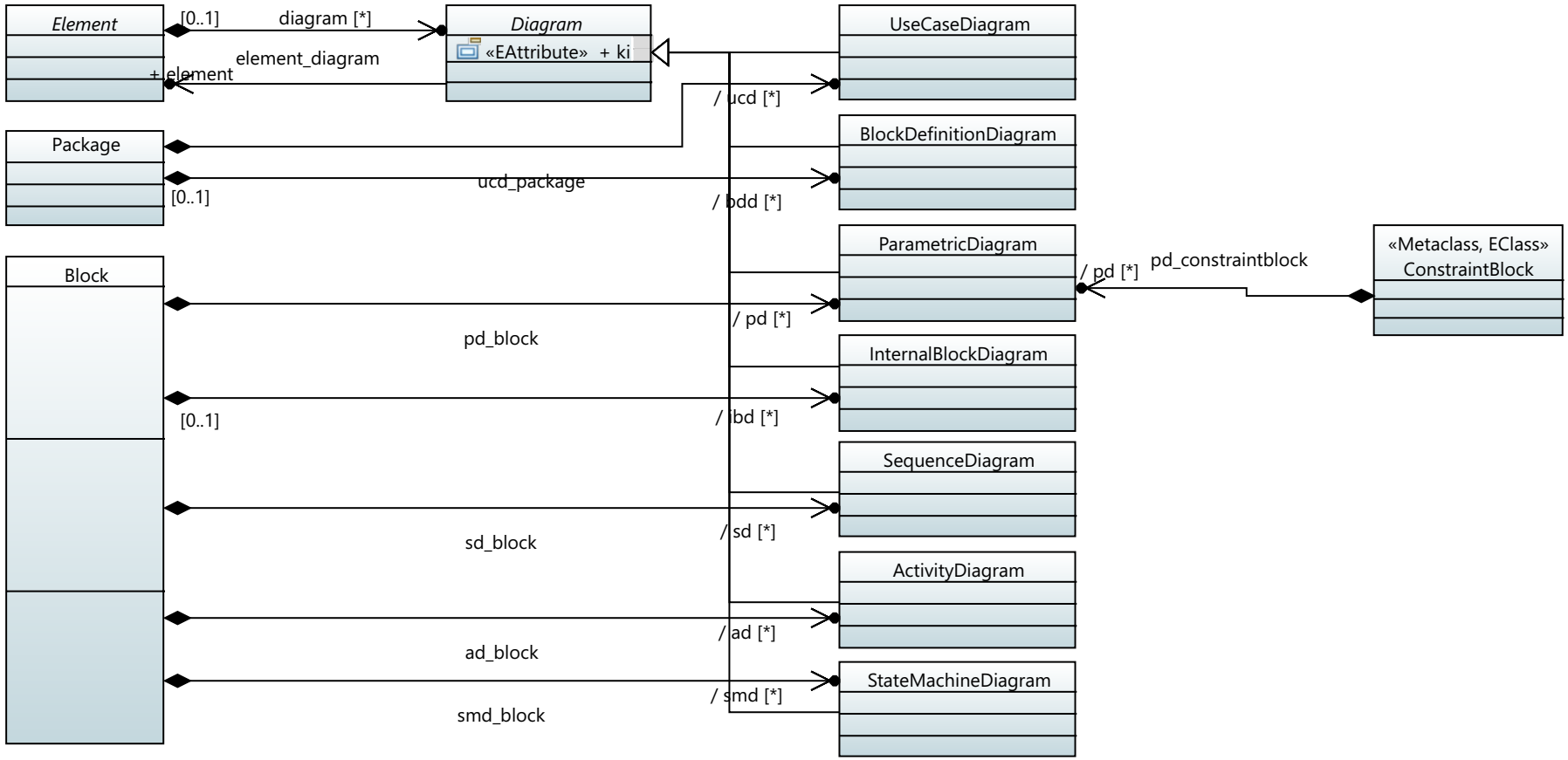
Diagram

Definition	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).
Inheritance	Diagram → TableOrDiagramObject
Attributes	`kind: DiagramKind (enumeration BDD, IBD, PD, AD, SD, SMD, UCD)` The kind of the diagram: Block Definition, Internal Block, Parametric, Activity, Sequence, State Machine, or Use Case.
Associations	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)

Definition	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'.
Inheritance	BlockDefinitionDiagram → Diagram
Attributes	n/a
Associations	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)

Definition	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'.
Inheritance	InternalBlockDiagram → Diagram
Attributes	n/a
Associations	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)

Definition	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'.
Inheritance	ParametricDiagram → Diagram
Attributes	n/a
Associations	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)

Definition	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’.
Inheritance	ActivityDiagram → Diagram
Attributes	n/a
Associations	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)

Definition	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’.
Inheritance	SequenceDiagram → Diagram
Attributes	n/a
Associations	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)

Definition	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'.
Inheritance	StateMachineDiagram → Diagram
Attributes	n/a
Associations	n/a

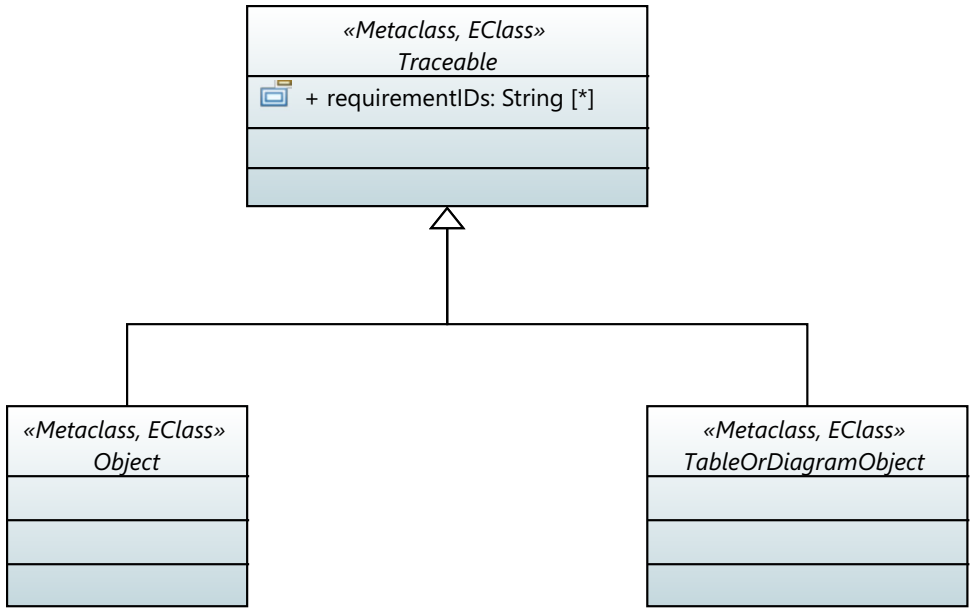
Such a diagram shows possible States and Transitions of a Block. Complex State Machine could be managed using Sub-Machine or using State into State. Transitions allow users to type free text (with multi-line support), persistence is performed by a Constraint object that wraps a LiteralString. A State 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)

Definition	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'.
Inheritance	UseCaseDiagram → Diagram
Attributes	n/a
Associations	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

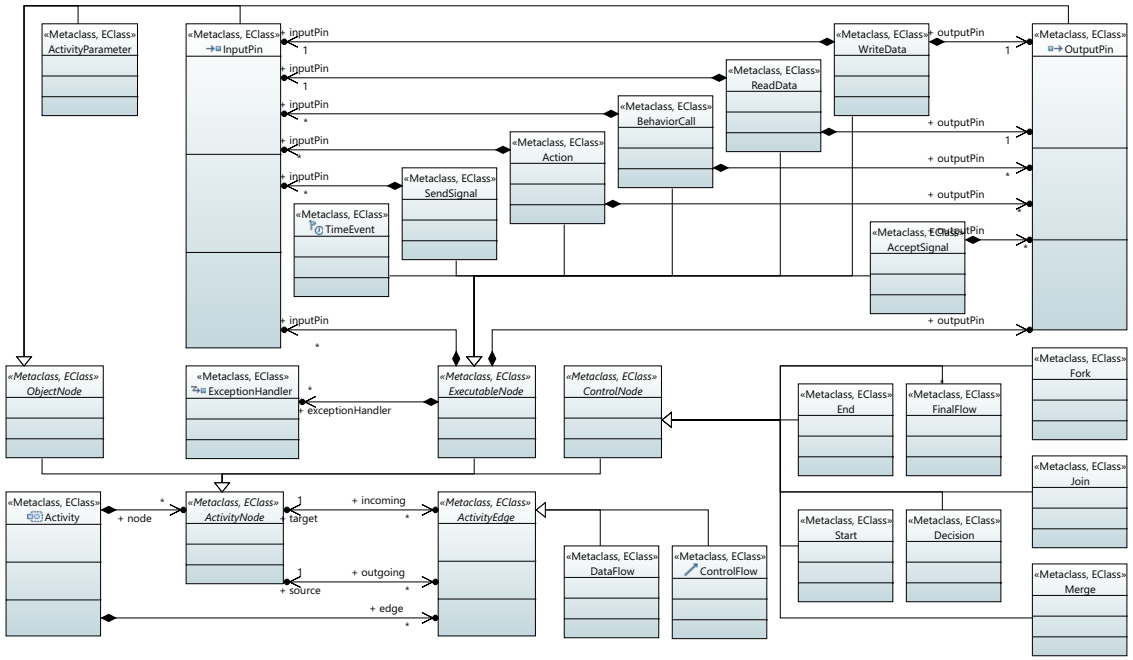
These metamodels present the data structures that give access to SCADE Architect behavior modeling using Python API, Java API, or Tcl API:

- “Activity”
- “State Machine”

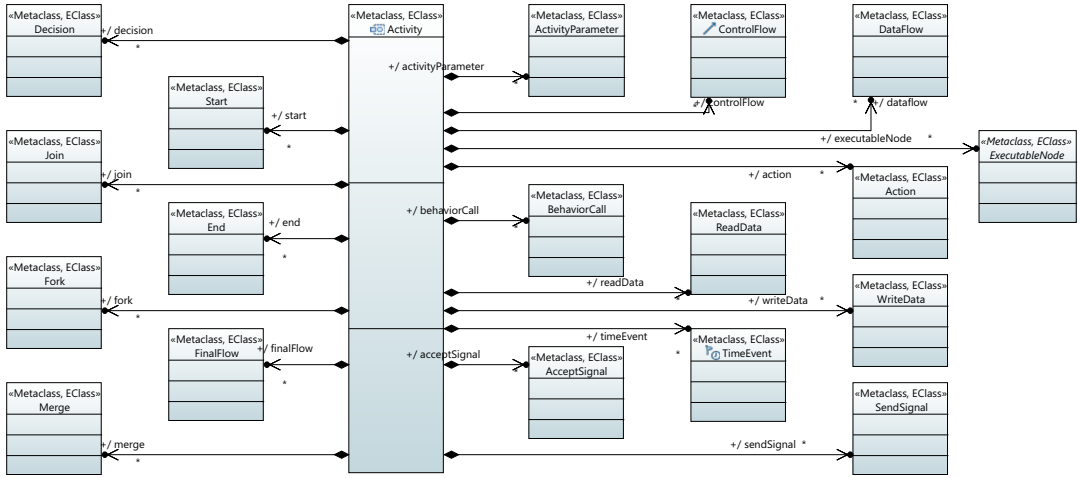
Activity

The Activity concept relies on the constructs described below.

- “Activity”
- “ActivityNode”
- “ActivityEdge”
- “ObjectNode”
- “ExecutableNode”
- “ControlNode”
- “AcceptSignal”
- “Action”
- “ActivityParameter”
- “BehaviorCall”
- “ControlFlow”
- “DataFlow”
- “Decision”
- “End”
- “ExceptionalHandler”
- “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

Associations	`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

ActivityNode

Definition	ActivityNodes are used to model the individual steps in the behavior specified by an Activity.
Inheritance	NamedElement → Annotable
Attributes	n/a

Associations	`outgoing: ActivityEdge[*]`
	Outgoing edges of the node
	`incoming: ActivityEdge[*]`
	IncomingEdges of the node

ActivityEdge

Definition	An ActivityEdge is a directed connection between two ActivityNodes along which tokens may flow, from the source ActivityNode to the target ActivityNode.
Inheritance	NamedElement → Annotable
Attributes	n/a

Associations	`source: ActivityNode`
	Source of the edge
	`target: ActivityNode`
	Target of the edge

ObjectNode

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

ExecutableNode

Definition	An ExecutableNode is a kind of ActivityNode that may be executed as a step in the overall desired behavior of the containing Activity.
Inheritance	ActivityNode → NamedElement
Attributes	n/a

Associations	`exceptionHandler: ExceptionHandler[*]`
	ExceptionHandler of the node
	`inputPin: InputPin[*]`
	Input pins of the node
	`outputPin: OutputPin[*]`
	Output pins of the node

ControlNode

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

AcceptSignal

Definition	An AcceptSignal is an action that waits for the occurrence of one Signal event.
Inheritance	ExecutableNode → ActivityNode
Attributes	n/a

Associations	`signal:Signal`
	The signal the action is waiting for
	`outputPin: OutputPin[*]`
	Output pins of the action

Action

Definition	An Action is a fundamental unit of executable functionality whose specification may be given in a textual concrete syntax.
Inheritance	ExecutableNode → ActivityNode
Attributes	n/a

Associations	`inputPin: InputPin[*]`
	Input pins of the action
	`outputPin: OutputPin[*]`
	Output pins of the action

ActivityParameter

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

BehaviorCall

Definition	A BehaviorCall is an Action that results in the invocation of a Behavior.
Inheritance	ExecutableNode → ActivityNode
Attributes	n/a

Associations	`behavior: Behavior`
	The behavior invoked by the action
	`inputPin: InputPin[*]`
	Input pins of the action
	`outputPin: OutputPin[*]`
	Output pins of the action

ControlFlow

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

DataFlow

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

Decision

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

End

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

ExceptionalHandler

Definition	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.
Inheritance	Annotable → Element
Attributes	n/a
Associations	n/a

FinalFlow

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

Fork

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

InputPin

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

Join

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

Merge

Definition	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.
Inheritance	ControlNode → ActivityNode
Attributes	n/a
Associations	n/a

OutputPin

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

ReadData

Definition	A ReadData is an Action that retrieves the values of a StructuralFeature.
Inheritance	ExecutableNode → ActivityNode
Attributes	n/a

Associations	`data: Data`
	The data to be read
	`inputPin: InputPin`
	Input pin of the action
	`outputPin: OutputPin`
	Output pin of the action

SendSignal

Definition	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.
Inheritance	ExecutableNode → ActivityNode
Attributes	n/a

Associations	`signal:Signal`
	The signal to be sent by the action
	`inputPin: InputPin[*]`
	Input pins of the action

Start

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

TimeEvent

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

WriteData

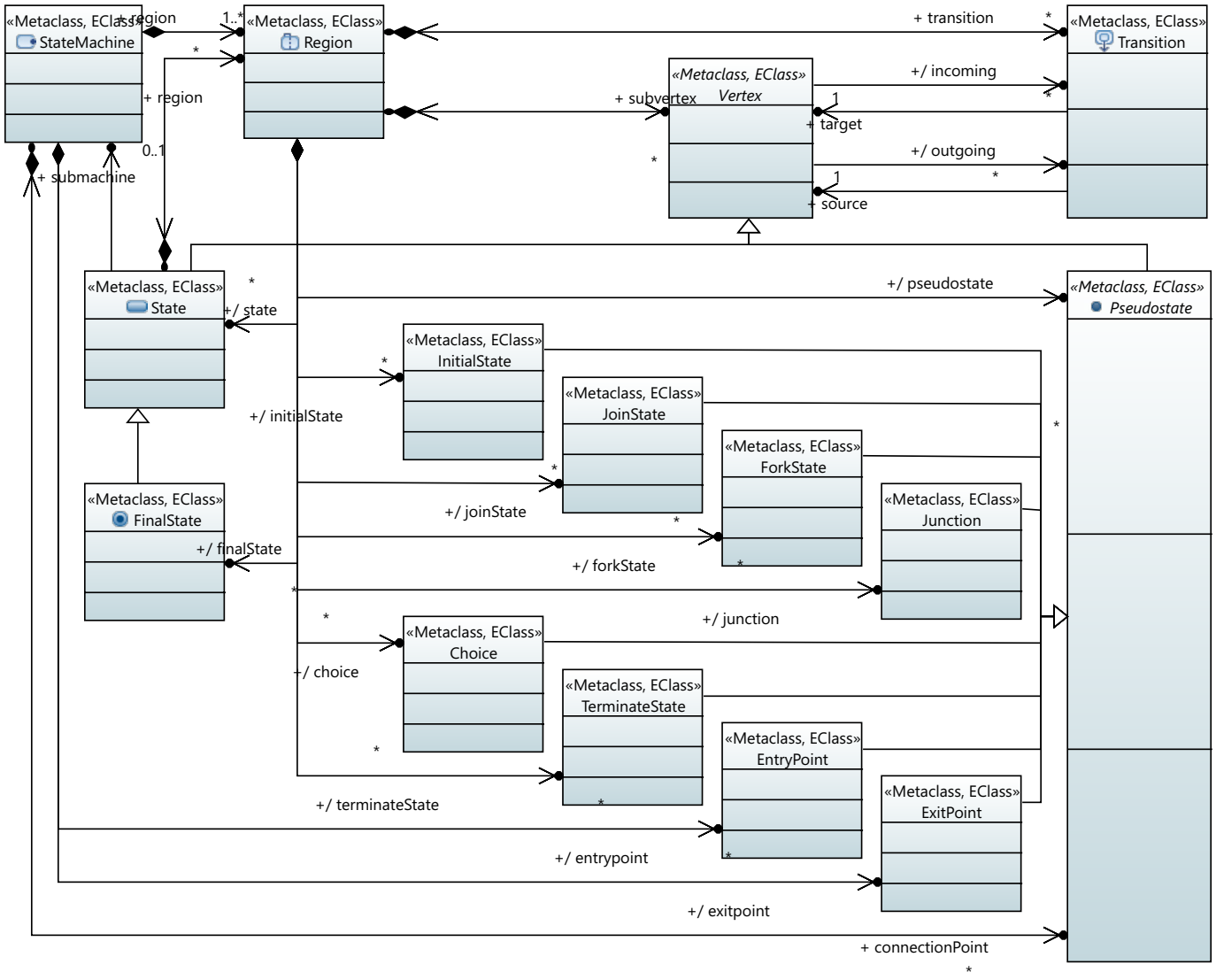
Definition	A WriteData is an Action for adding values to a StructuralFeature.
Inheritance	ActivityNode → NamedElement
Attributes	n/a

Associations	`data: Data`
	The data to be added
	`inputPin: InputPin`
	Input pin of the action
	`outputPin: OutputPin`
	Output pin of the action

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”


«Metaclass, EClass»
StateMachine
+ region
1..*
«Metaclass, EClass»
Region
+ region
0..1
+ submachine
+ transition
«Metaclass, EClass»
Transition
«Metaclass, EClass»
Vertex
+ subvertex
+/ incoming
1
+ target
+/ outgoing
1
+ source
«Metaclass, EClass»
State
+/ state
+/ pseudostate
«Metaclass, EClass»
Pseudostate
«Metaclass, EClass»
InitialState
+/ initialState
«Metaclass, EClass»
JoinState
+/ joinState
«Metaclass, EClass»
ForkState
+/ forkState
«Metaclass, EClass»
Junction
+/ junction
«Metaclass, EClass»
FinalState
+/ finalState
«Metaclass, EClass»
Choice
+/ choice
«Metaclass, EClass»
TerminateState
+/ terminateState
«Metaclass, EClass»
EntryPoint
+/ entrypoint
«Metaclass, EClass»
ExitPoint
+/ exitpoint
+ connectionPoint

StateMachine

Definition	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.
Inheritance	Behavior → NamedElement
Attributes	n/a

Associations	`connectionPoint: Pseudostate[*]`
	Connection points defined for this StateMachine
	`region: Region[1..*]`
	Regions owned by the StateMachine

Choice

Definition	Choice is used to realize a dynamic conditional branch.
Inheritance	Pseudostate → Vertex
Attributes	n/a
Associations	n/a

EntryPoint

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

ExitPoint

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

FinalState

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

ForkState

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

InitialState

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

JoinState

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

Junction

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

Pseudostate

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

Region

Definition	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.
Inheritance	NamedElement → Annotable
Attributes	n/a

Associations	`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 }`
	Setof 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.

State

Definition	A State models a situation during which some (usually implicit) invariant condition holds.
Inheritance	Vertex → NamedElement
Attributes	n/a

Associations	`region: Region[*]`
	Regions owned directly by the State
	`submachine: StateMachine[0..1]`
	StateMachine to be inserted in place of the (submachine) State

TerminateState

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

Transition

Definition	A Transition represents an arc between exactly one source Vertex and exactly one Target vertex (the source and targets may be the same Vertex).
Inheritance	NamedElement → Annotable

Attributes	`condition: String`
	Constraint specification providing a fine-grained control over the firing of the Transition
	`effect: String`
	Specifies an optional behavior to be performed when the Transition fires
	`trigger: String`
	Specifies the Trigger that may fire the transition

Associations	`source: Vertex`
	Designates the originating Vertex (State or Pseudostate) of the Transition
	`target: Vertex`
	Designates the target Vertex that is reached when the Transition is taken
	`parent: Region`
	Designates the Region that owns this Transition

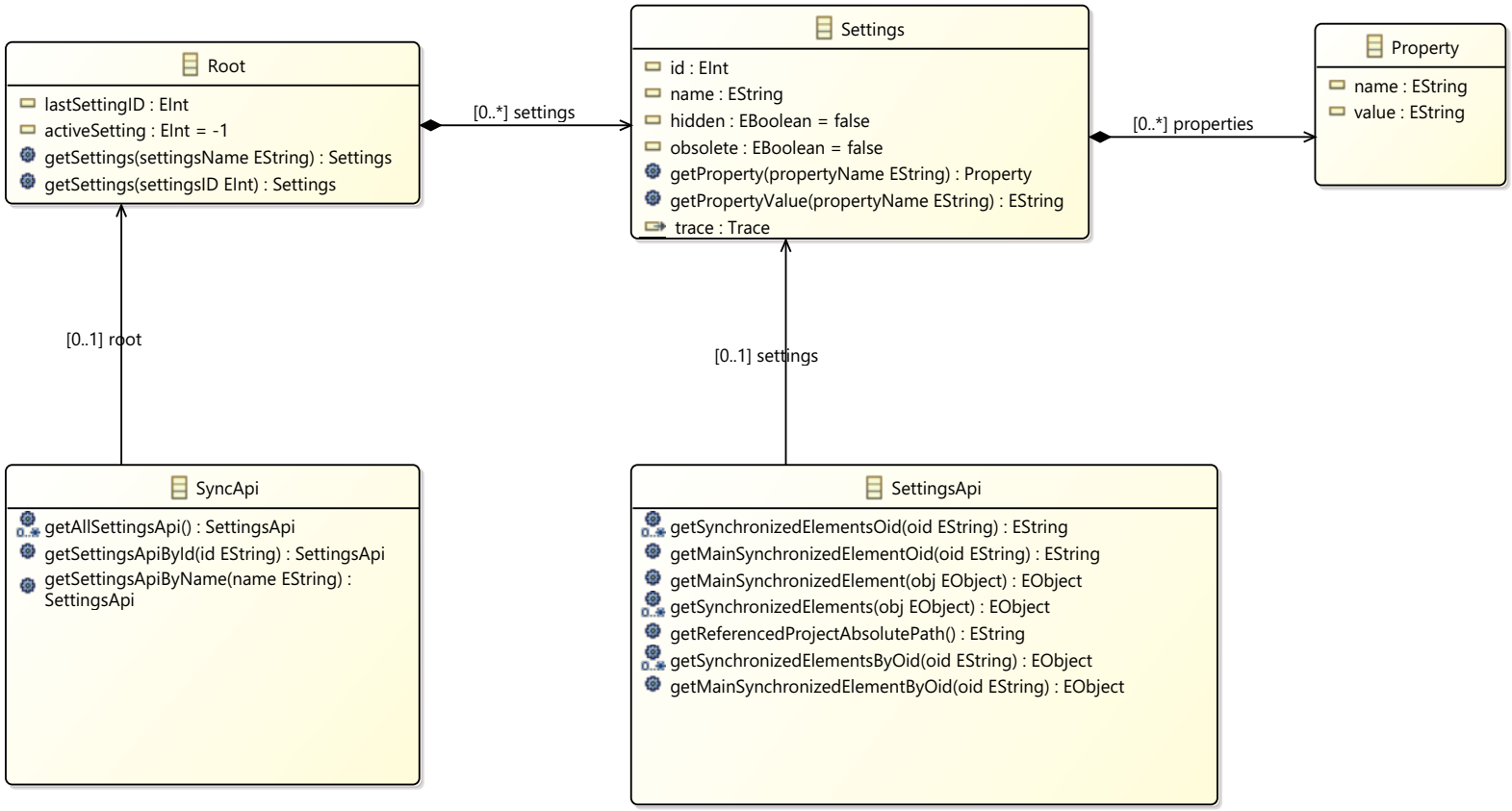
Vertex

Definition	A Vertex is an abstraction of a node in a StateMachine graph. It can be the source or destination of any number of Transitions.
Inheritance	NamedElement → Annotable
Attributes	n/a

Associations	`/incoming: Transition[*] {derived}`
	Specifies the Transitions entering this Vertex
	`/outgoing: Transition[*] {derived}`
	Specifies the Transitions departing from this Vertex
	`parent: Region`
	The Region that contains this Vertex

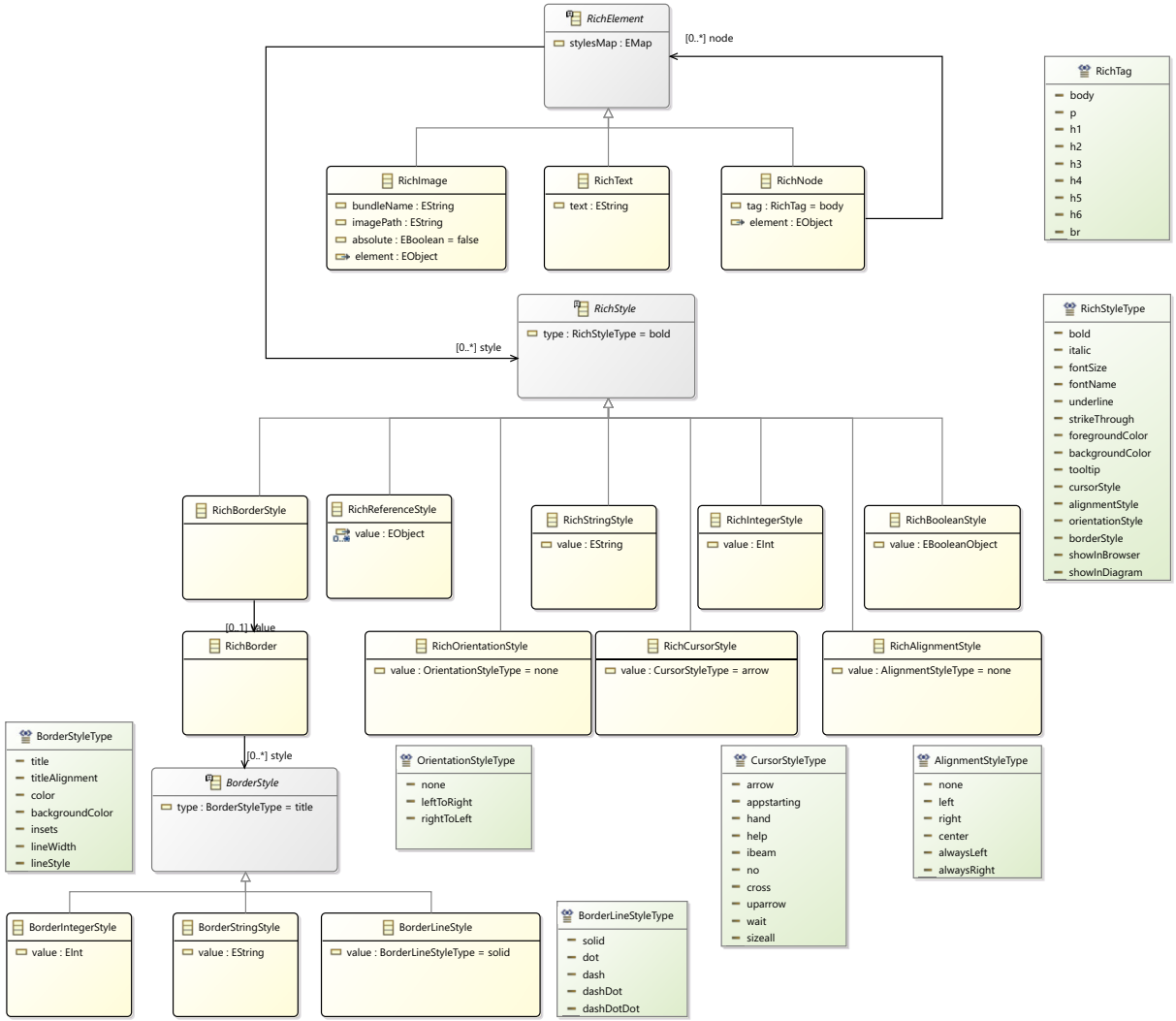
9 / Synchronization Metamodel

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:



Reference Card

Part 2

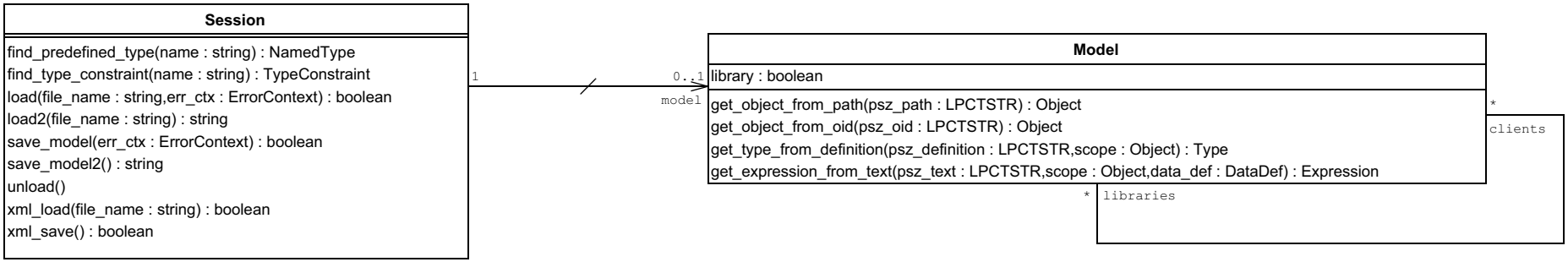
SCADE Suite Metamodels for Python API

11 /Editor Metamodels (Python API)

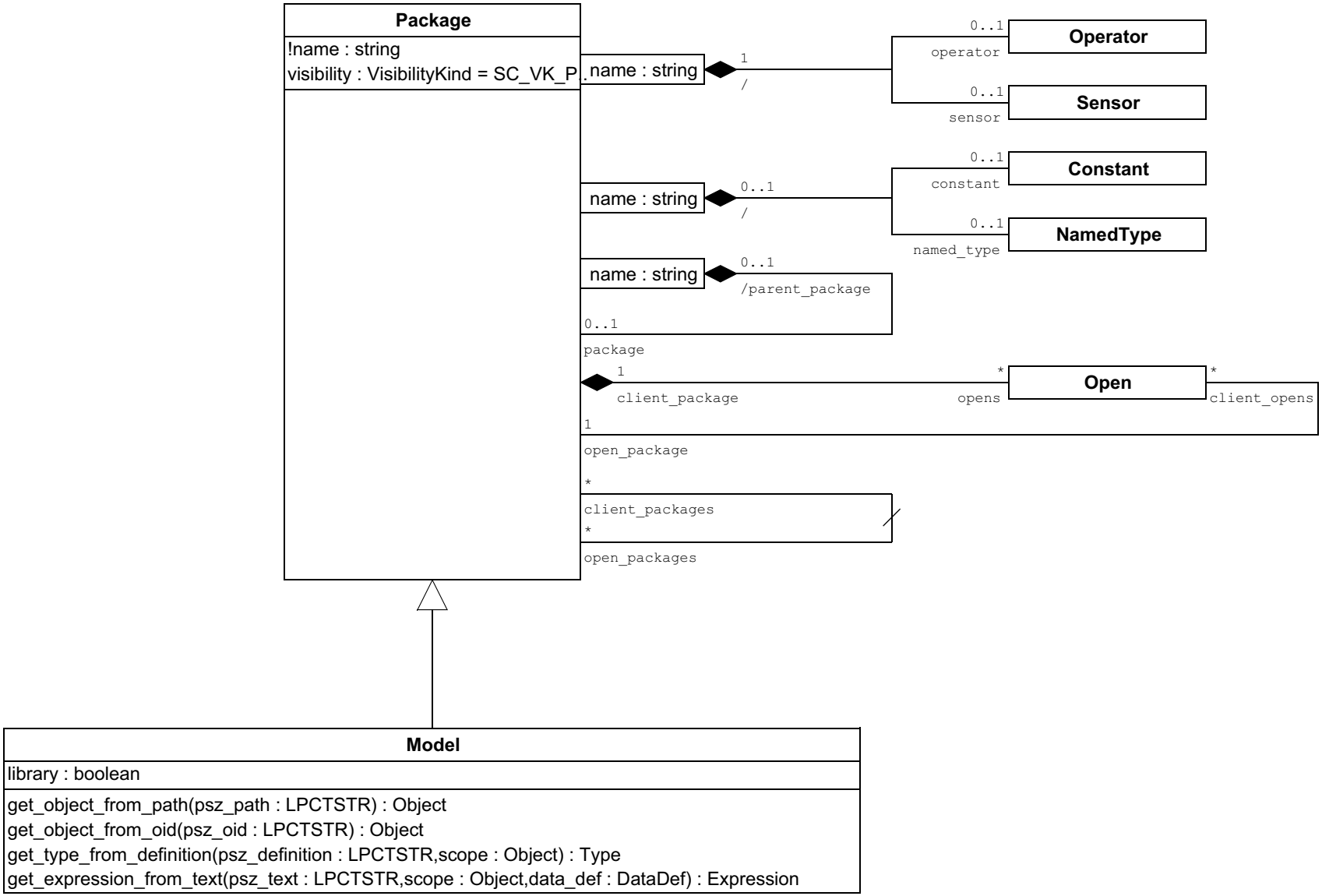
These metamodels present the data structures that give access to SCADE Suite Editor environment using Python API:

- “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)”

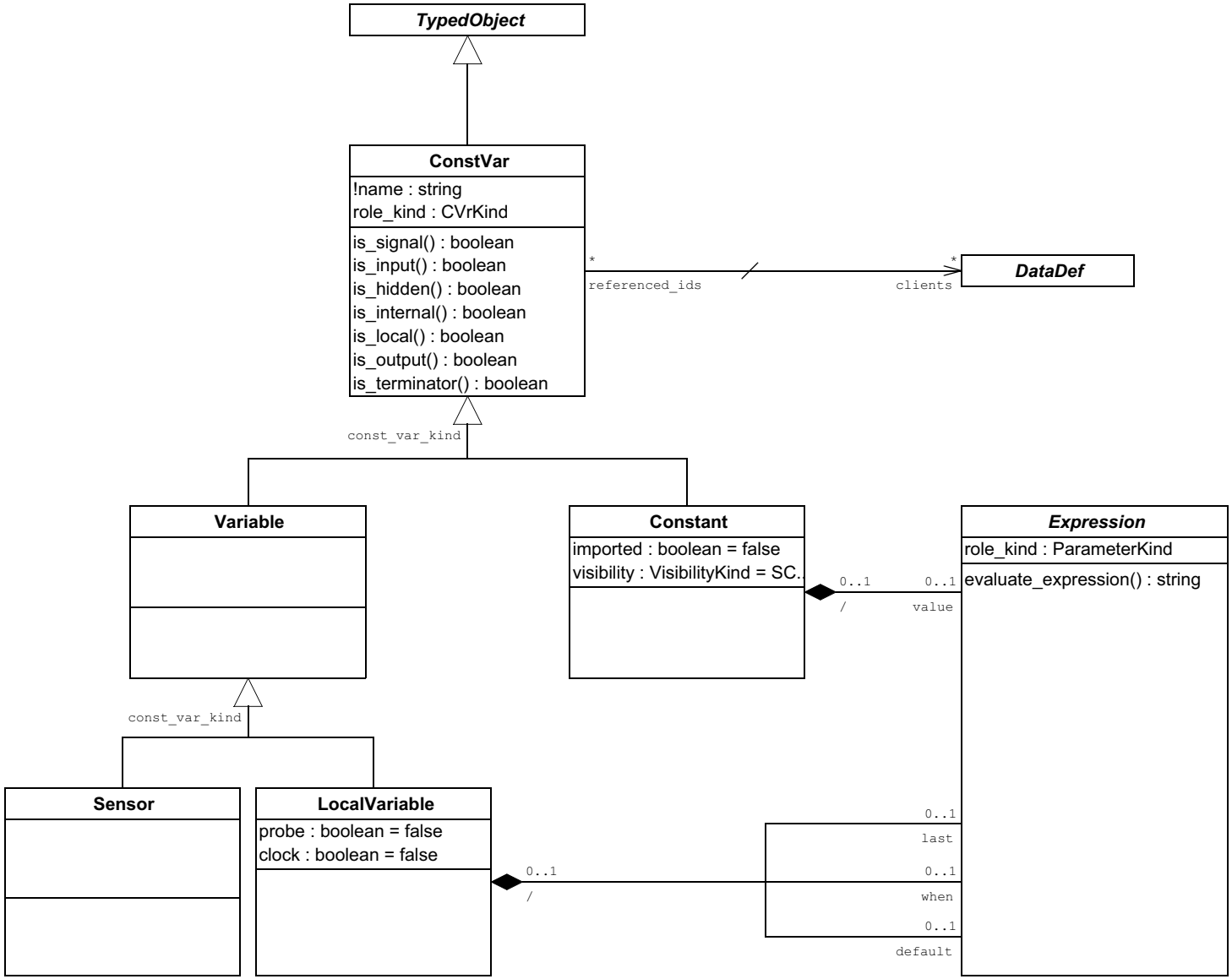
Session (Python)



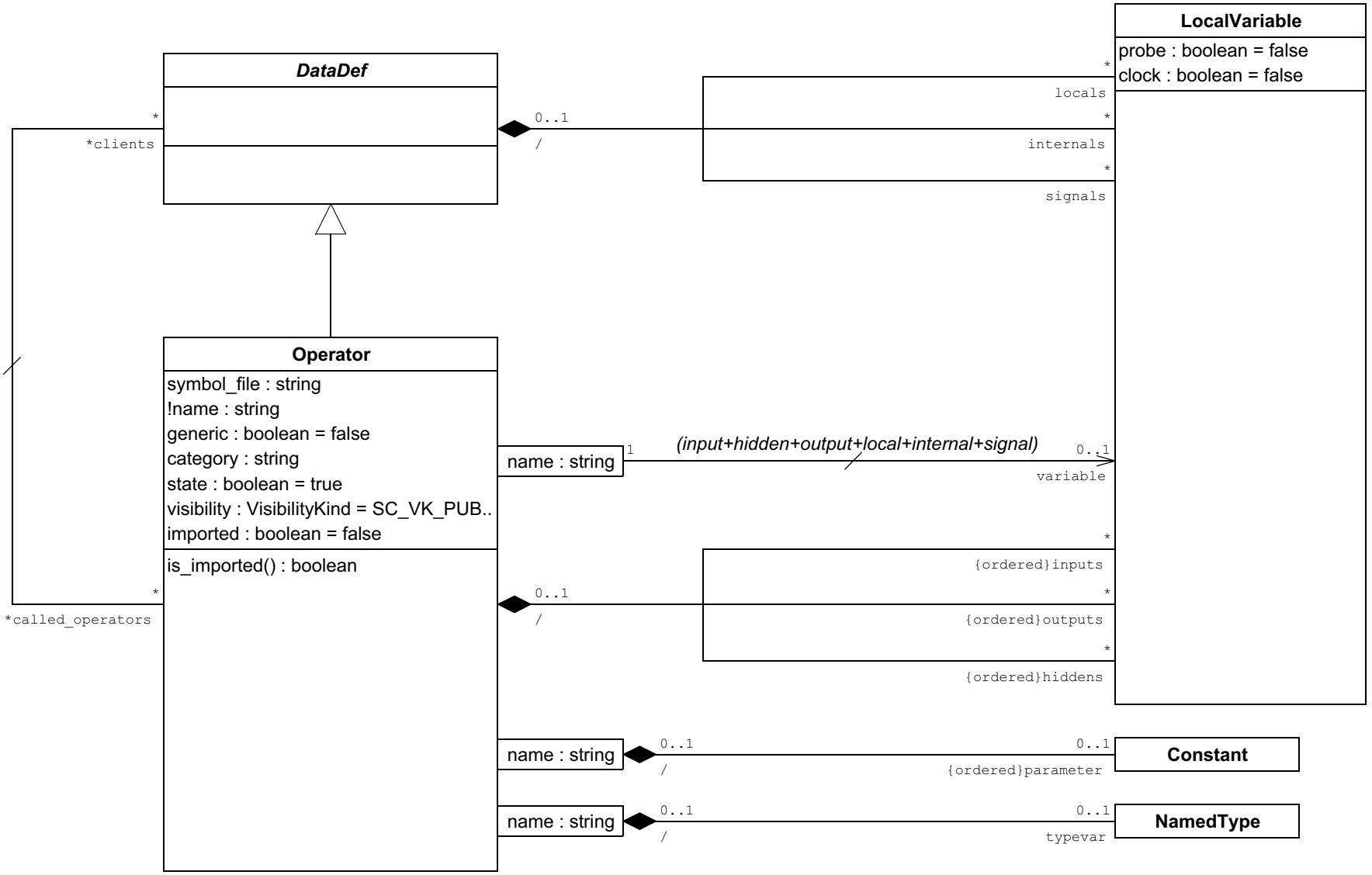
Model (Python)



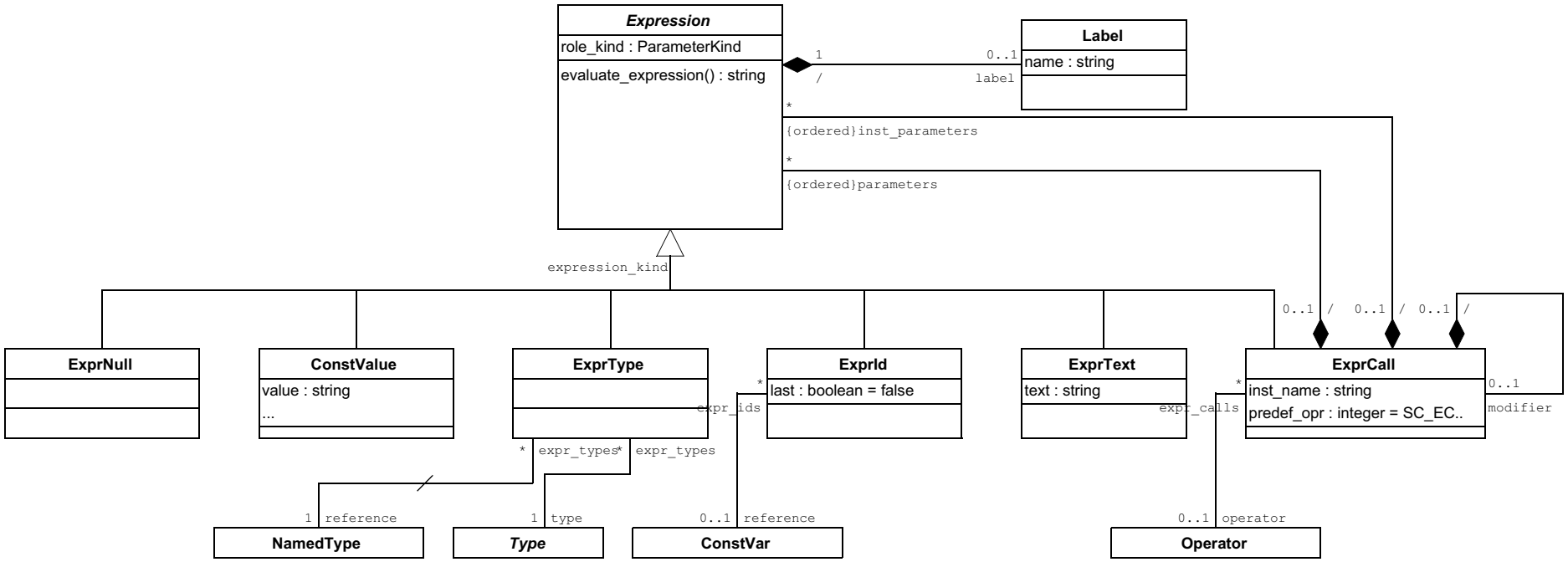
Variable (Python)



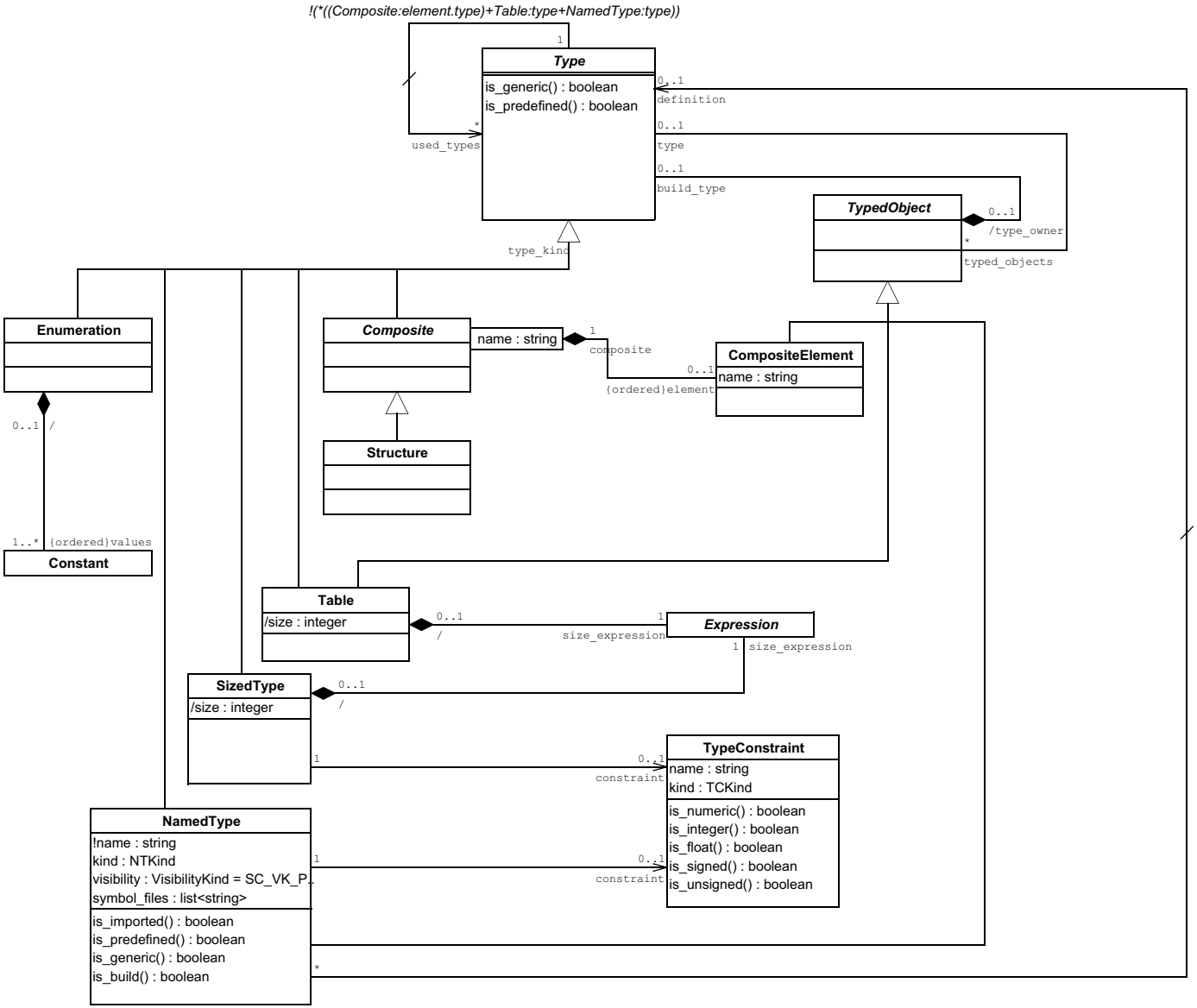
Operator (Python)



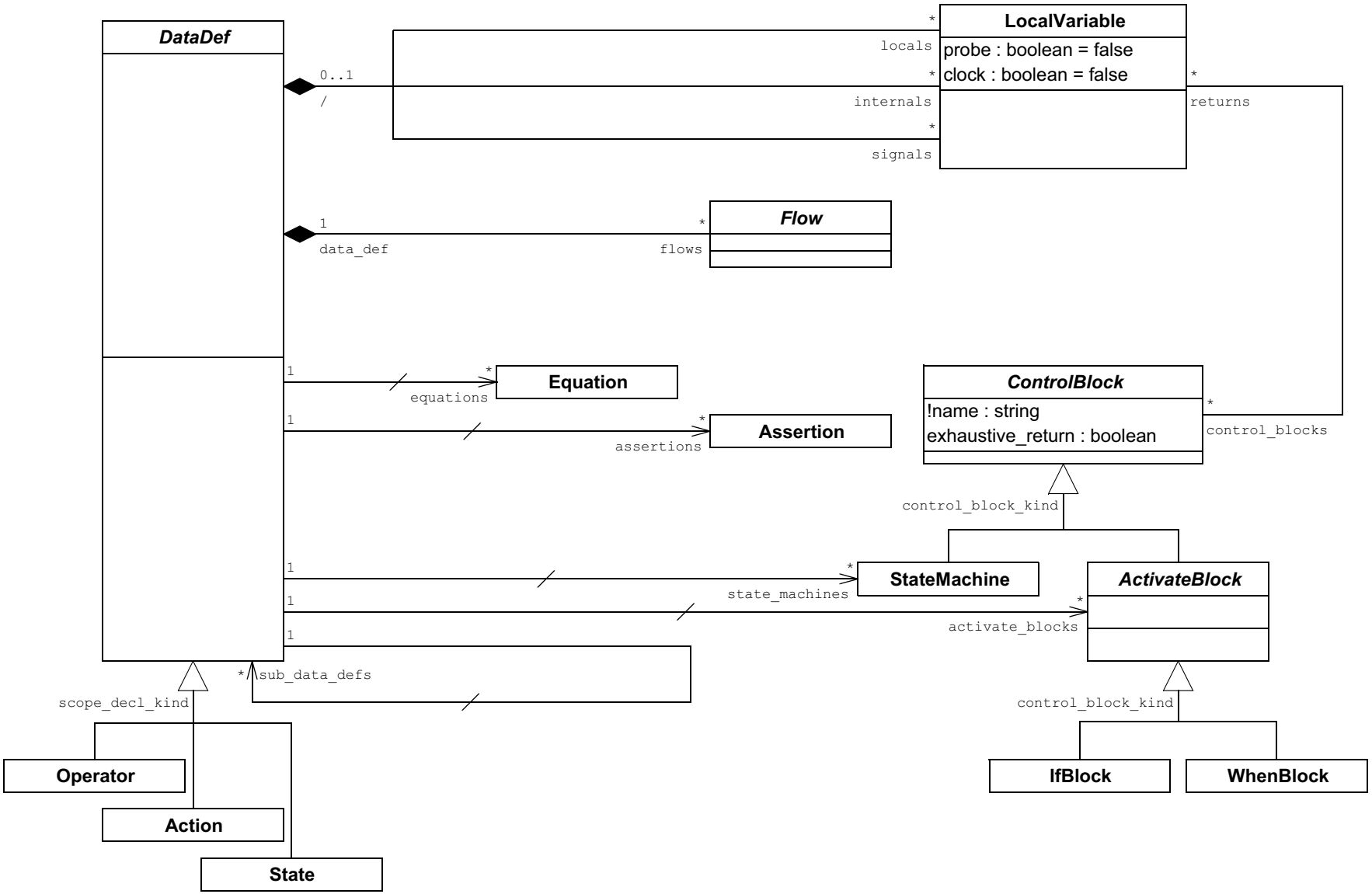
Expression (Python)



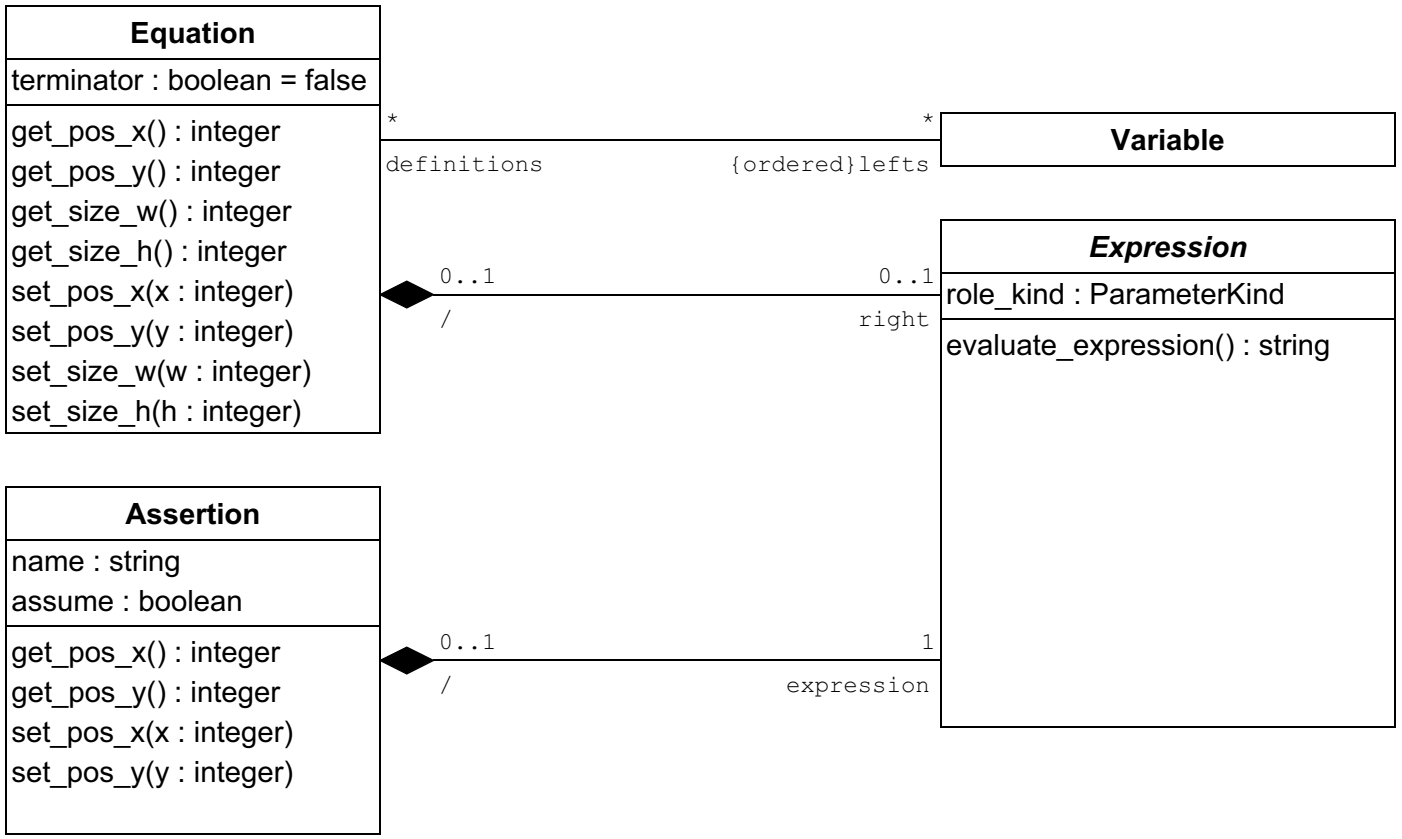
Type (Python)


!(*((Composite:element.type)+Table:type+NamedType:type))
Type
is_generic() : boolean
is_predefined() : boolean
used_types
definition
type
build_type
type_kind
TypedObject
/type_owner
typed_objects
Enumeration
Composite
name : string
composite
{ordered}element
CompositeElement
name : string
Structure
Constant
{ordered}values
Table
/size : integer
size_expression
Expression
SizedType
/size : integer
TypeConstraint
name : string
kind : TCKind
is_numeric() : boolean
is_integer() : boolean
is_float() : boolean
is_signed() : boolean
is_unsigned() : boolean
constraint
NamedType
!name : string
kind : NTKind
visibility : VisibilityKind = SC_VK_P...
symbol_files : list<string>
is_imported() : boolean
is_predefined() : boolean
is_generic() : boolean
is_build() : boolean

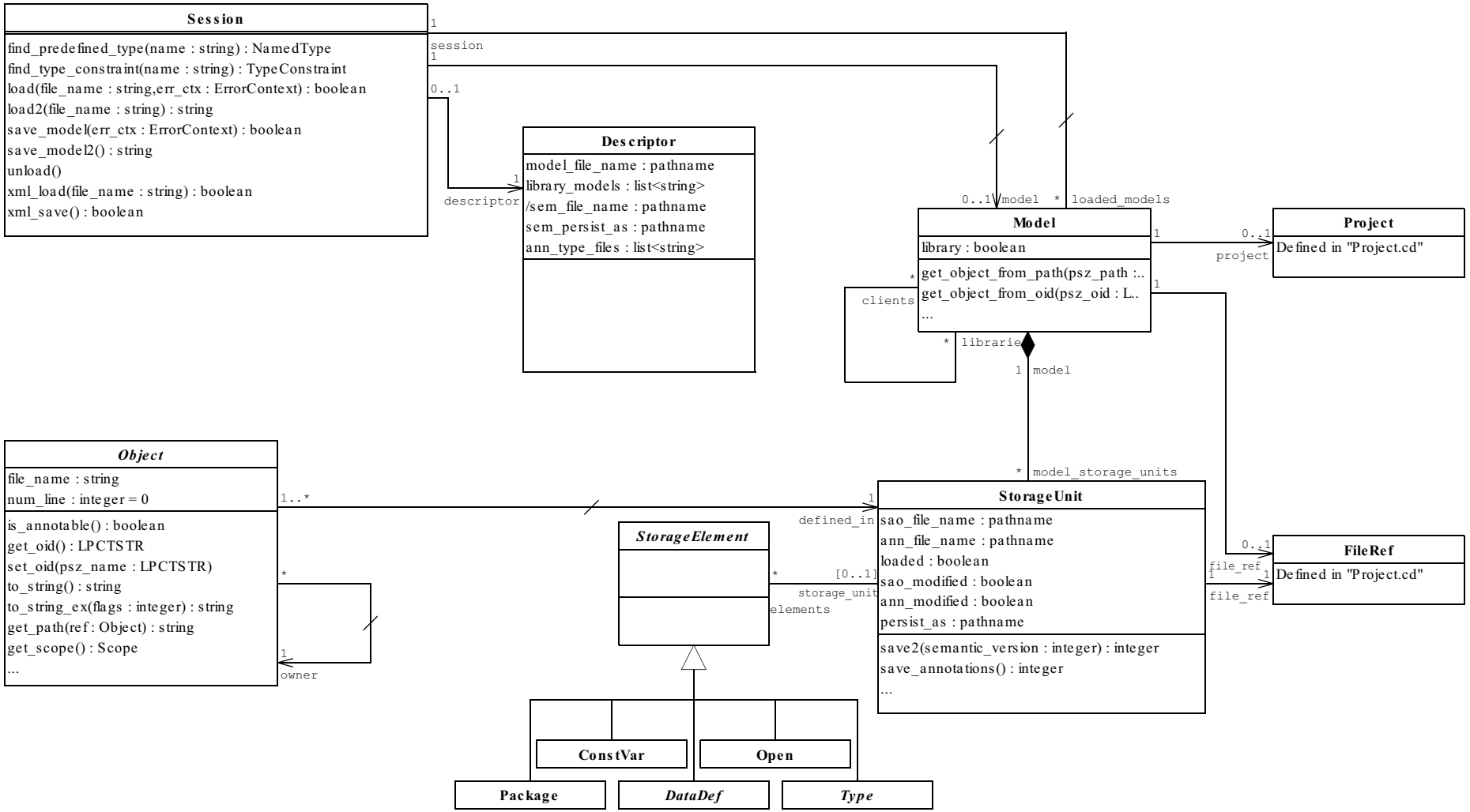
Data Definition (Python)



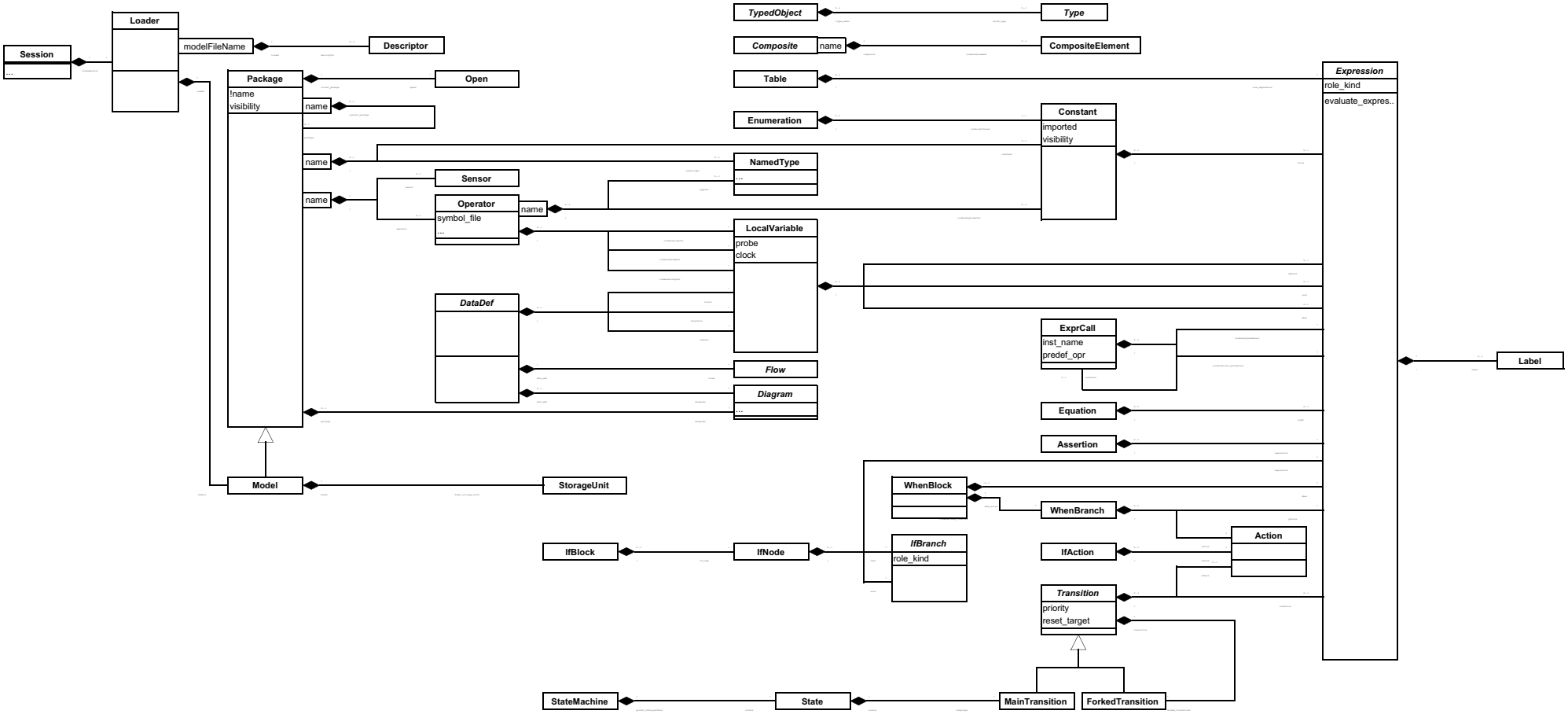
Data Flow (Python)



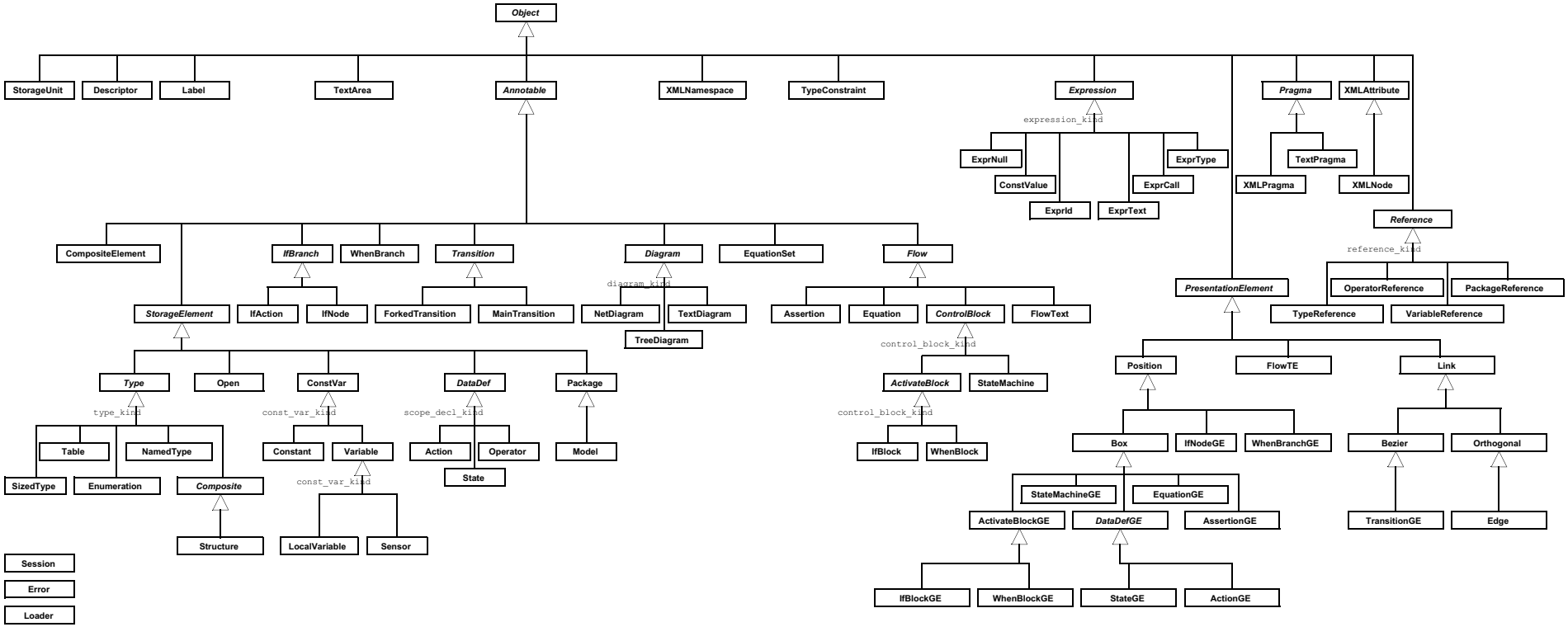
Model Storage - IO (Python)



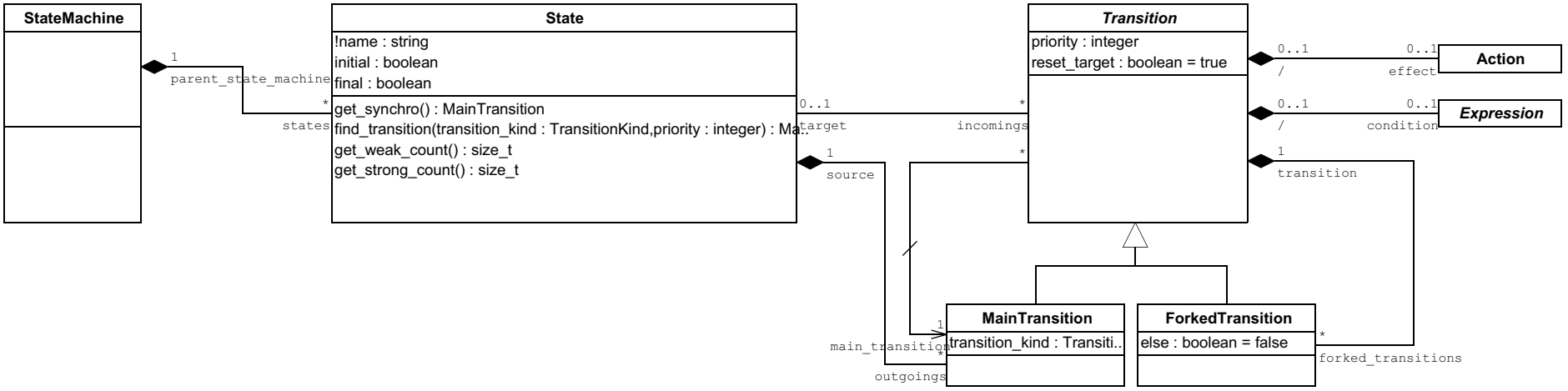
Composition (Python)



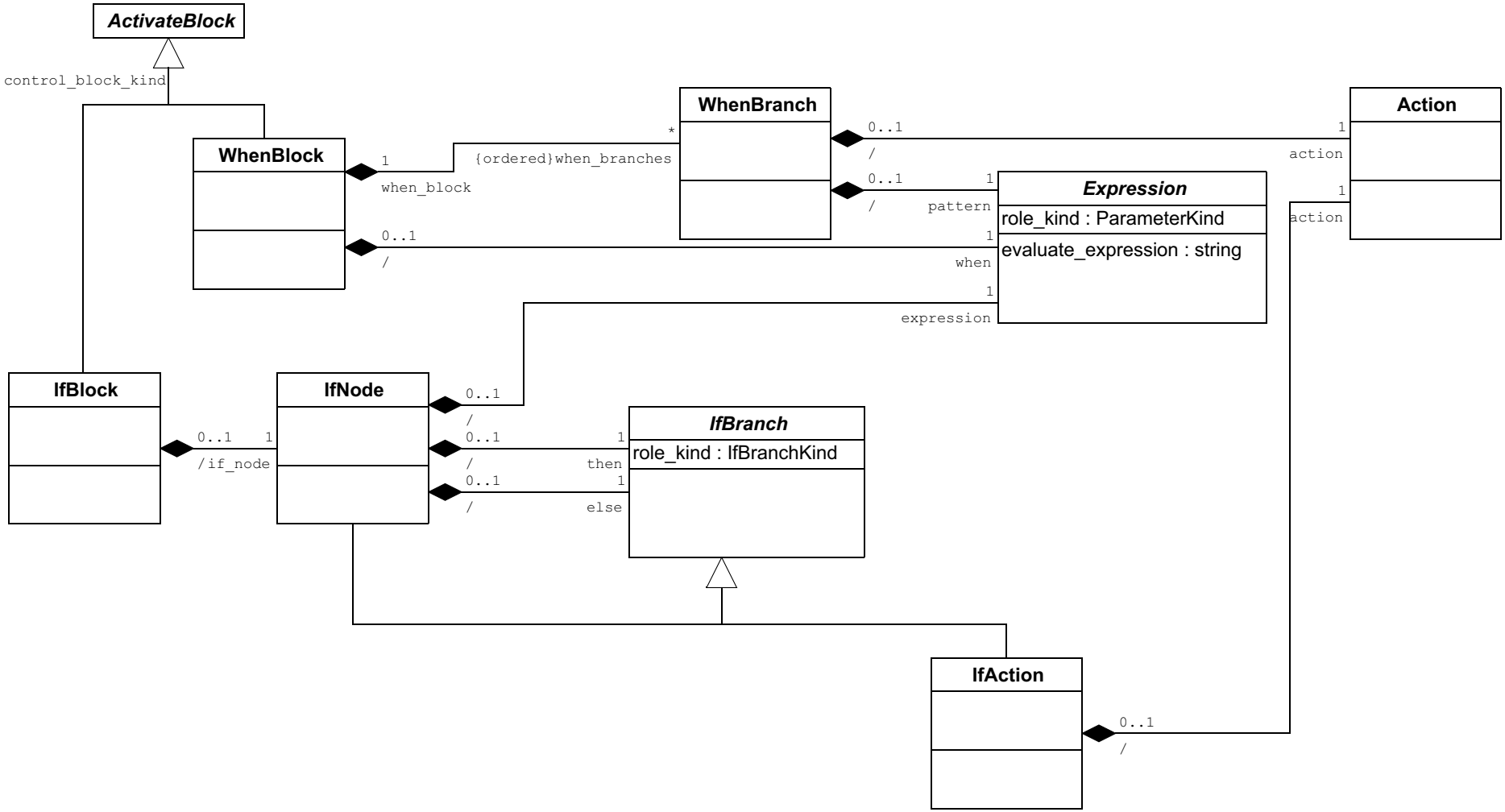
Inheritance (Python)



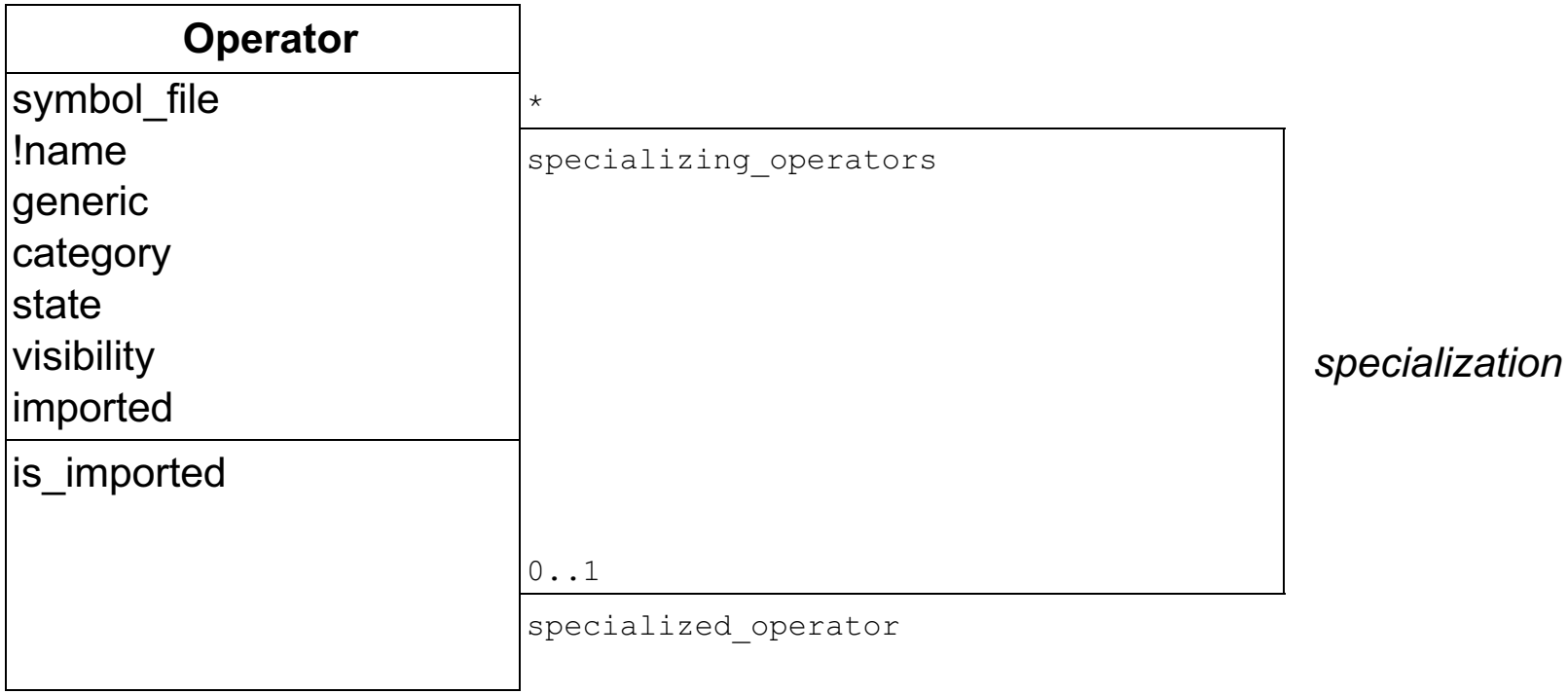
State Machine (Python)



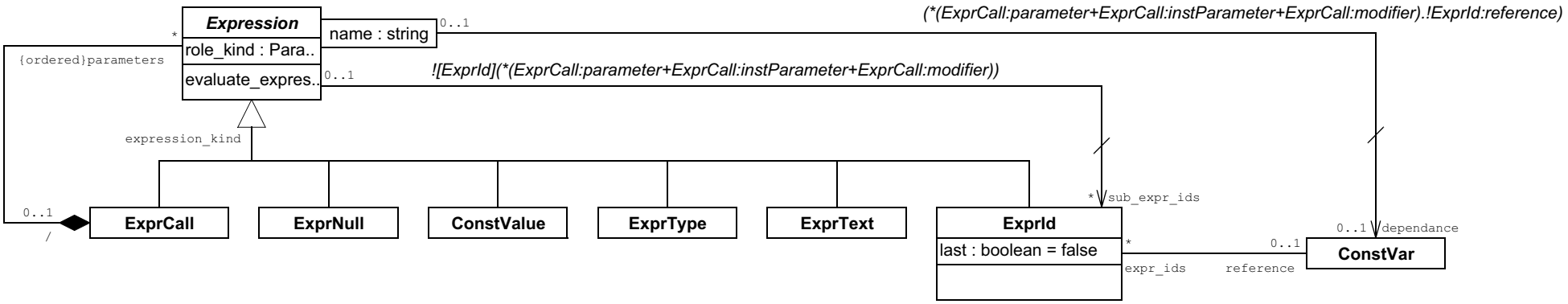
ActivateBlock (Python)



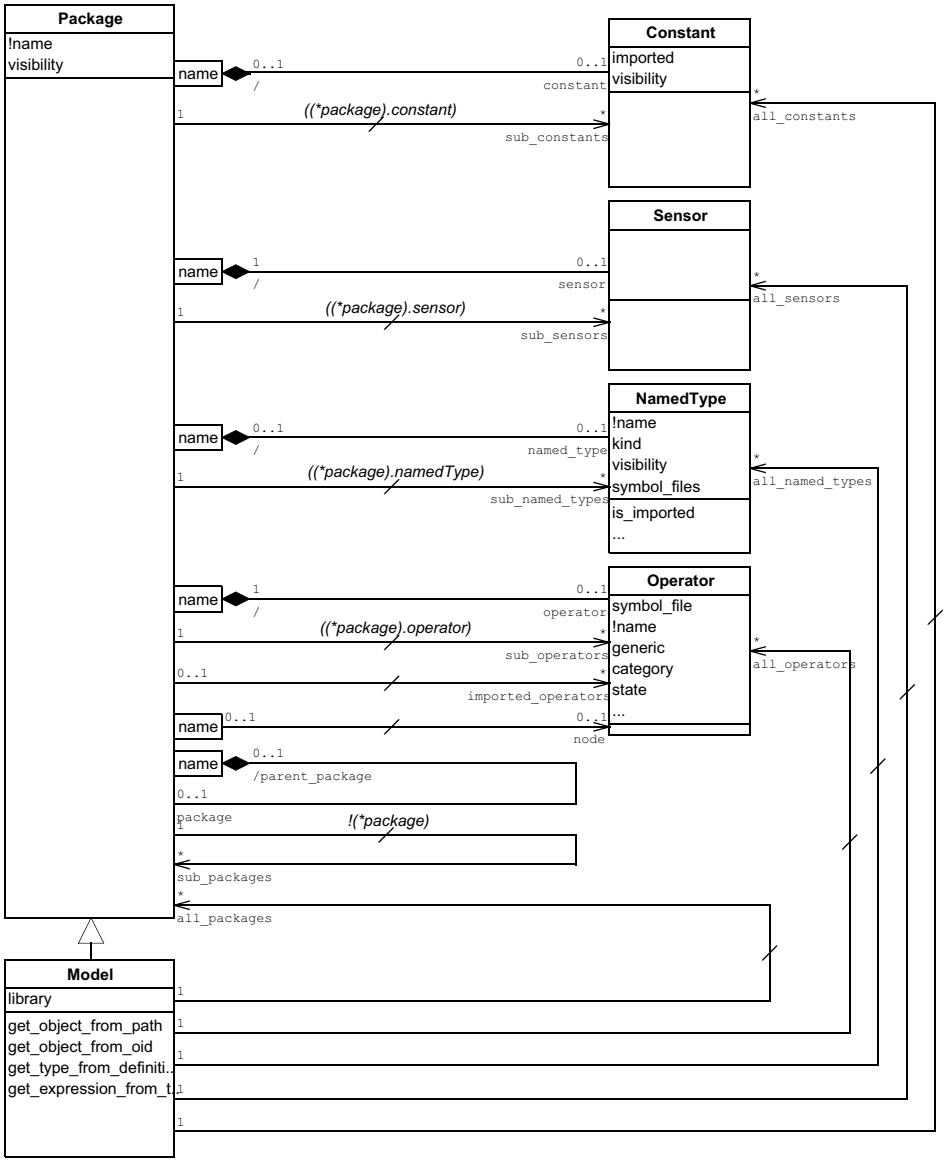
Specialization (Python)



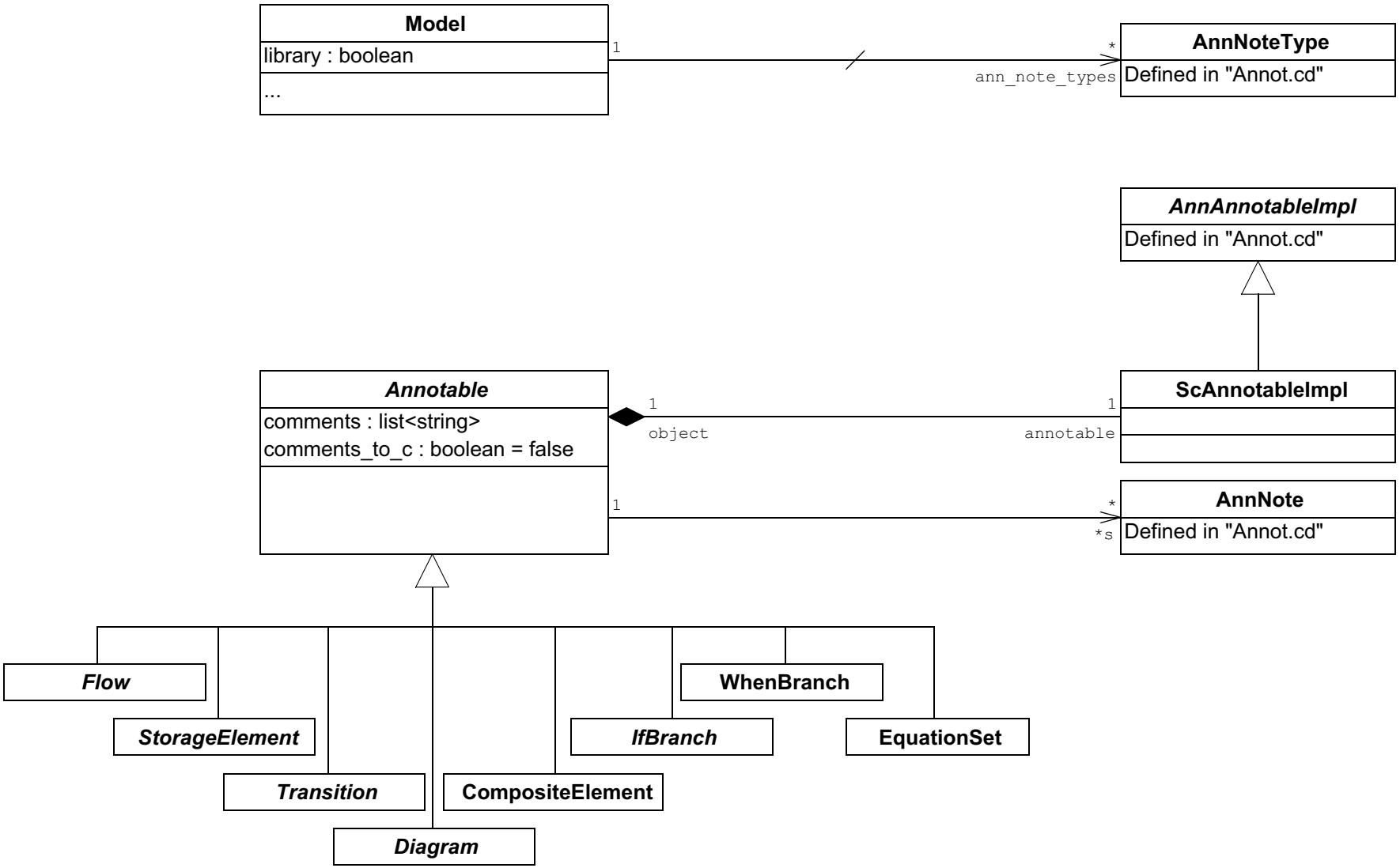
Expression Map (Python)



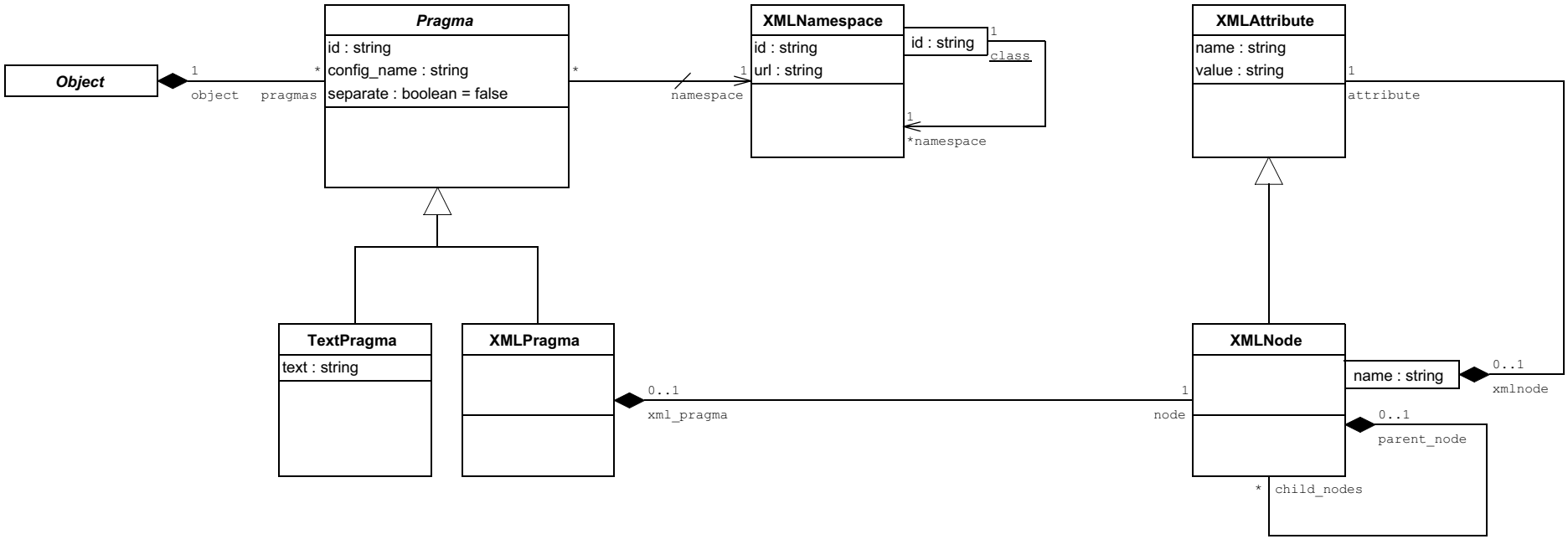
Model and Package Maps (Python)



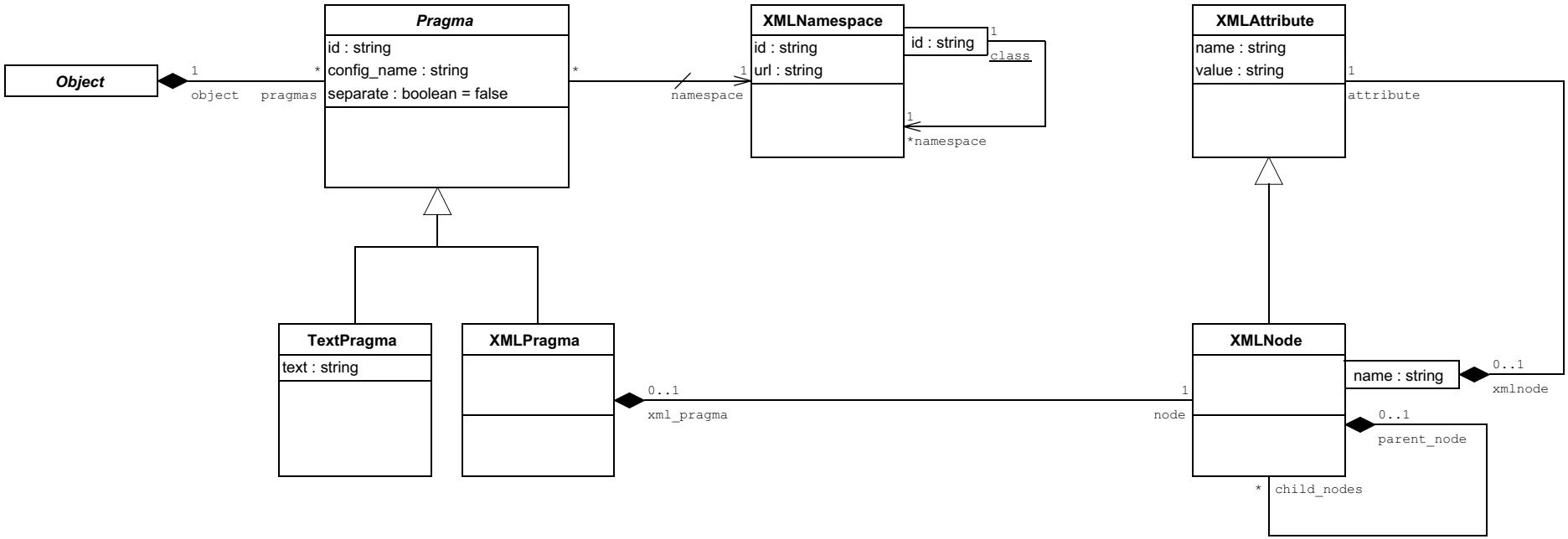
Annotable (Python)



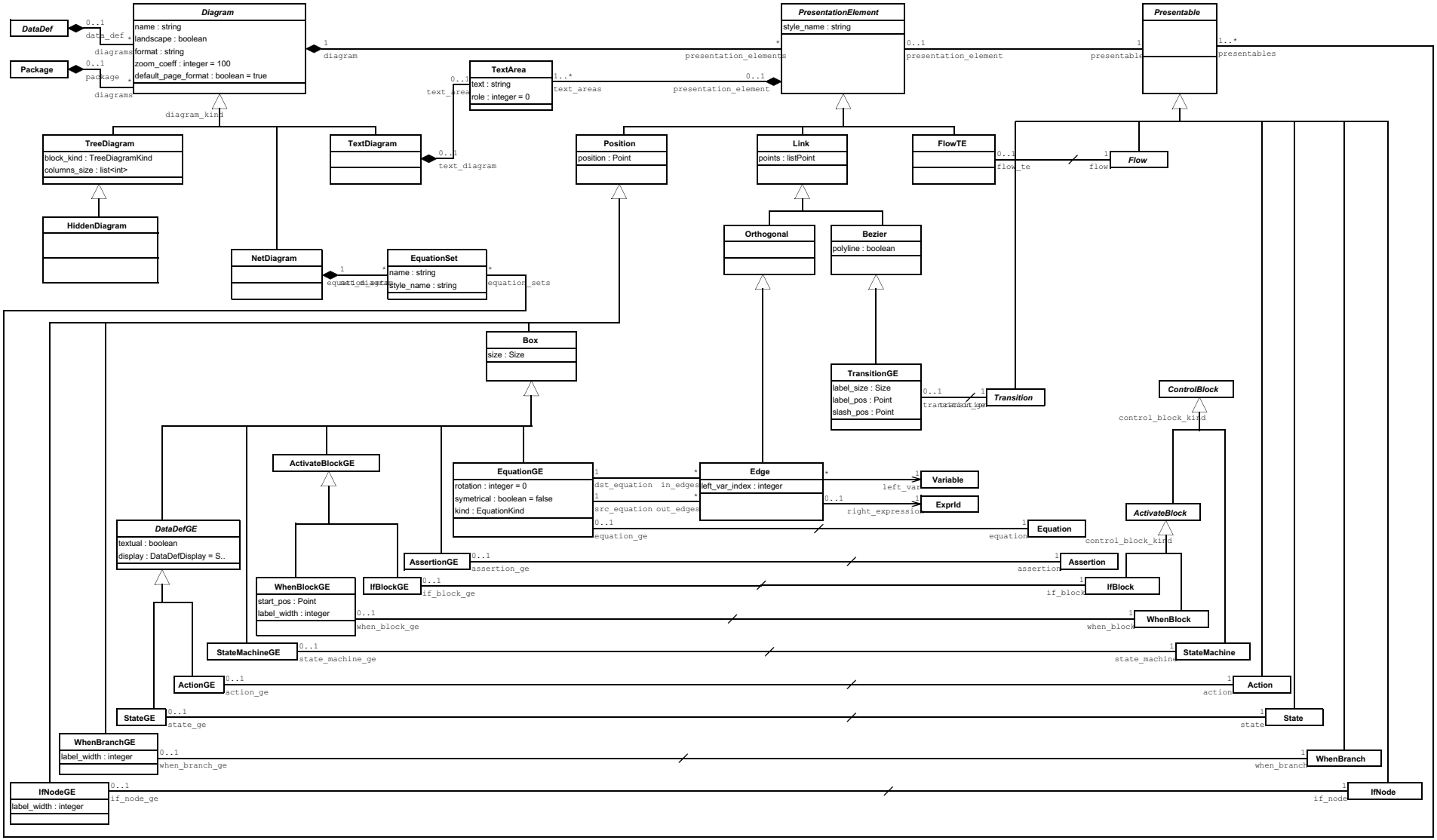
Pragmas (Python)



Scopes (Python)



Graphical (Python)

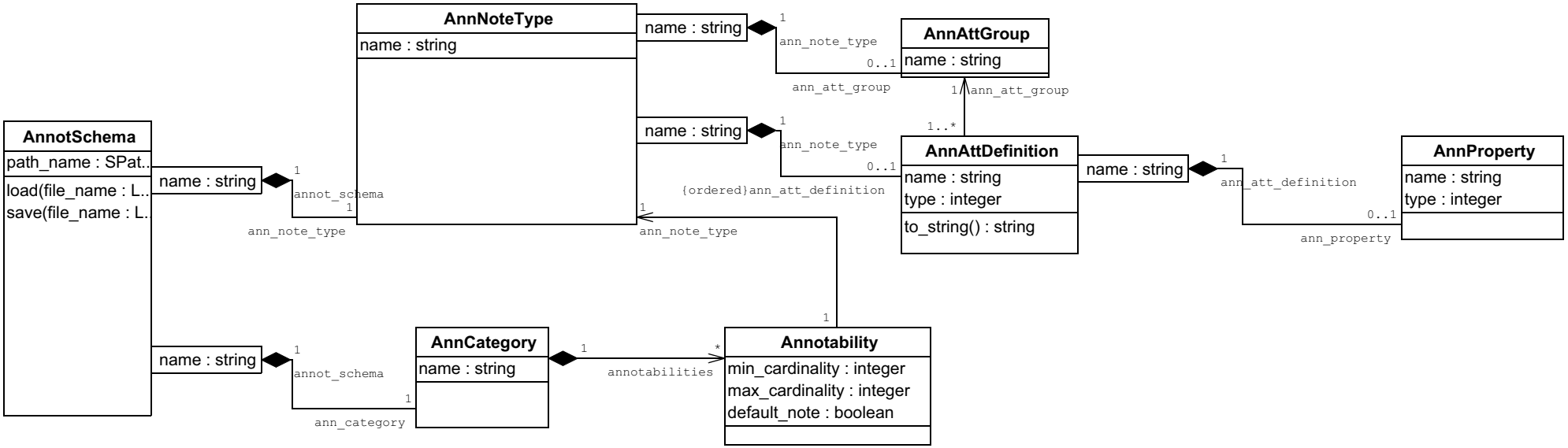


12 / Annotation Metamodels (Python API)

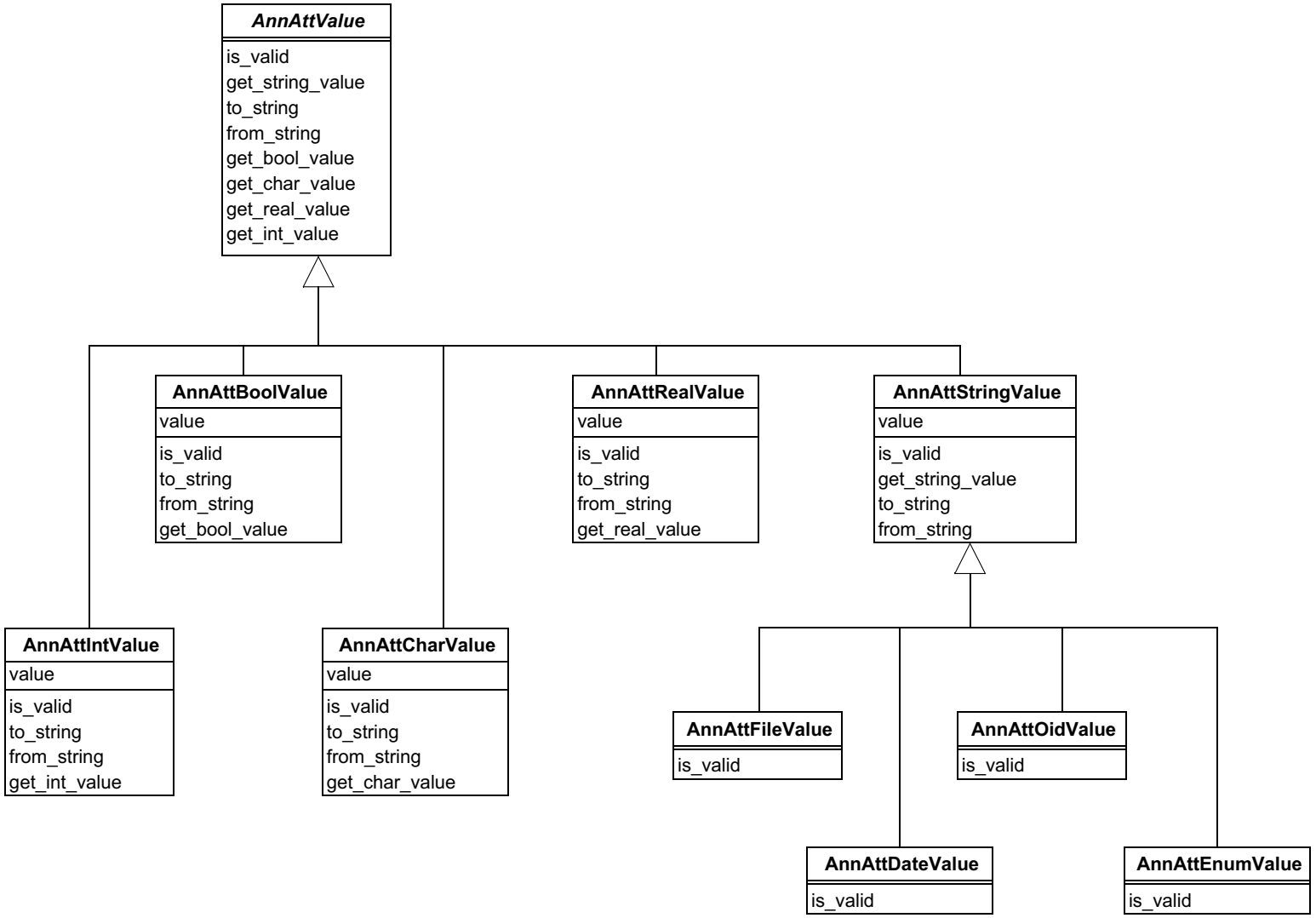
These metamodels present the data structures that give access to SCADE Suite annotations mechanisms using Python API:

- "Note Types (Python)"
- "Notes Attribute Values (Python)"
- "Notes Property Values (Python)"

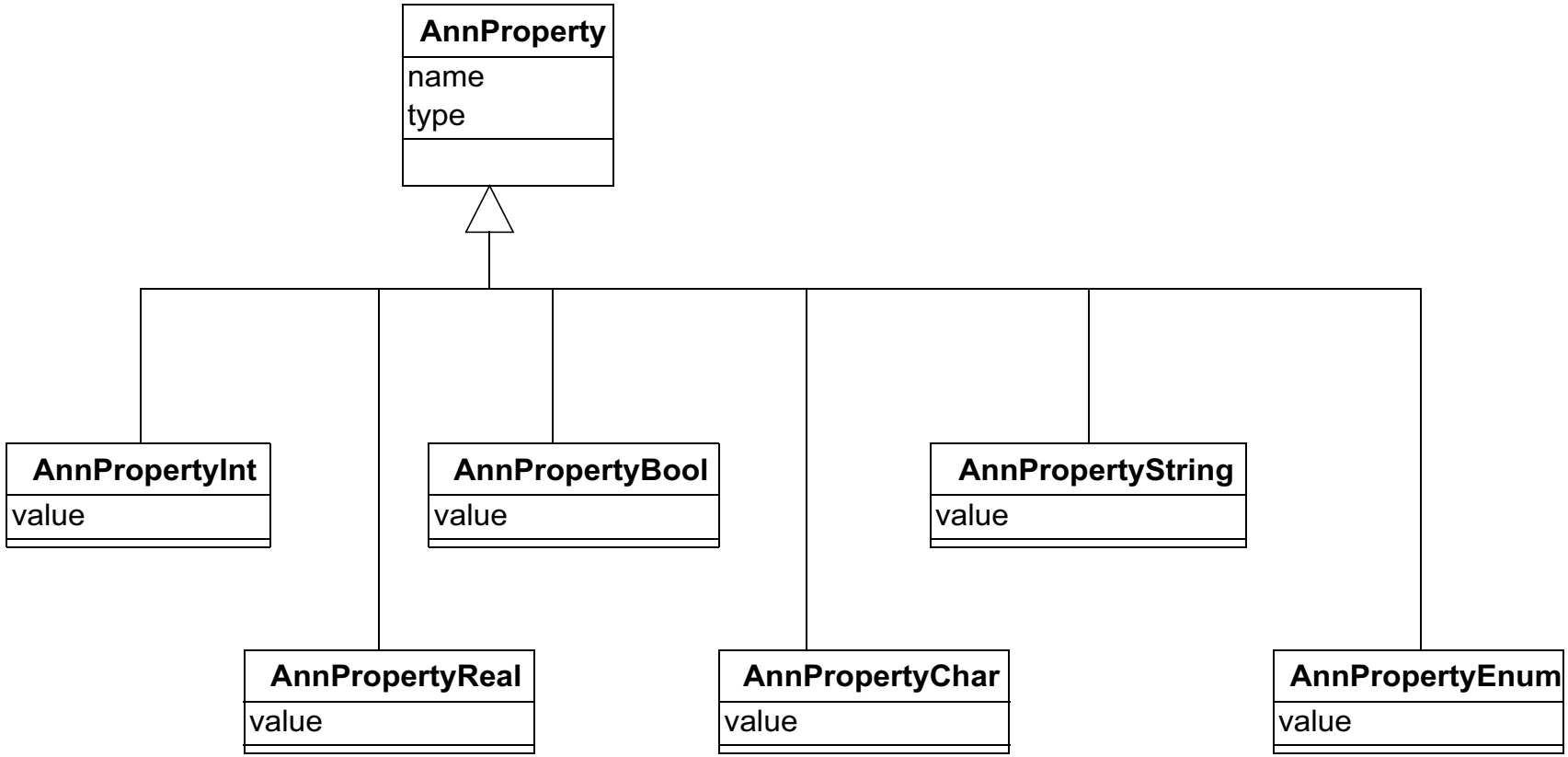
Note Types (Python)



Notes Attribute Values (Python)

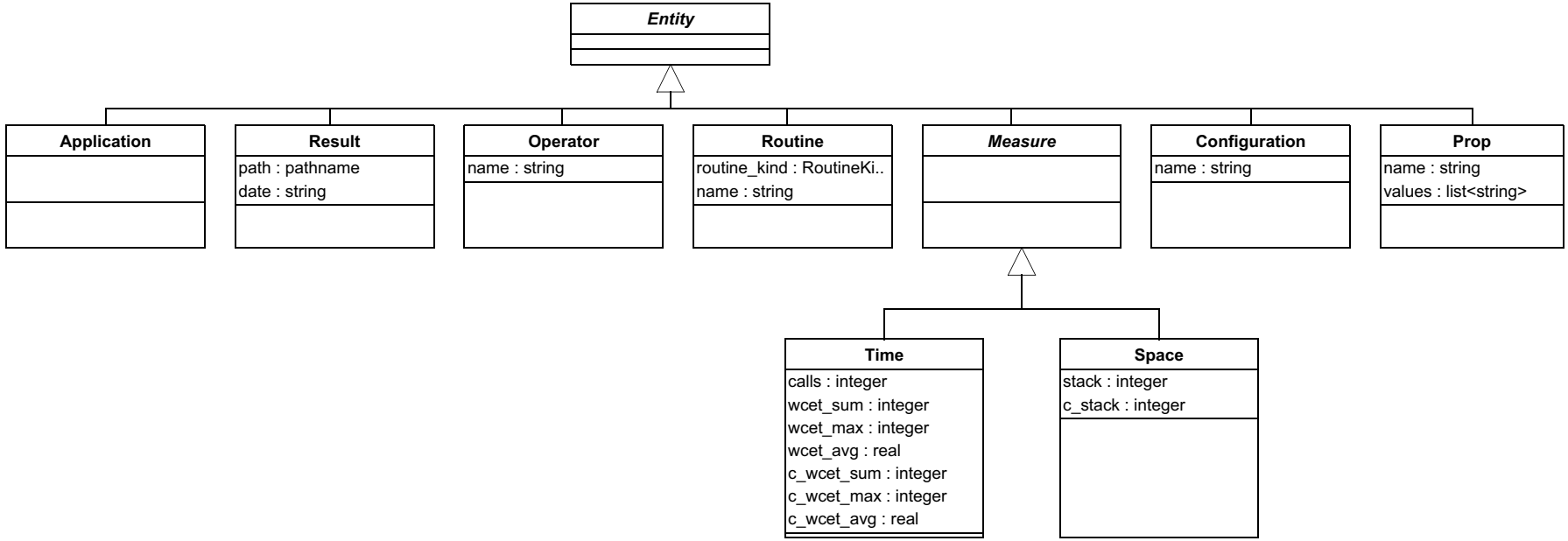


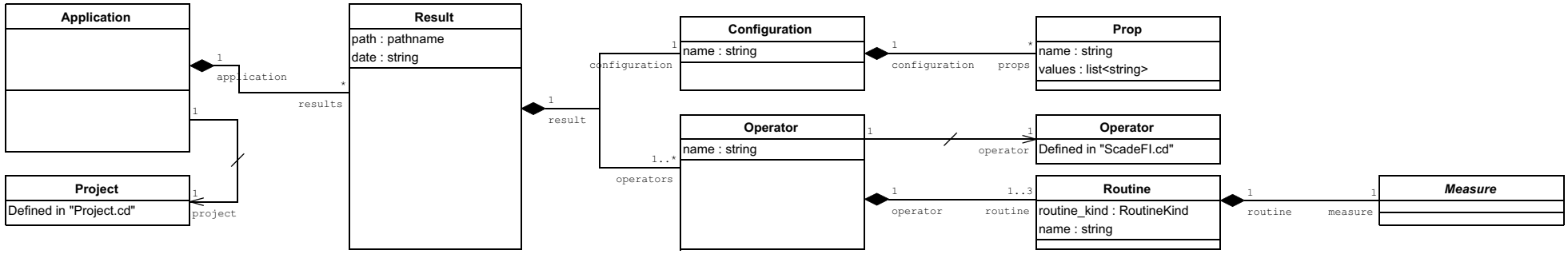
Notes Property Values (Python)



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:



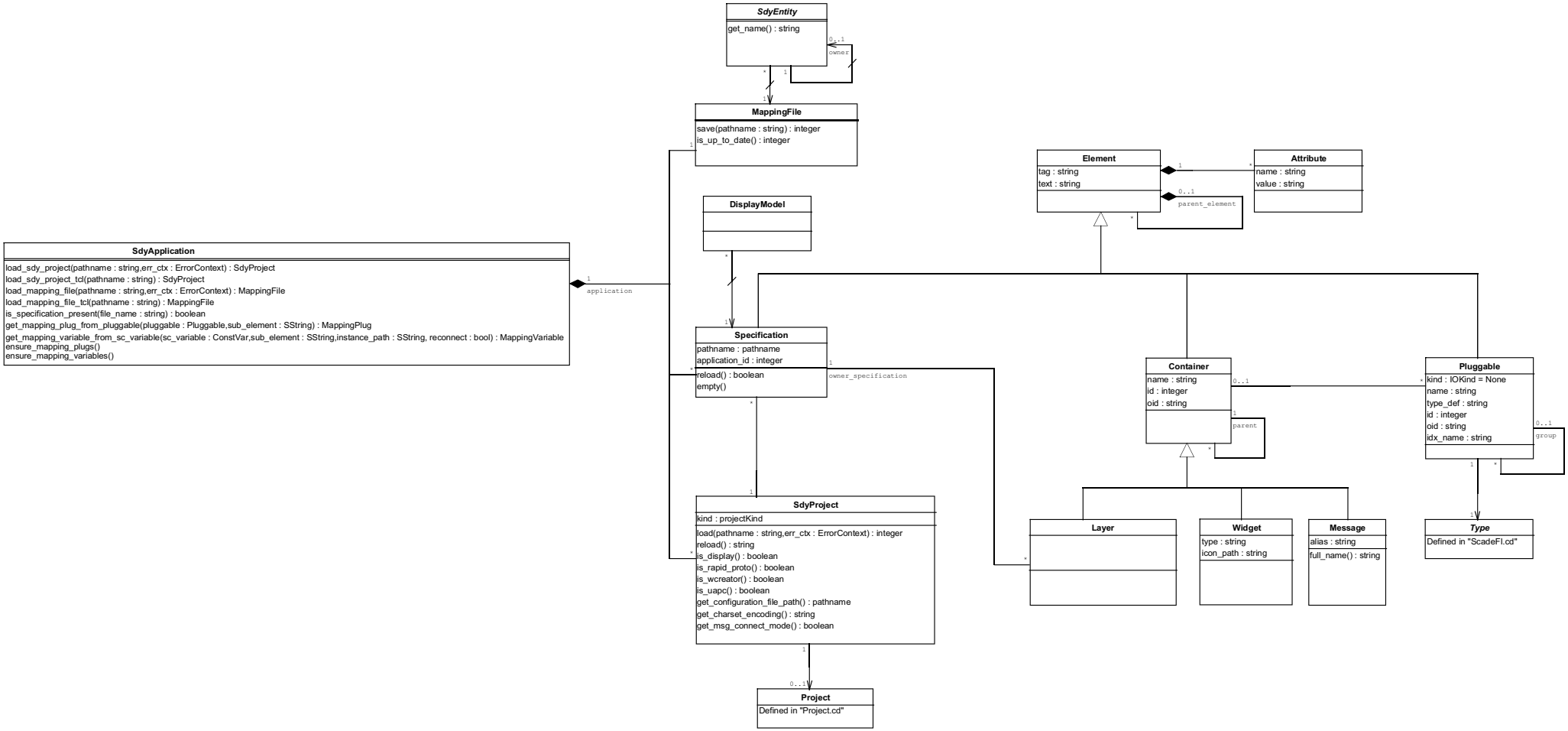


14 / Graphical Panel Coupling Metamodel (Python API)

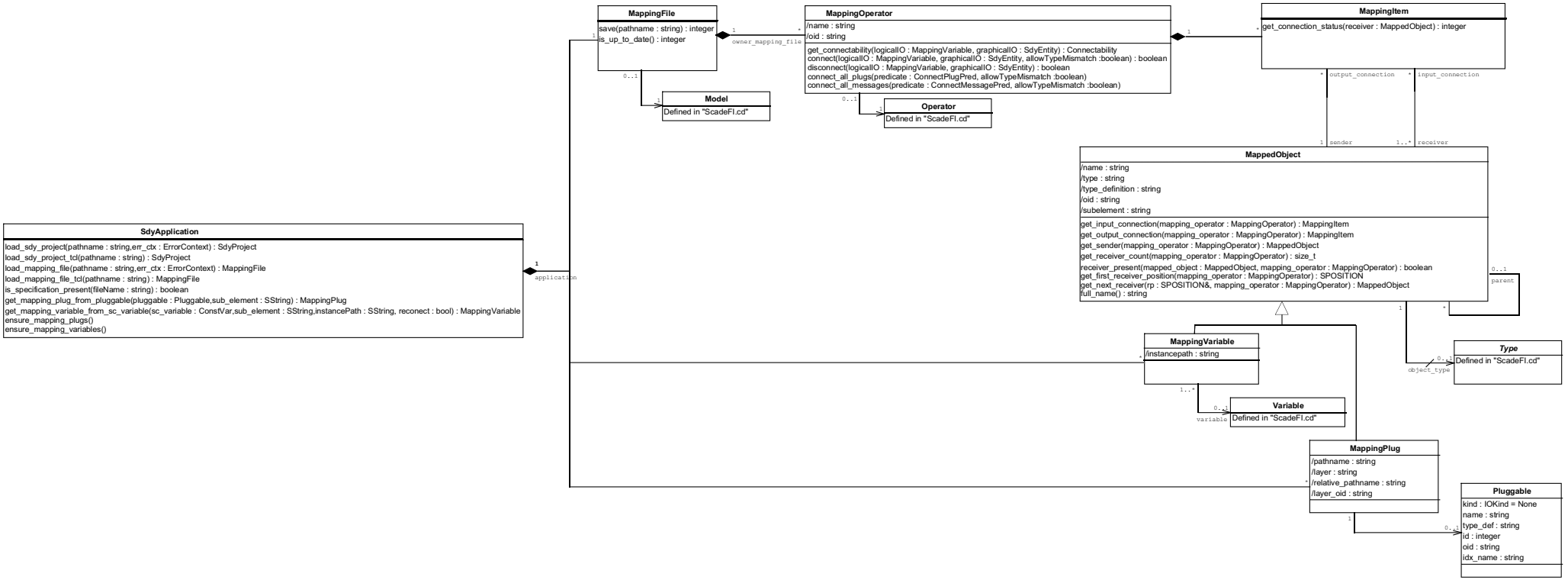
These metamodels present the data structures that give access to connection data using SCADE Display Coupling Python API:

- "Graphical Panel Coupling (main)"
- "Graphical Panel Coupling (mapping file)"
- "Graphical Panel Coupling (mapping plug for ARINC 661)"
- "Graphical Panel Coupling (inheritance)"

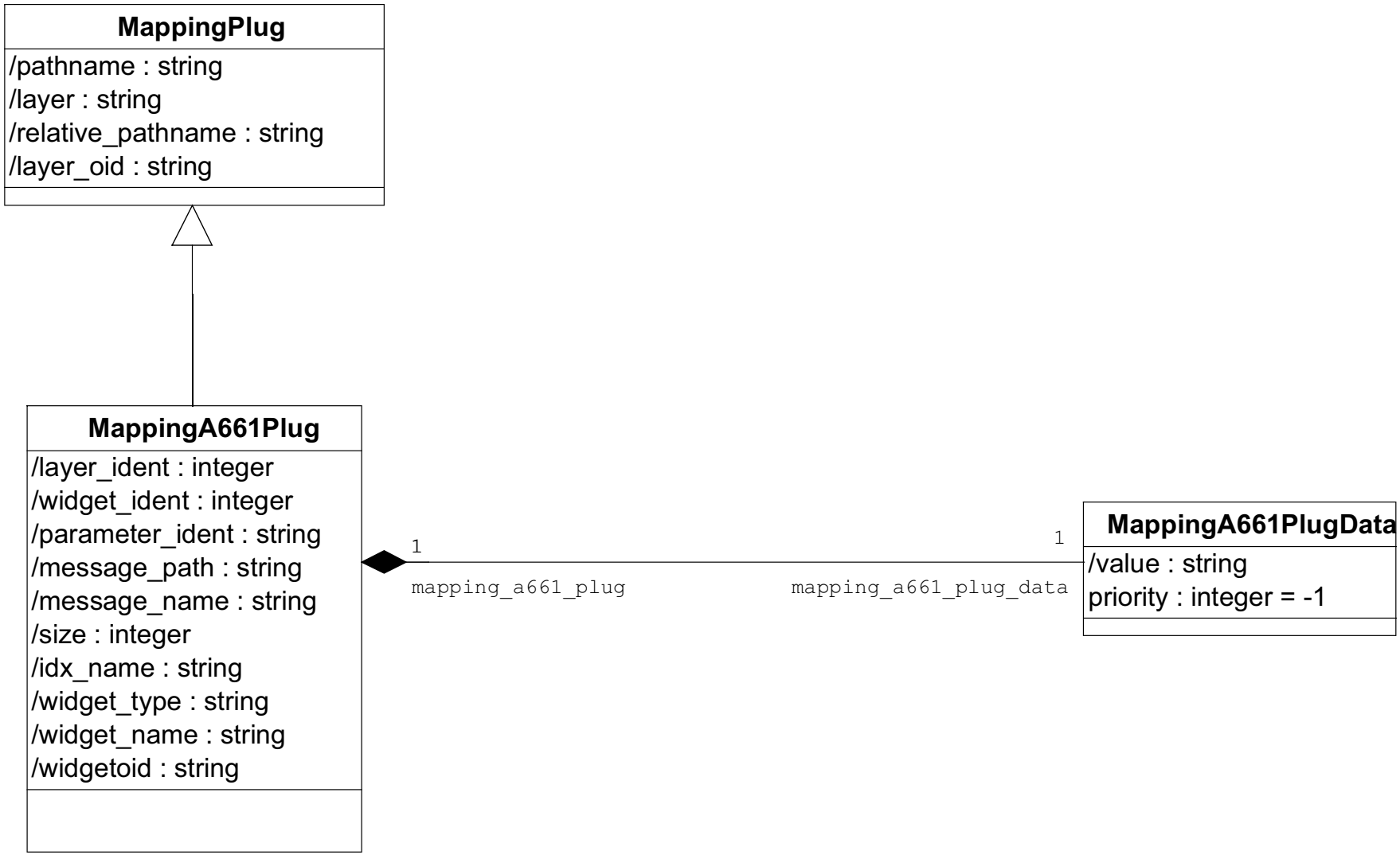
Graphical Panel Coupling (main)



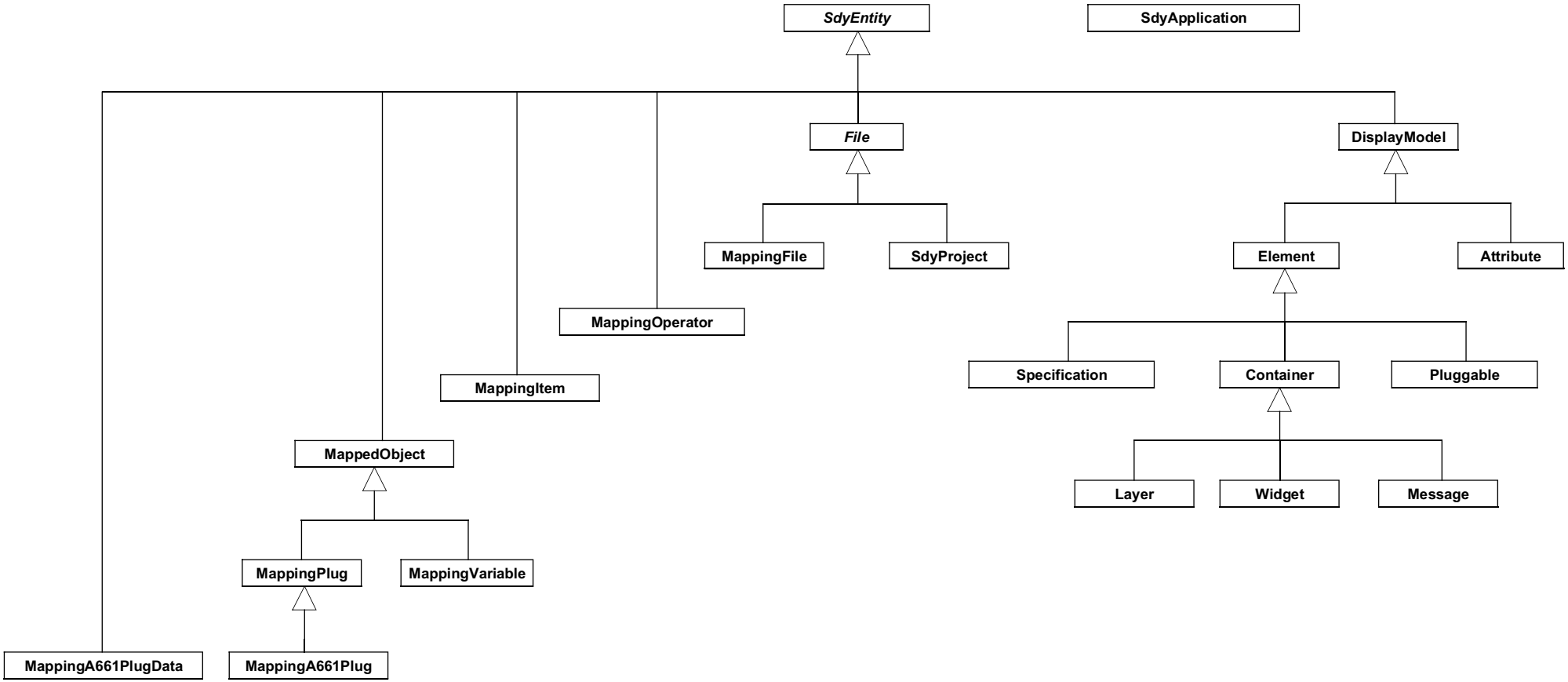
Graphical Panel Coupling (mapping file)



Graphical Panel Coupling (mapping plug for ARINC 661)



Graphical Panel Coupling (inheritance)



Reference Card

Part 3

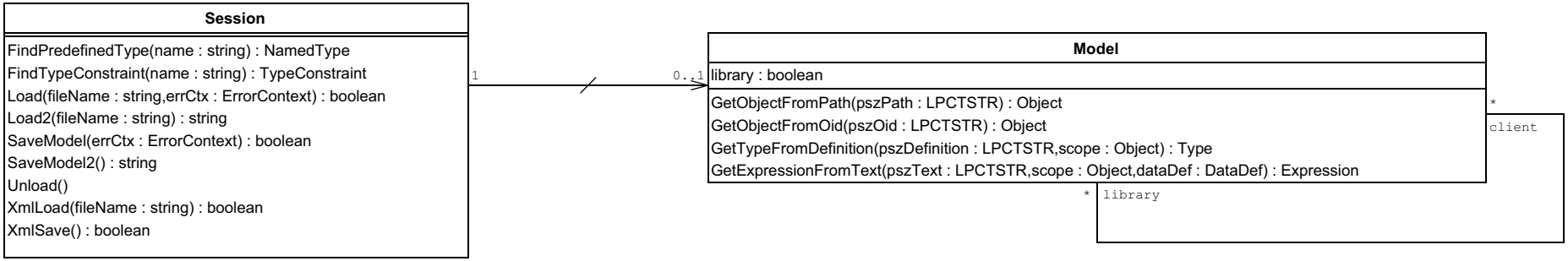
SCADE Suite Metamodels for Tcl API

15 / Editor Metamodels (Tcl API)

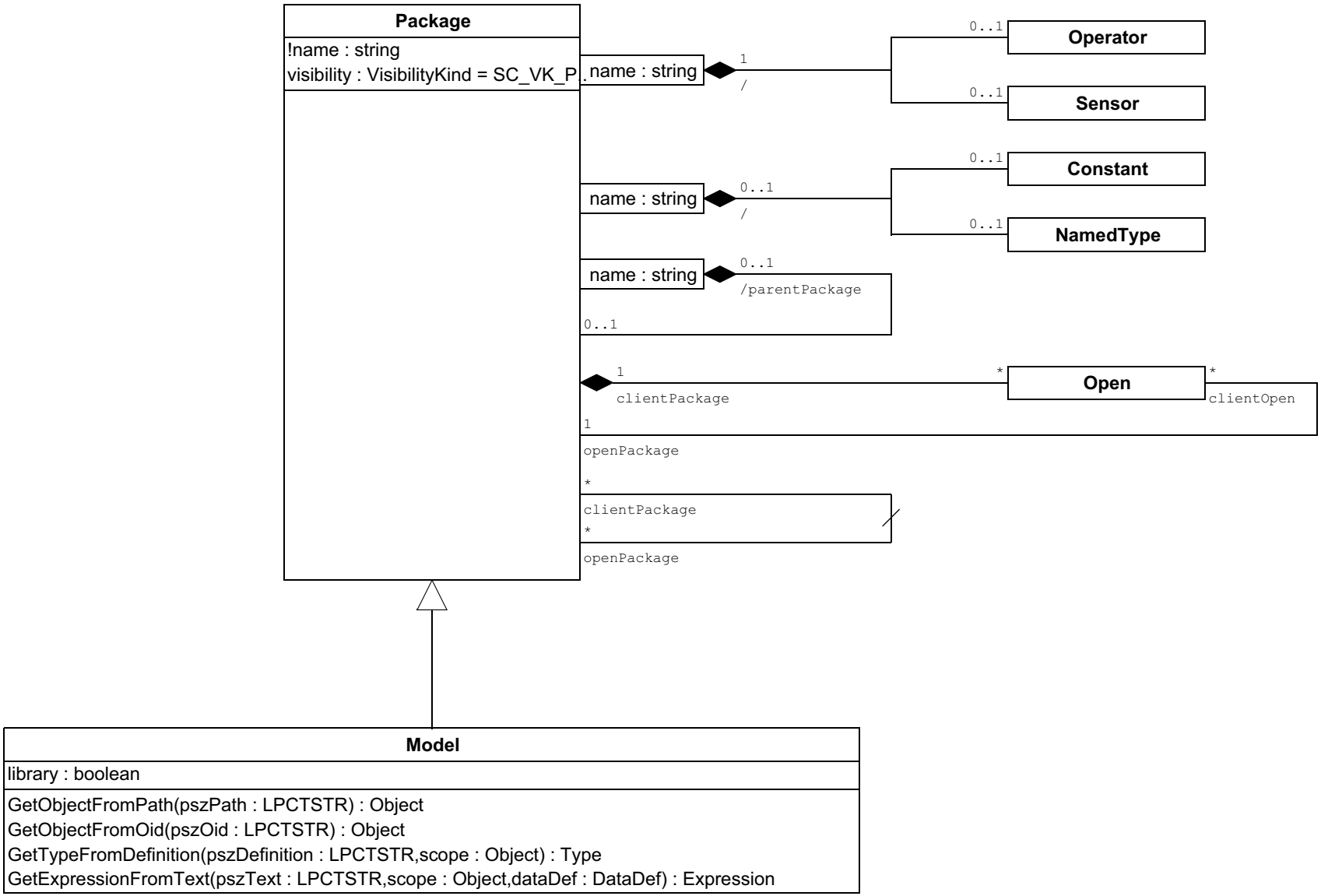
These metamodels present the data structures that give access to SCADE Suite Editor environment using Tcl API:

- "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)"

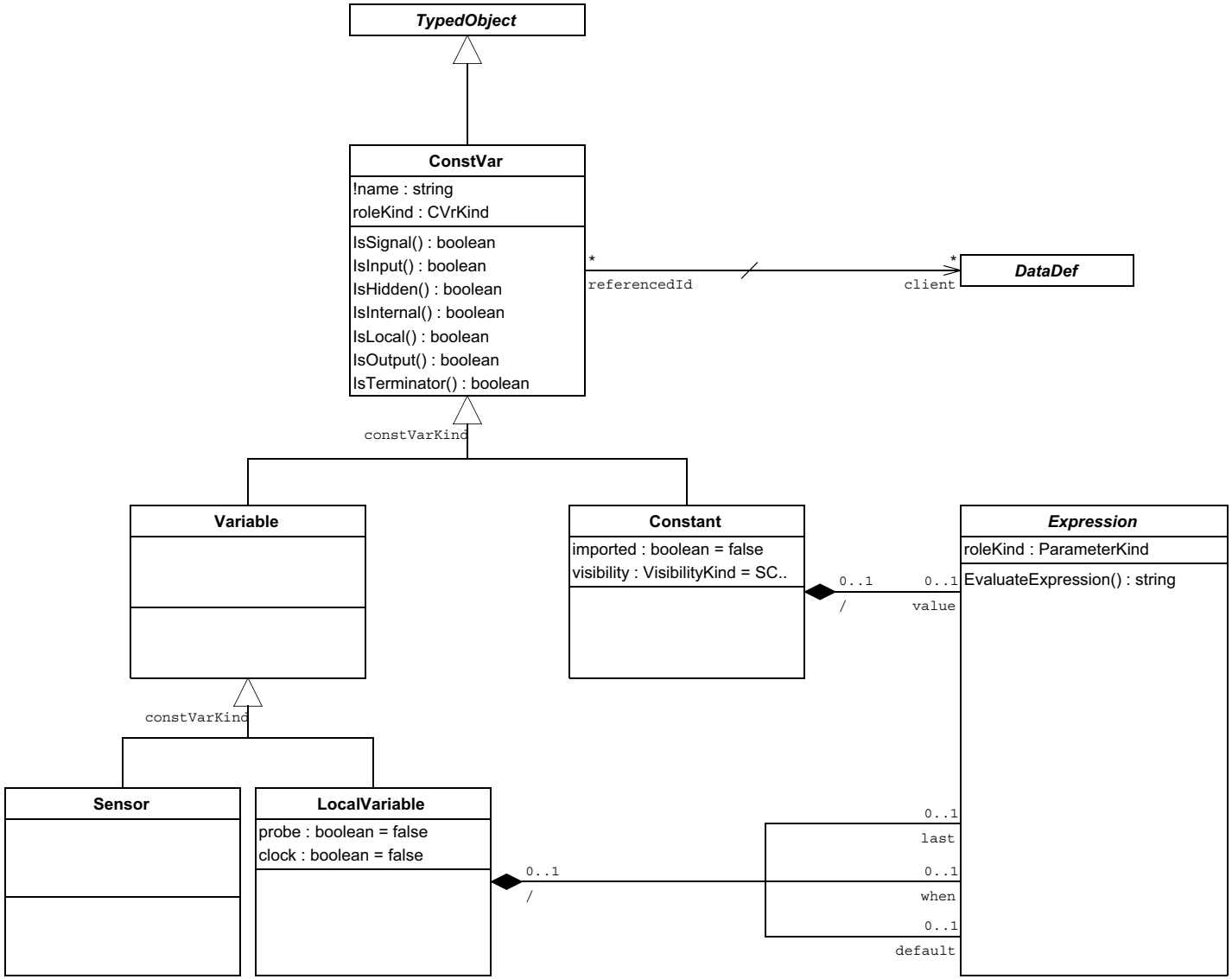
Session (Tcl)



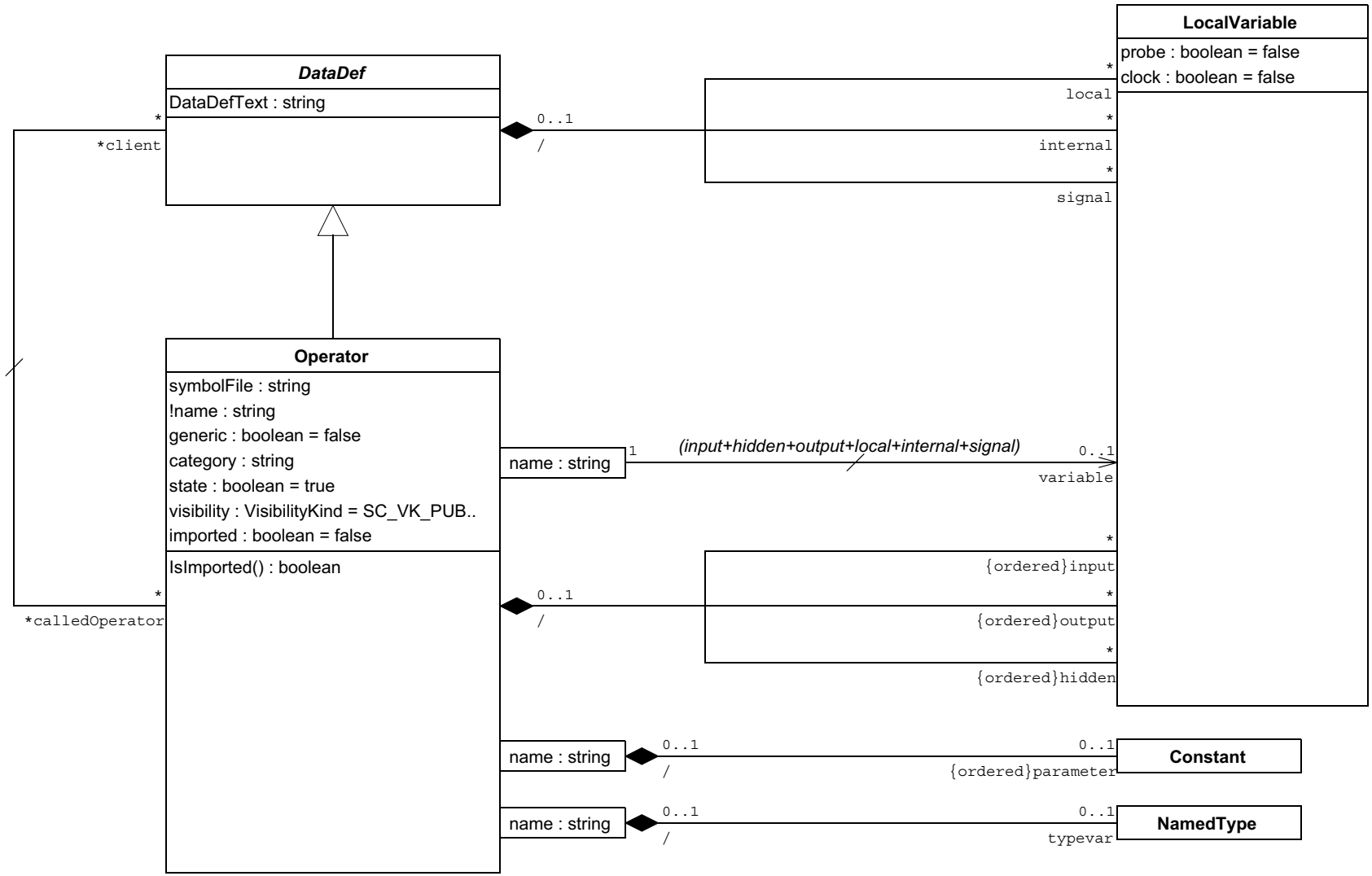
Model (Tcl)


Package
!name : string
visibility : VisibilityKind = SC_VK_P...
name : string
1
/
0..1
Operator
0..1
Sensor
name : string
0..1
/
0..1
Constant
0..1
NamedType
name : string
0..1
/parentPackage
0..1
1
clientPackage
*
Open
*
clientOpen
1
openPackage
*
clientPackage
*
openPackage
Model
library : boolean
GetObjectFromPath(pszPath : LPCTSTR) : Object
GetObjectFromOid(pszOid : LPCTSTR) : Object
GetTypeFromDefinition(pszDefinition : LPCTSTR,scope : Object) : Type
GetExpressionFromText(pszText : LPCTSTR,scope : Object,dataDef : DataDef) : Expression

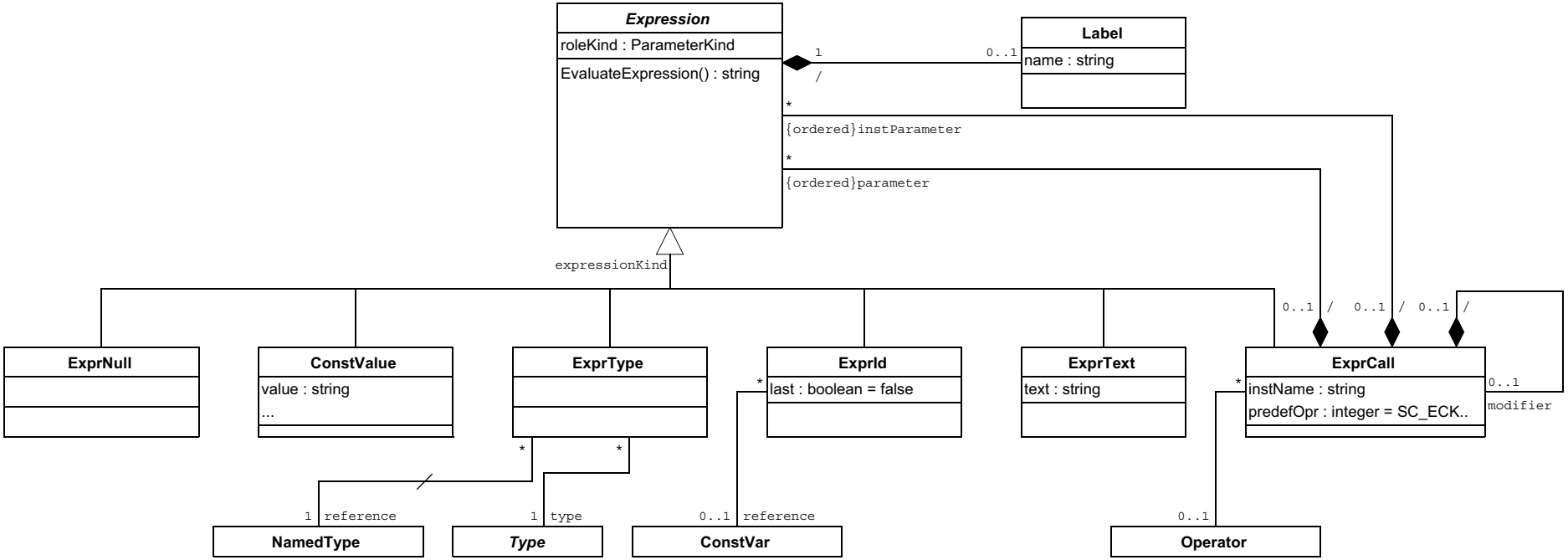
Variable (Tcl)



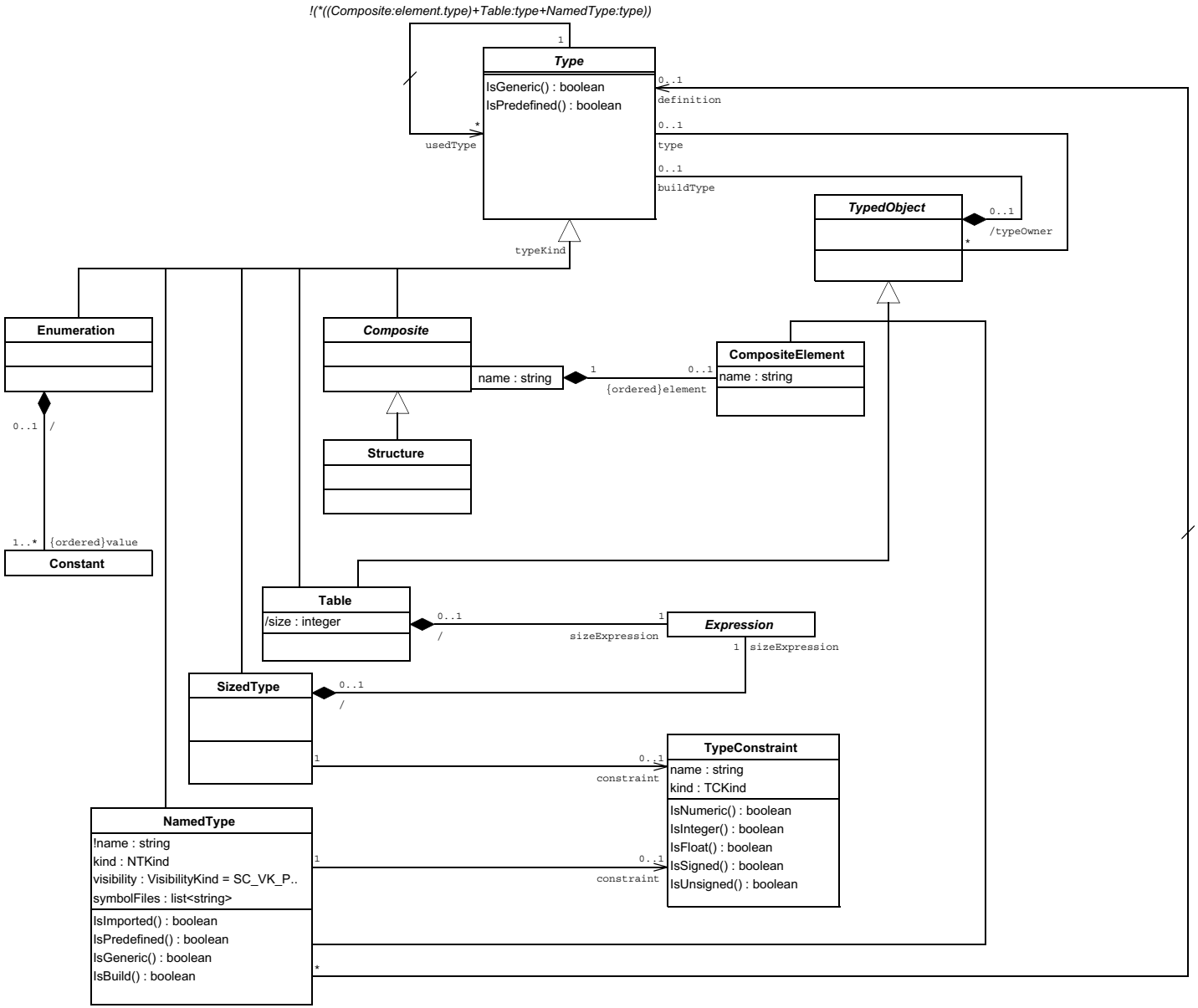
Operator (Tcl)



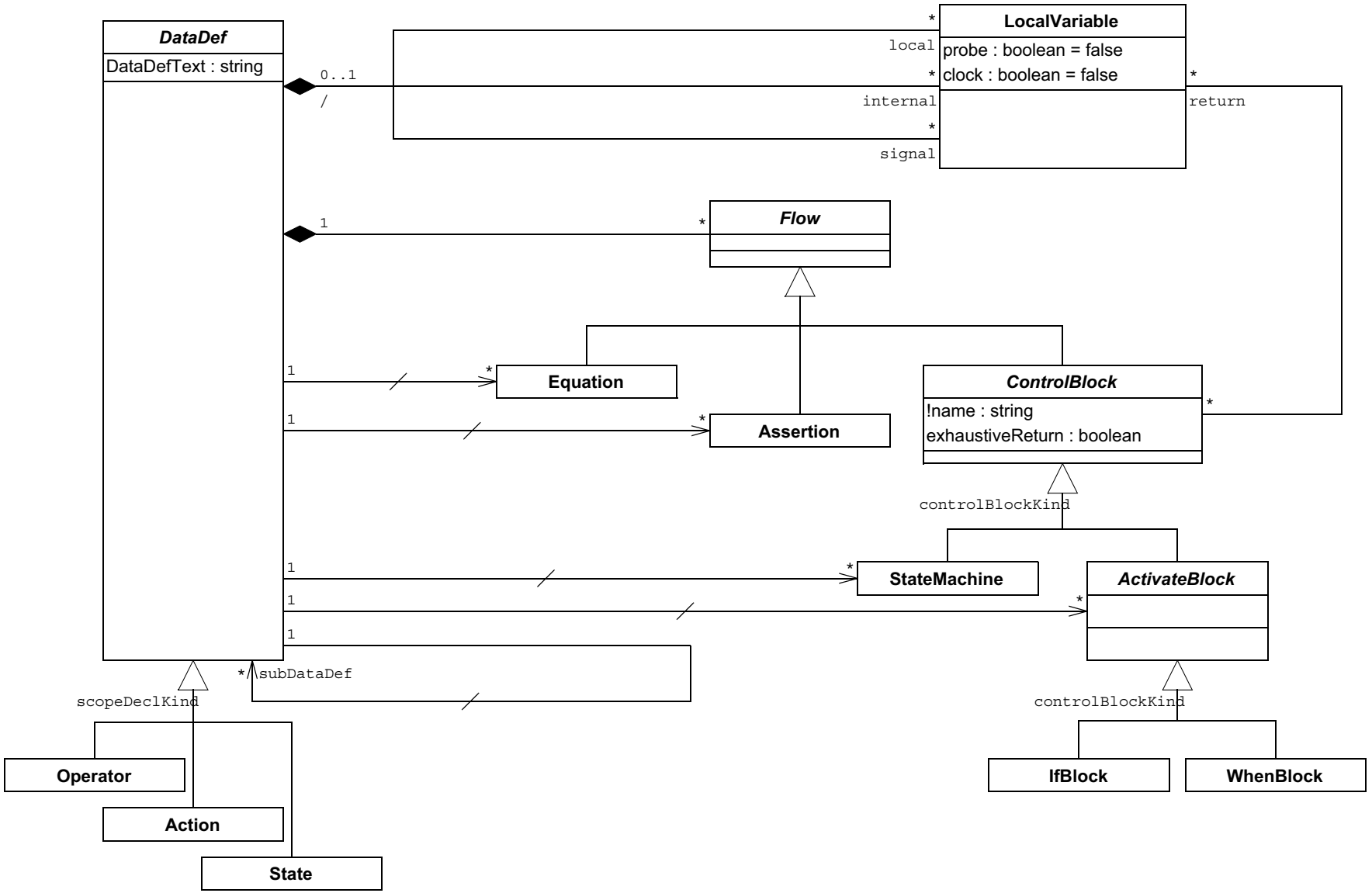
Expression (Tcl)



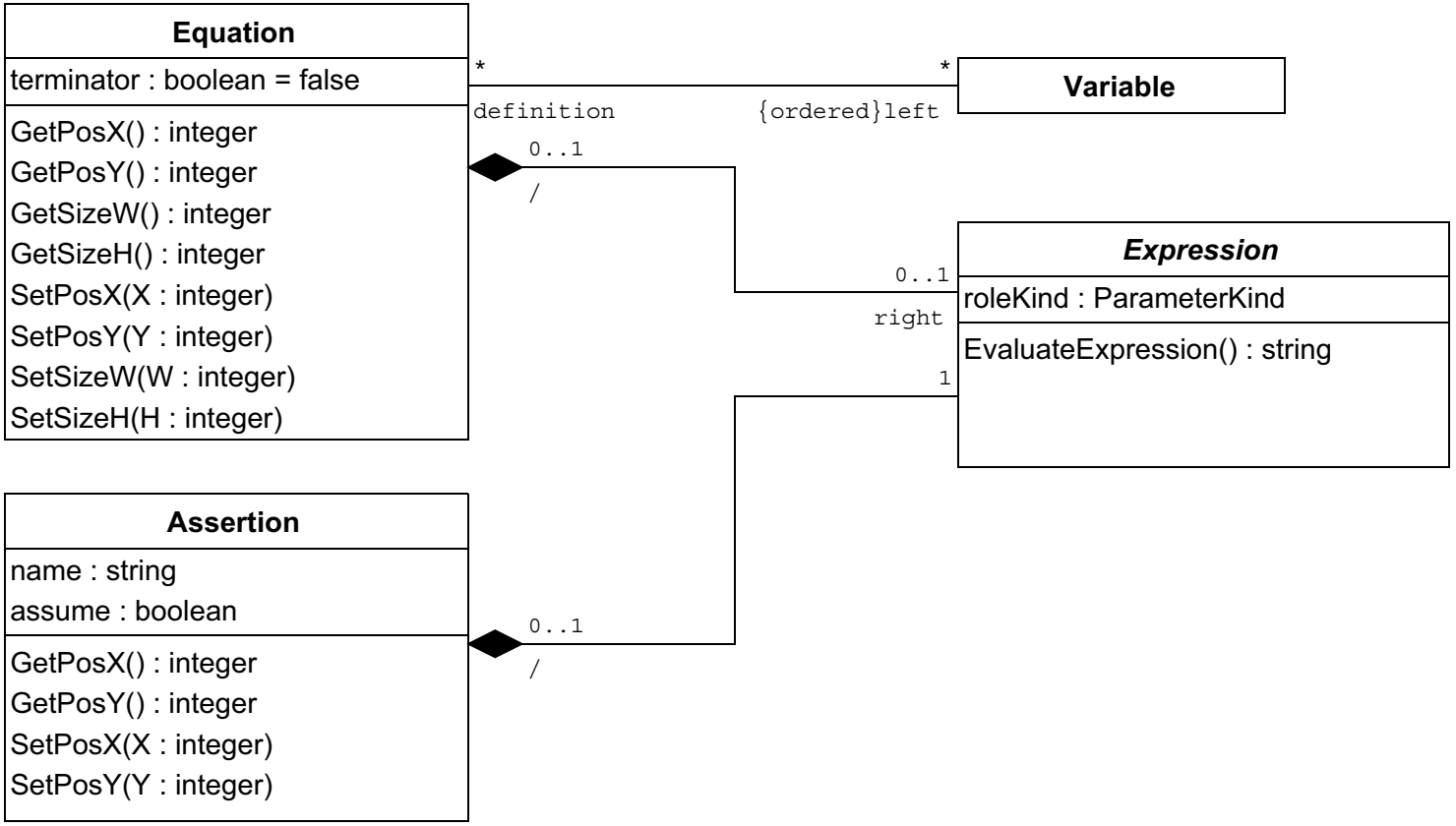
Type (Tcl)


!(*((Composite:element.type)+Table:type+NamedType:type))
Type
IsGeneric() : boolean
IsPredefined() : boolean
usedType
definition
type
buildType
typeKind
TypedObject
/typeOwner
Enumeration
Composite
name : string
CompositeElement
name : string
{ordered}element
Structure
Constant
{ordered}value
Table
/size : integer
Expression
sizeExpression
SizedType
TypeConstraint
name : string
kind : TCKind
IsNumeric() : boolean
IsInteger() : boolean
IsFloat() : boolean
IsSigned() : boolean
IsUnsigned() : boolean
constraint
NamedType
!name : string
kind : NTKind
visibility : VisibilityKind = SC_VK_P..
symbolFiles : list<string>
IsImported() : boolean
IsPredefined() : boolean
IsGeneric() : boolean
IsBuild() : boolean

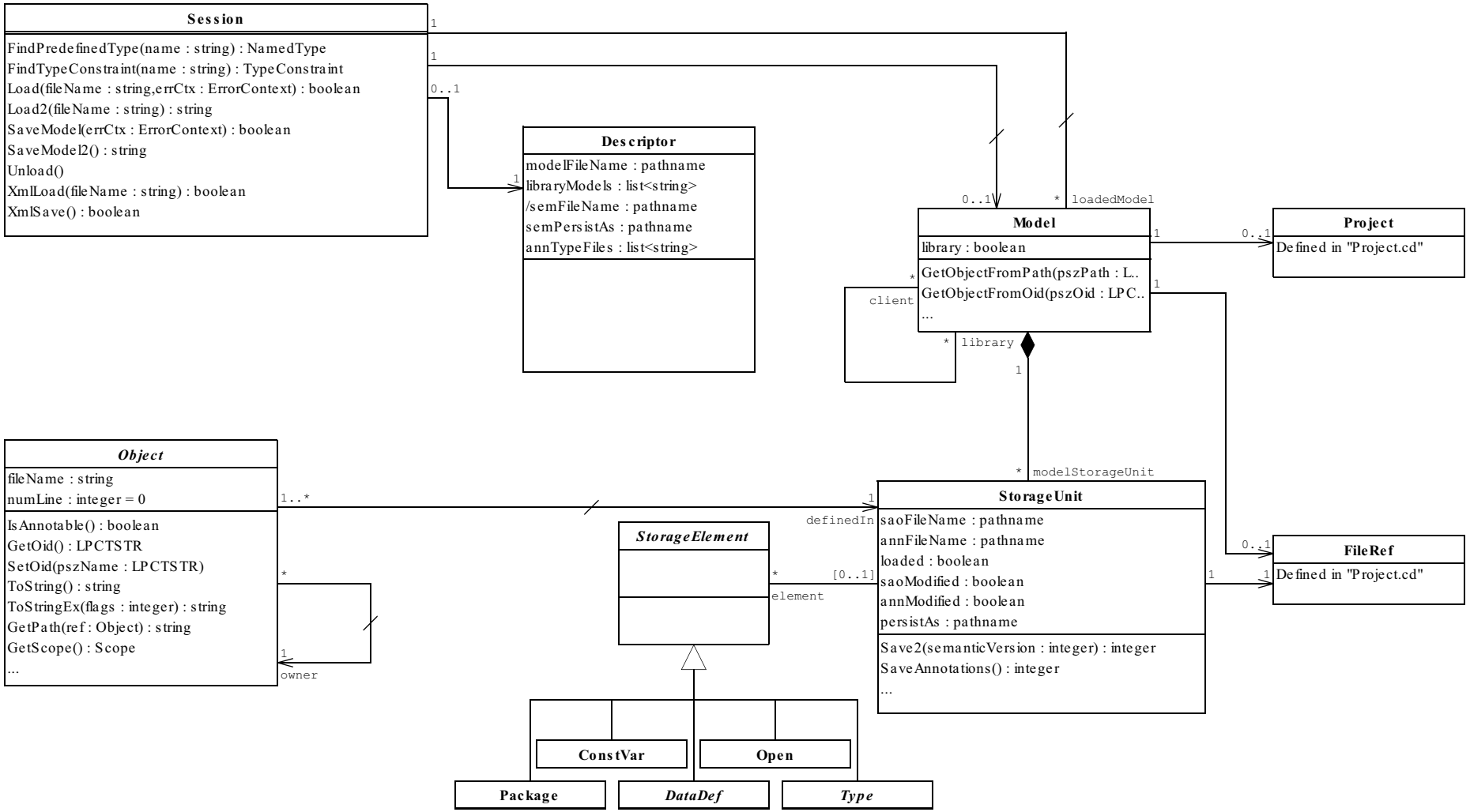
Data Definition (Tcl)



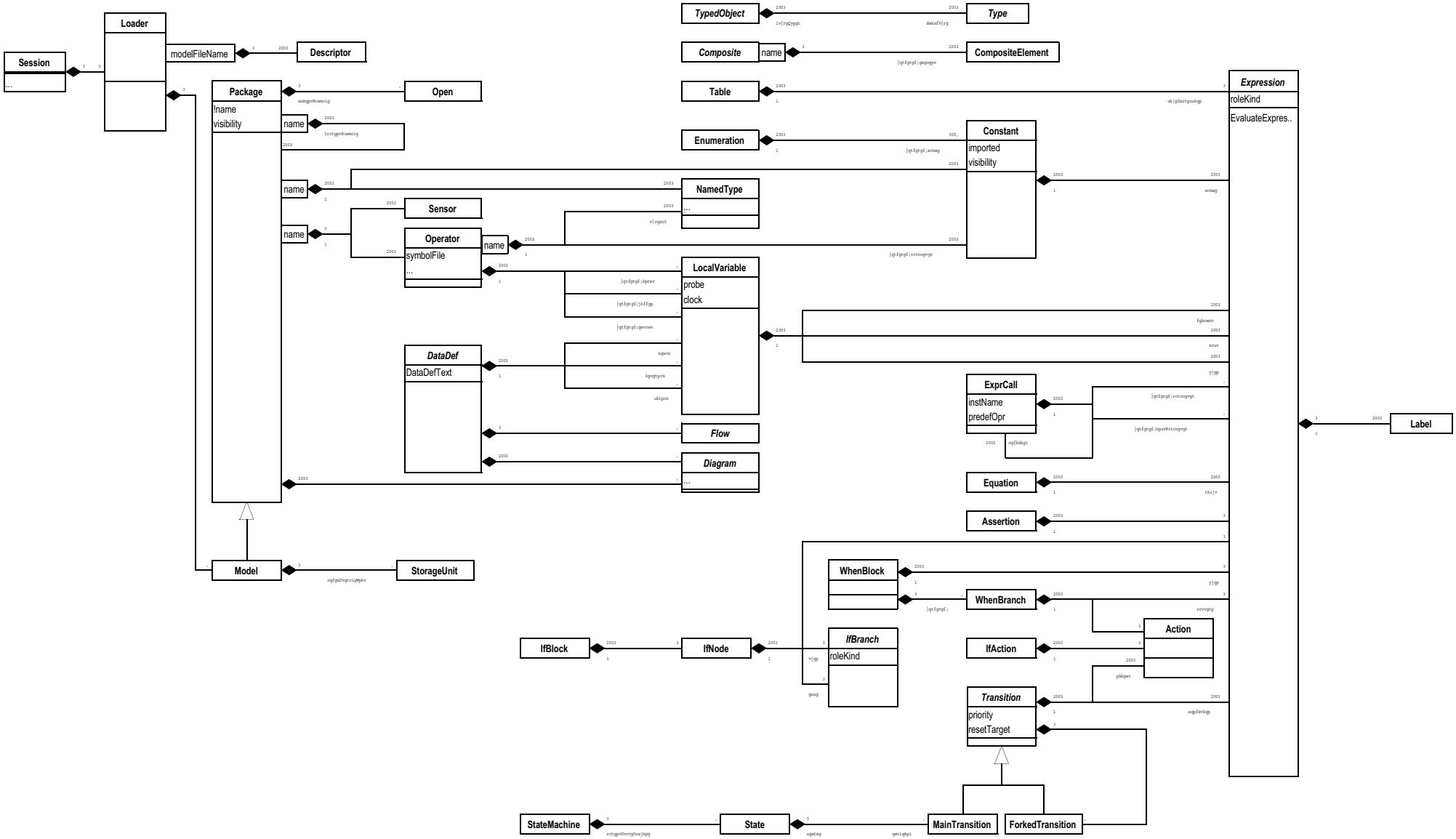
Data Flow (Tcl)



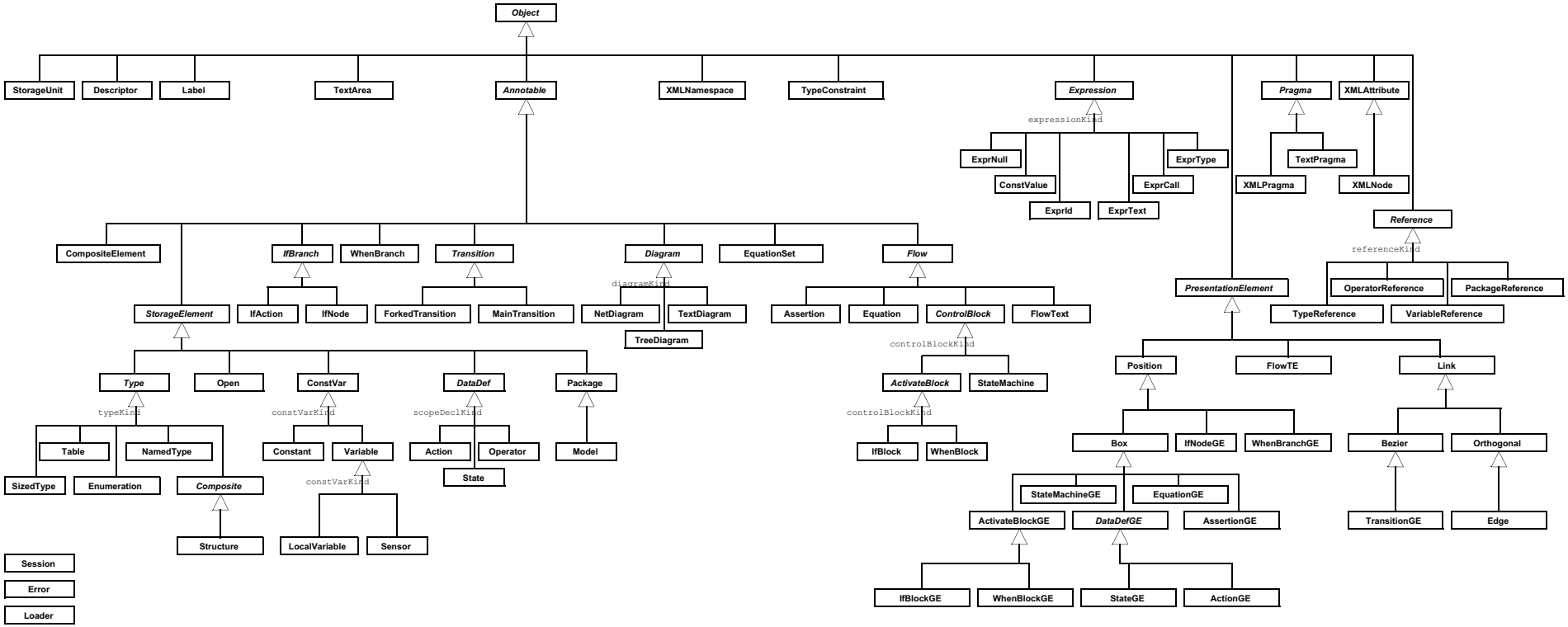
Model Storage - IO (Tcl)



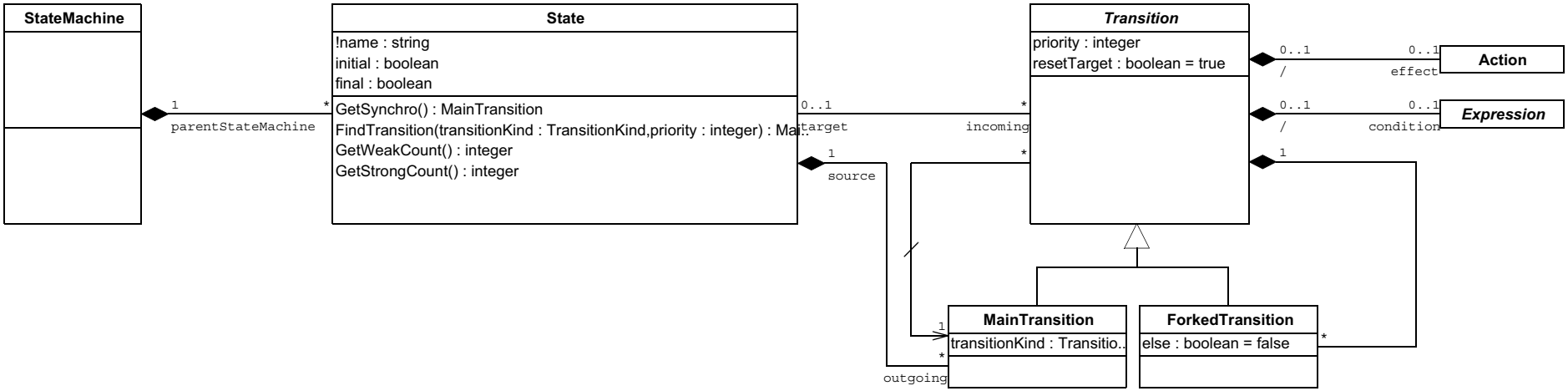
Composition (Tcl)



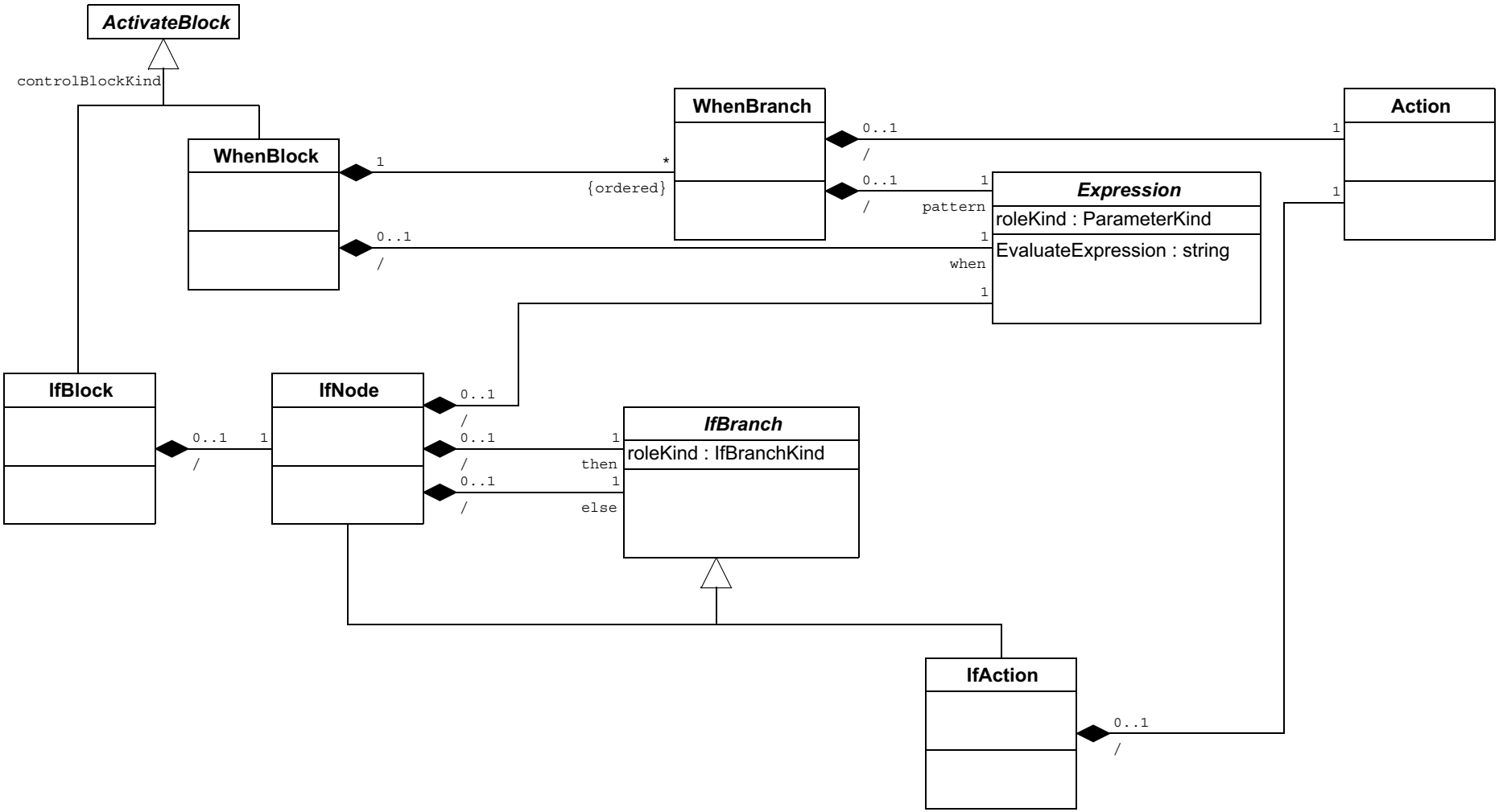
Inheritance (Tcl)



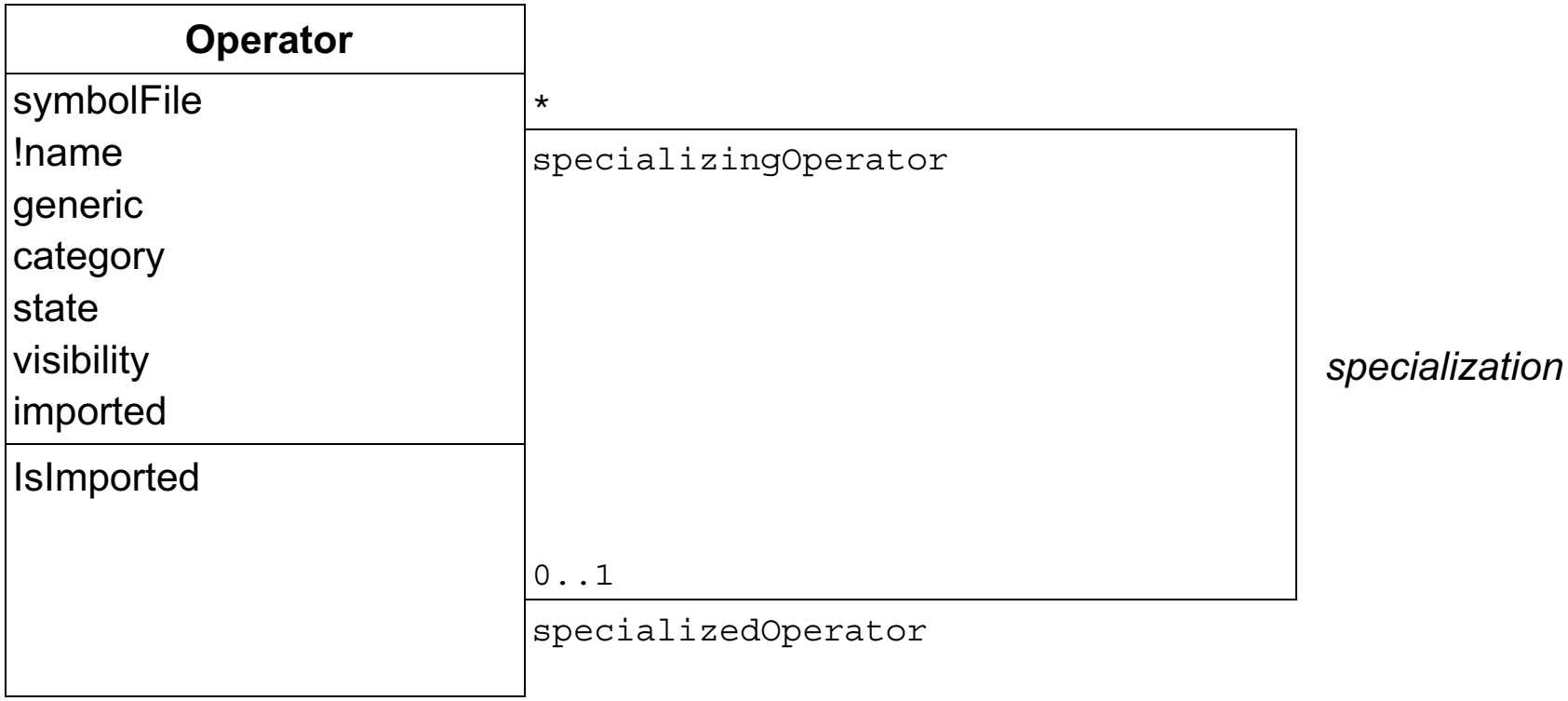
State Machine (Tcl)



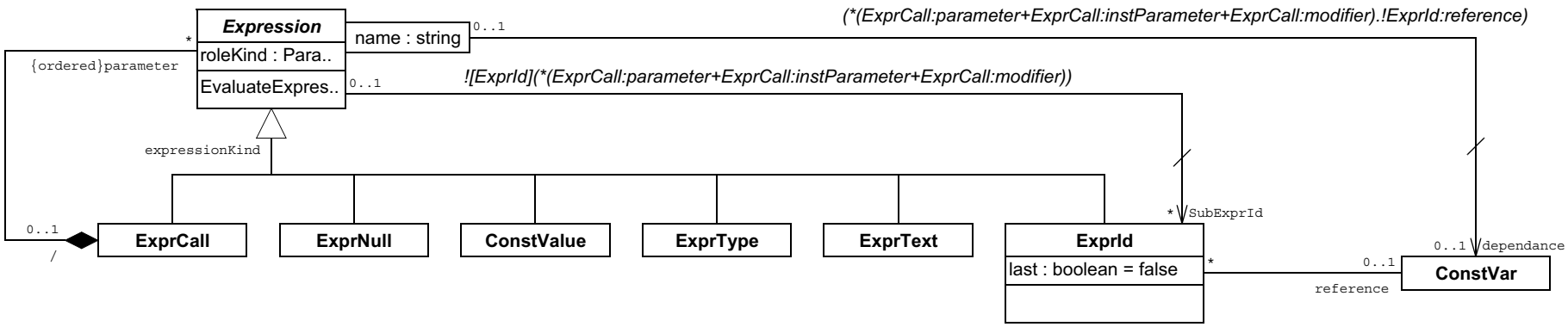
ActivateBlock (Tcl)



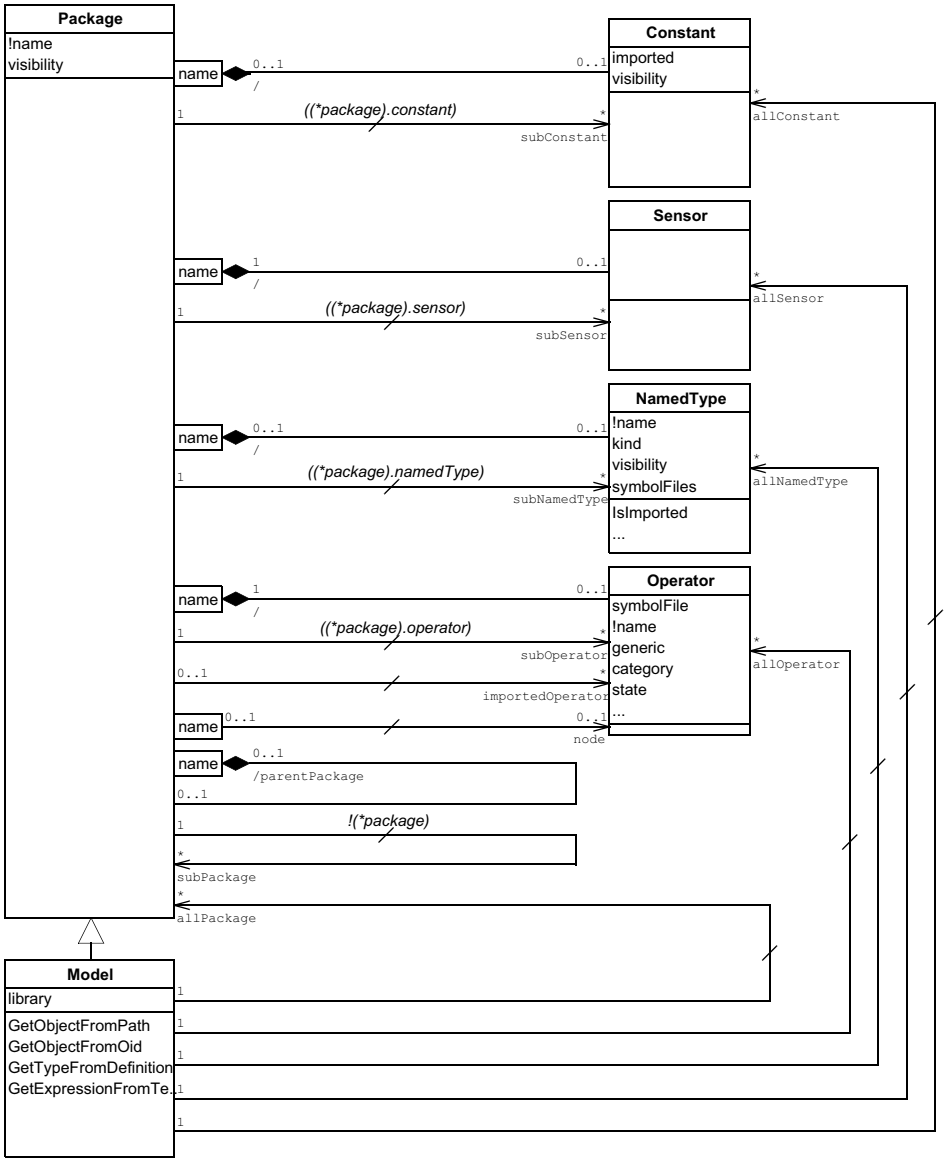
Specialization (Tcl)



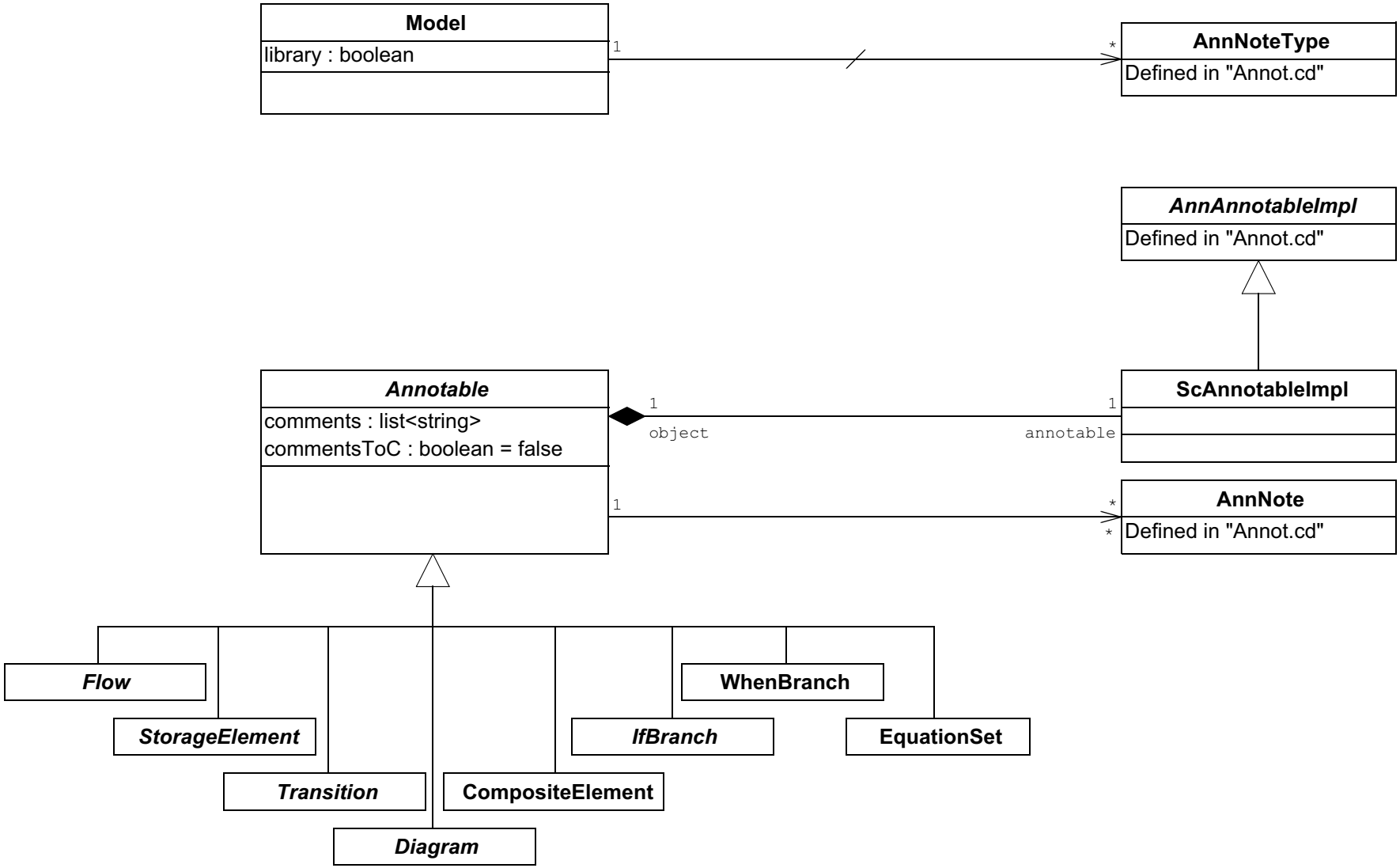
Expression Maps (Tcl)



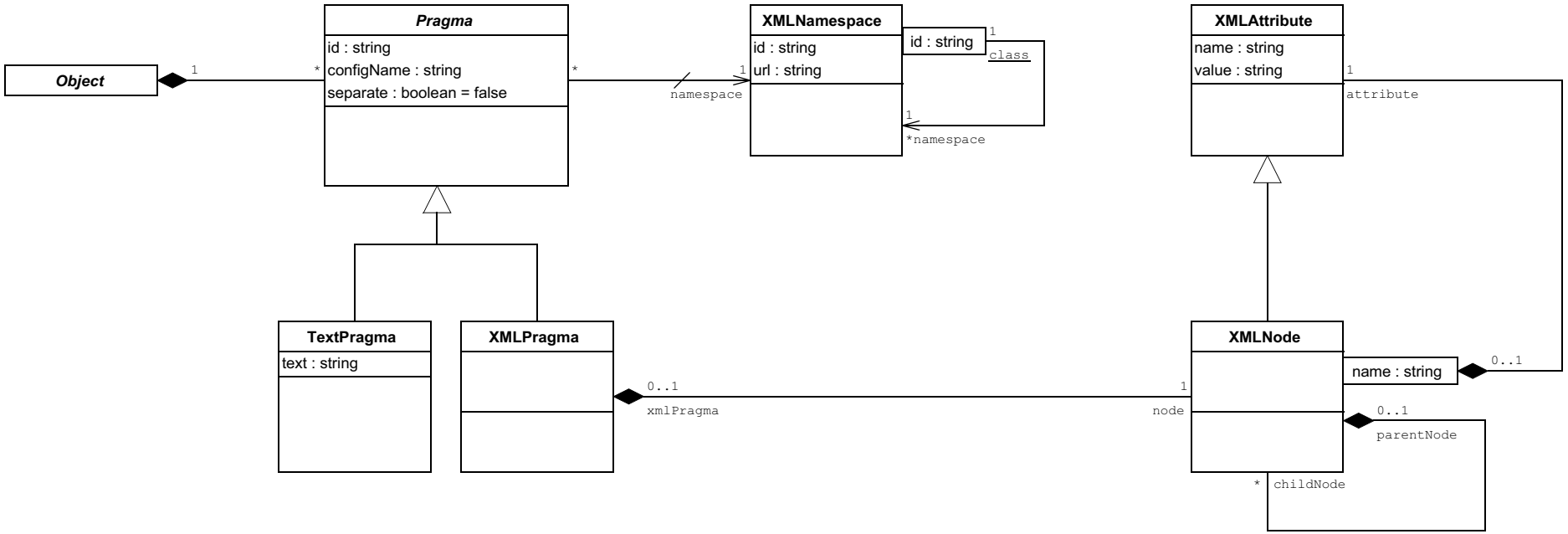
Model and Package Maps (Tcl)



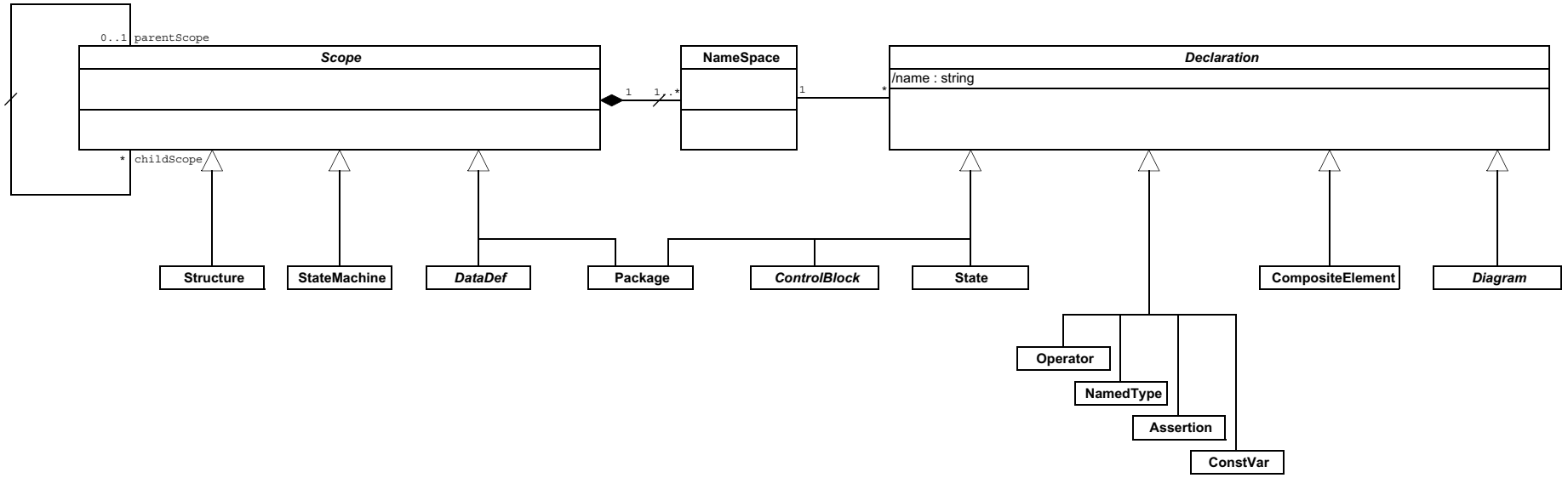
Annotable (Tcl)



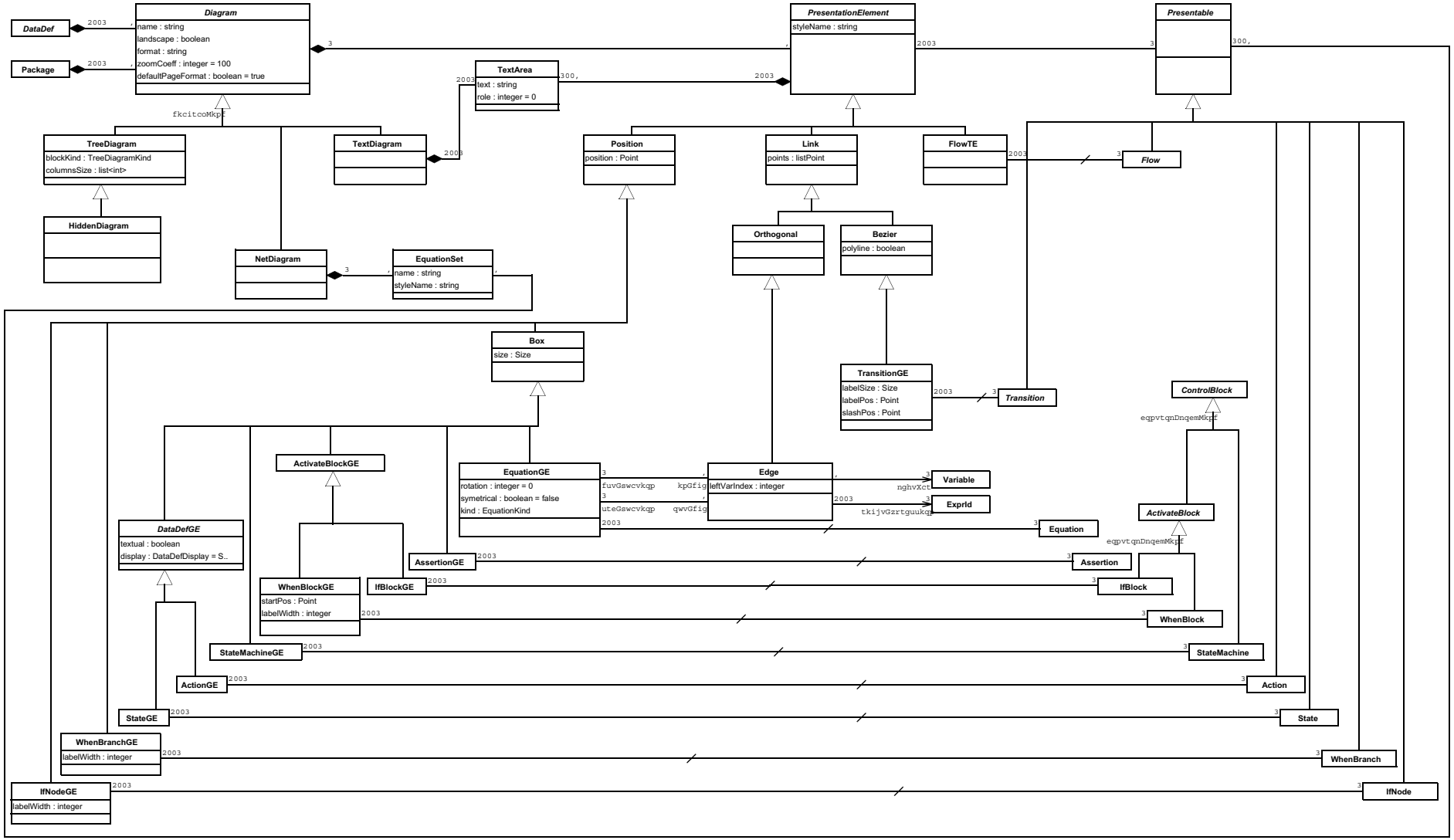
Pragmas (Tcl)


Pragma
id : string
configName : string
separate : boolean = false
Object
1
*
*
1
namespace
XMLNamespace
id : string
url : string
id : string
1
class
1
*namespace
XMLAttribute
name : string
value : string
1
attribute
TextPragma
text : string
XMLPragma
0..1
xmlPragma
1
node
XMLNode
name : string
0..1
0..1
parentNode
*
childNode

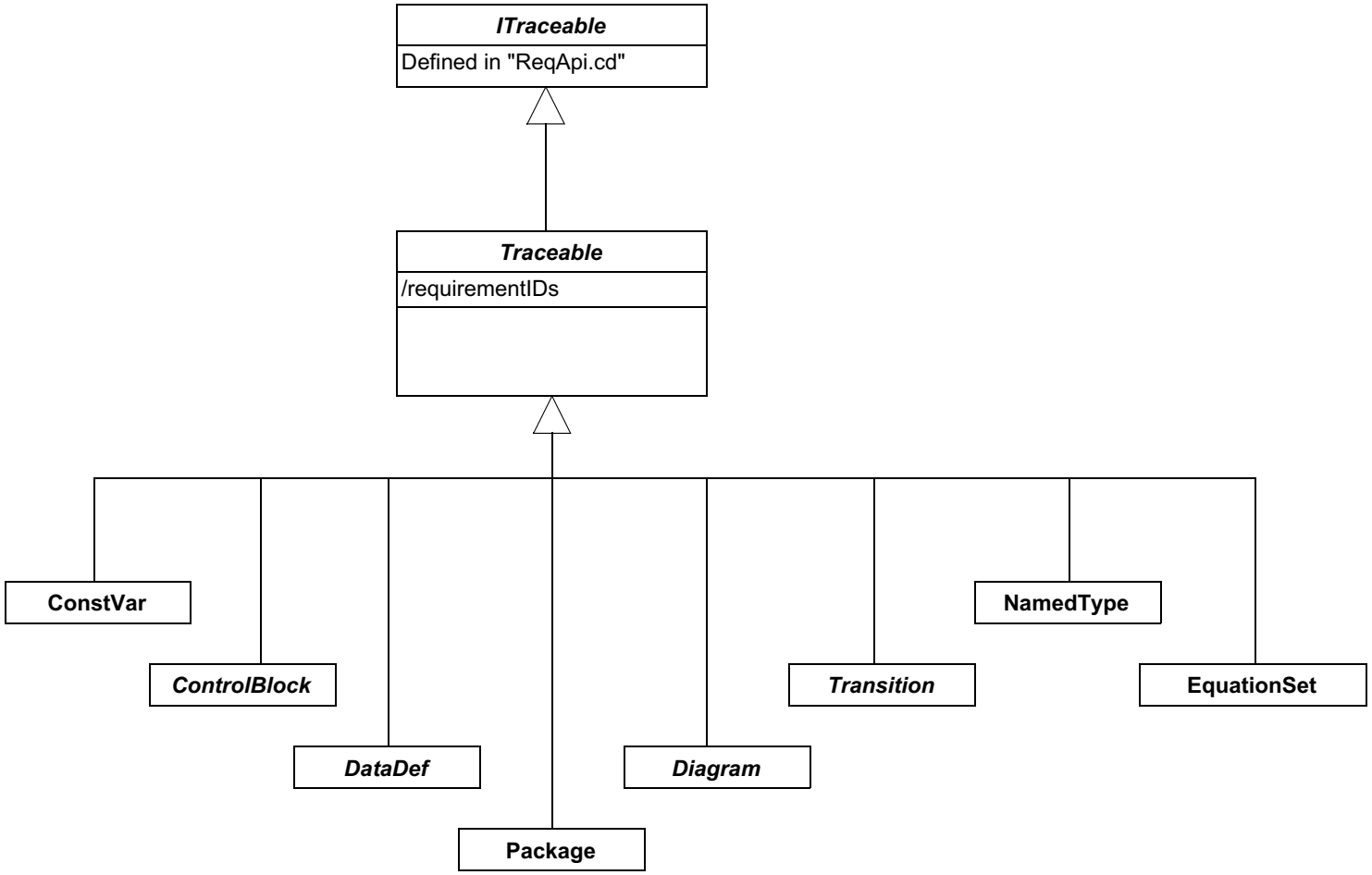
Scopes (Tcl)


0..1 parentScope
Scope
NameSpace
Declaration
/name : string
1
1..*
1
*
* childScope
Structure
StateMachine
DataDef
Package
ControlBlock
State
CompositeElement
Diagram
Operator
NamedType
Assertion
ConstVar

Graphical (Tcl)



Traceability (Tcl)

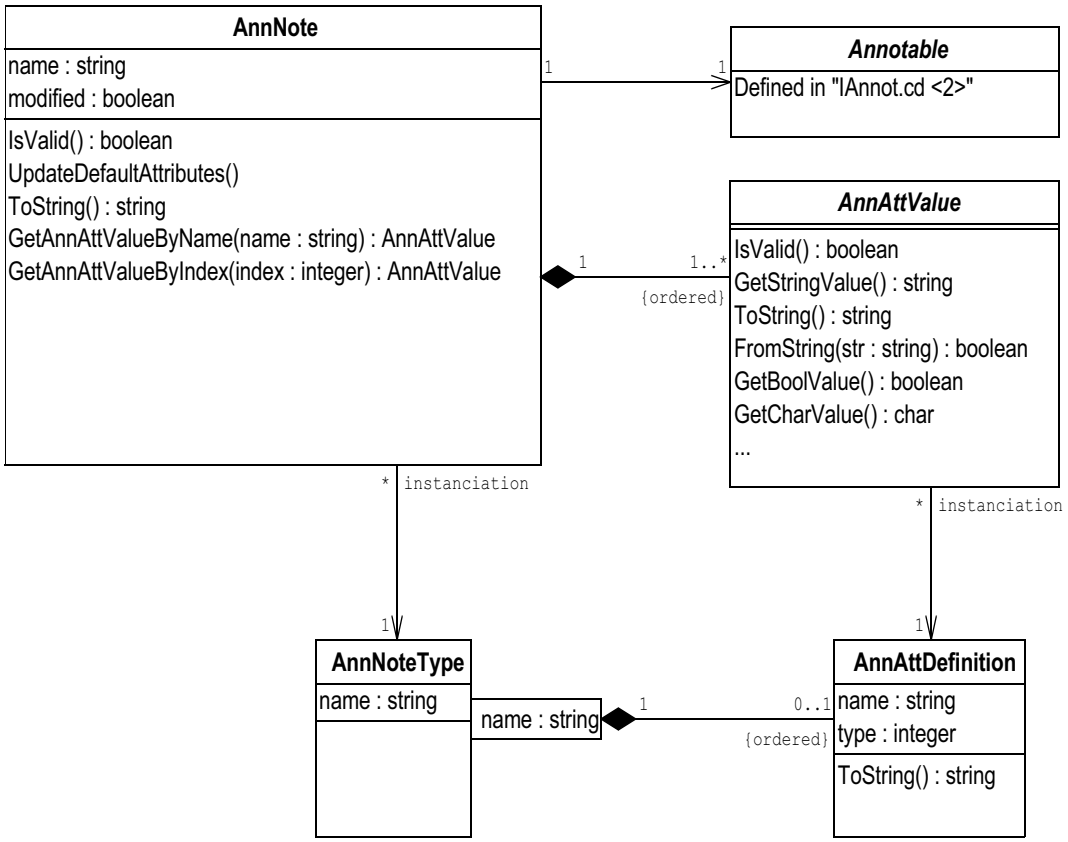


16 / Annotation Metamodels (Tcl API)

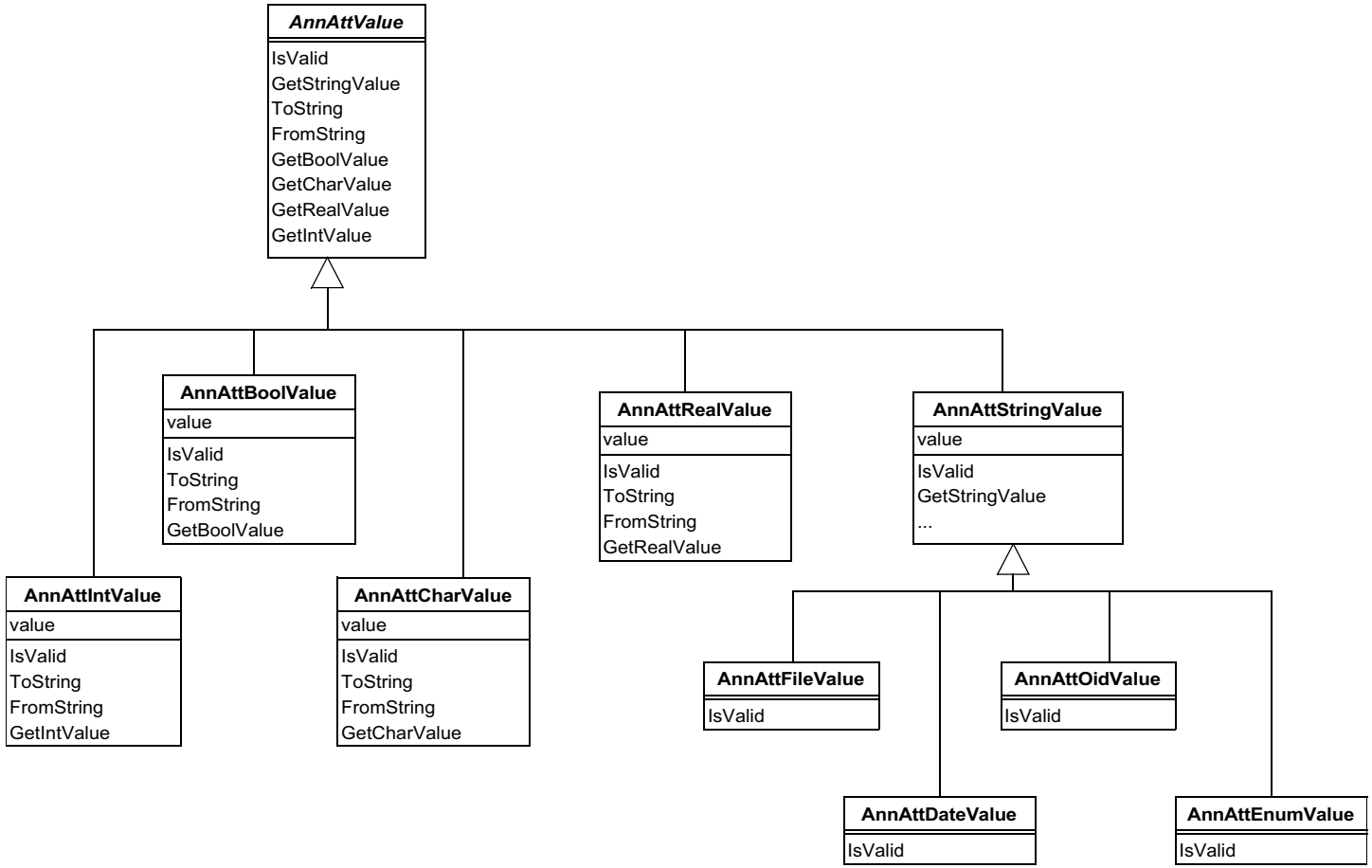
These metamodels present the data structures that give access to SCADE Suite annotations mechanisms using Tcl API:

- “Notes (Tcl)”
- “Notes Attribute Values (Tcl)”
- “Notes Property Values (Tcl)”

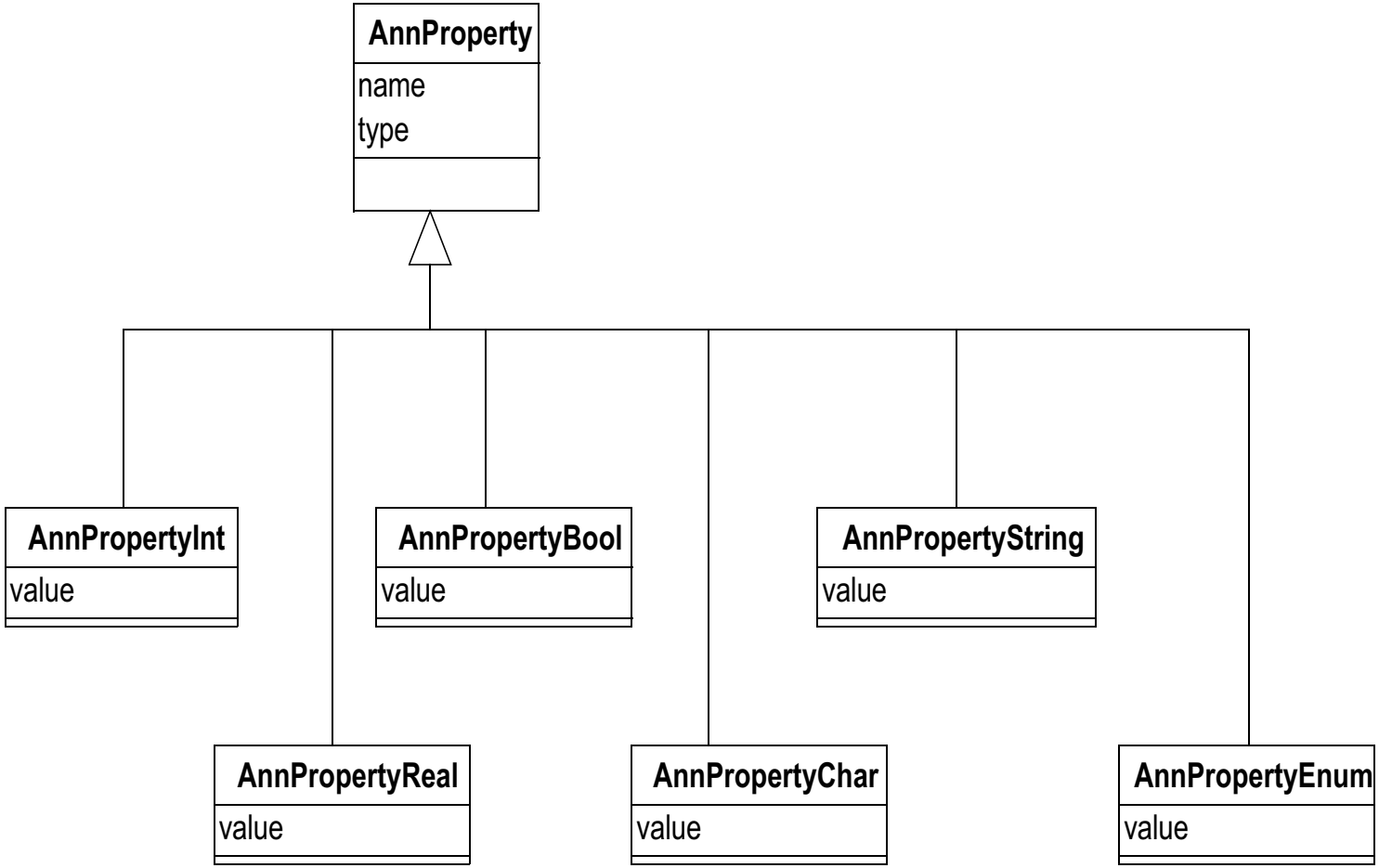
Notes (Tcl)


AnnNote
name : string
modified : boolean
IsValid() : boolean
UpdateDefaultAttributes()
ToString() : string
GetAnnAttValueByName(name : string) : AnnAttValue
GetAnnAttValueByIndex(index : integer) : AnnAttValue
1
1
Annotable
Defined in "IAnnot.cd <2>"
1
1..*
{ordered}
AnnAttValue
IsValid() : boolean
GetStringValue() : string
ToString() : string
FromString(str : string) : boolean
GetBoolValue() : boolean
GetCharValue() : char
...
* instanciation
1
* instanciation
1
AnnNoteType
name : string
name : string
1
0..1
{ordered}
AnnAttDefinition
name : string
type : integer
ToString() : string

Notes Attribute Values (Tcl)

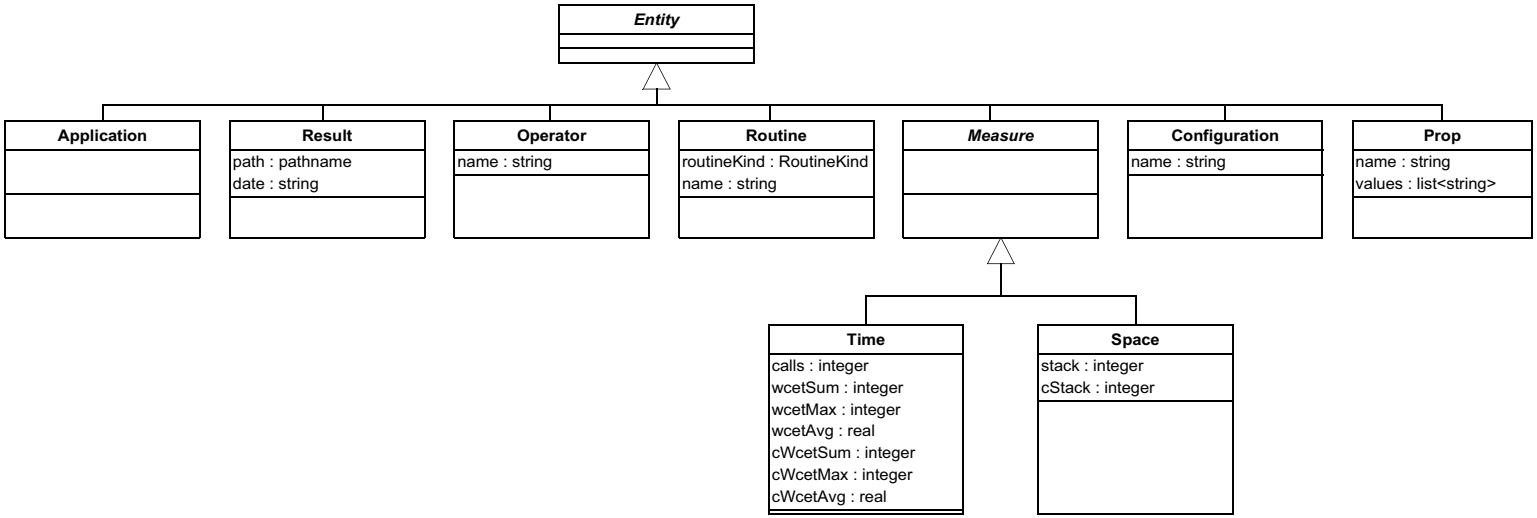


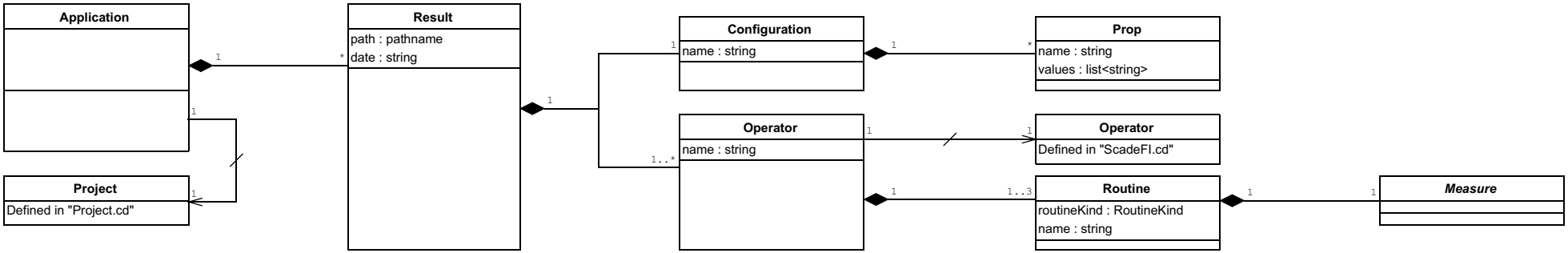
Notes Property Values (Tcl)



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:





Reference Card

Part 4

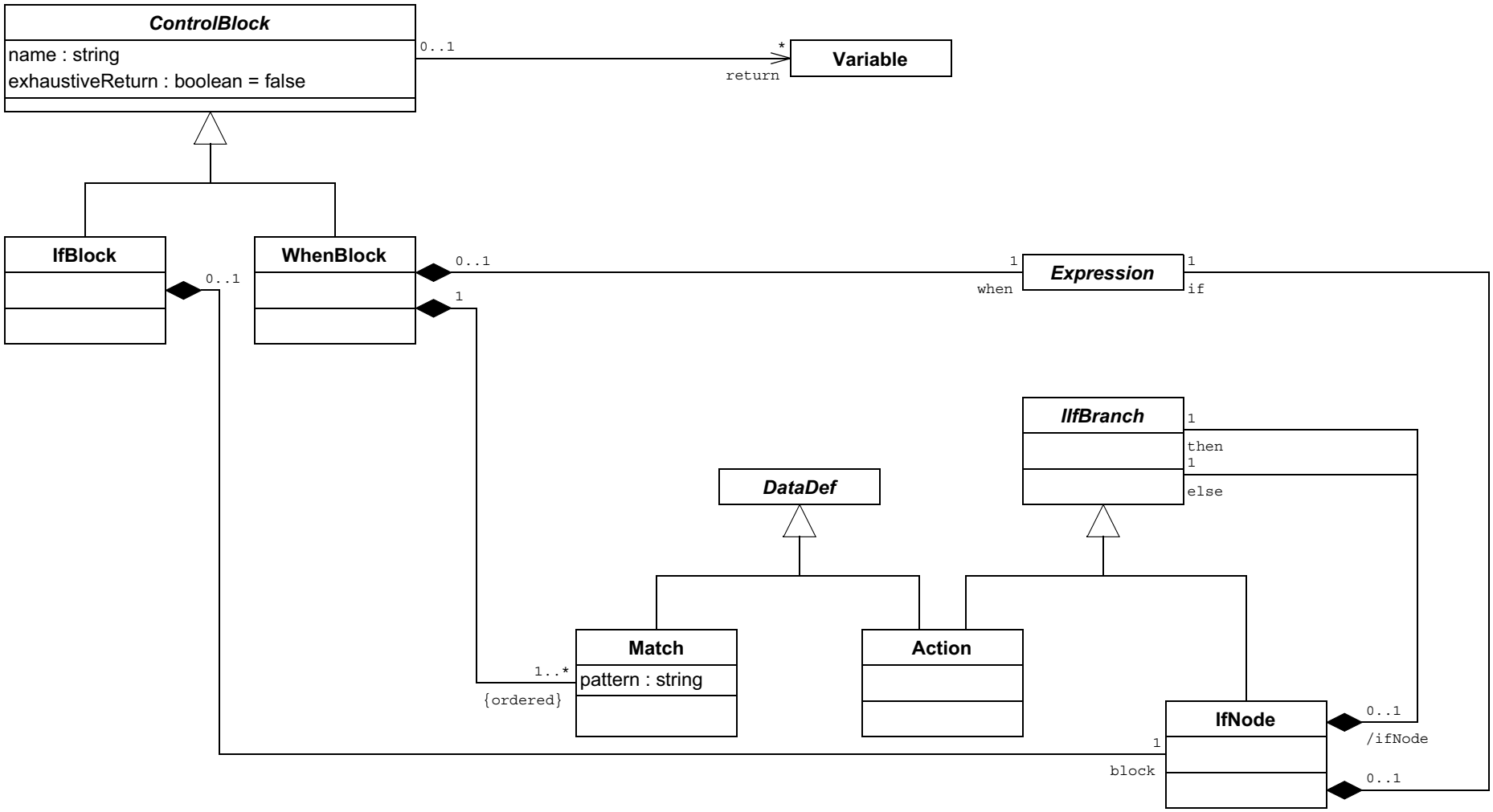
SCADE Suite Metamodels for Java API

18 / Scade Language Metamodels

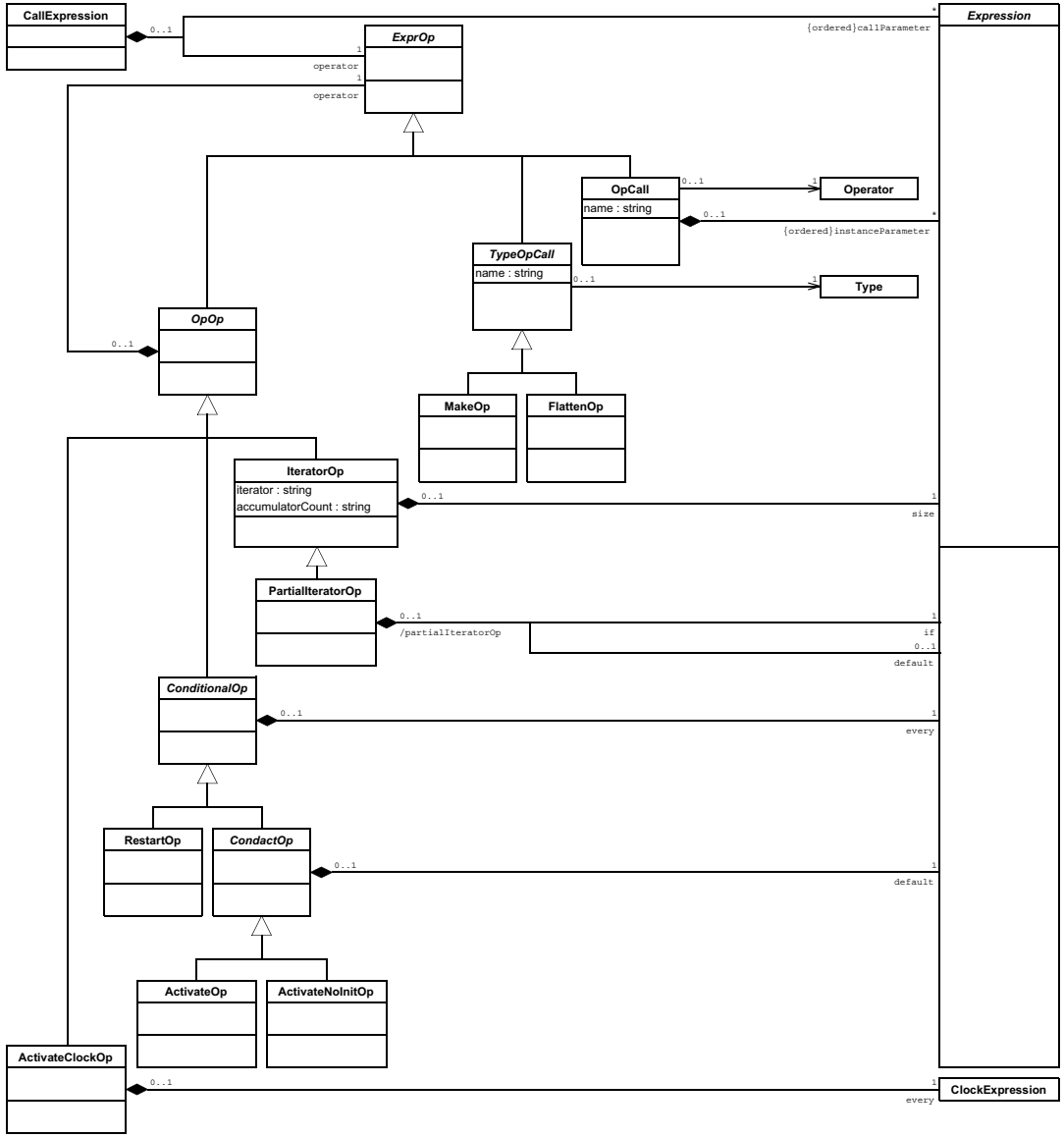
These metamodels present the data structures that give access to Scade language constructs using Java API:

- "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"

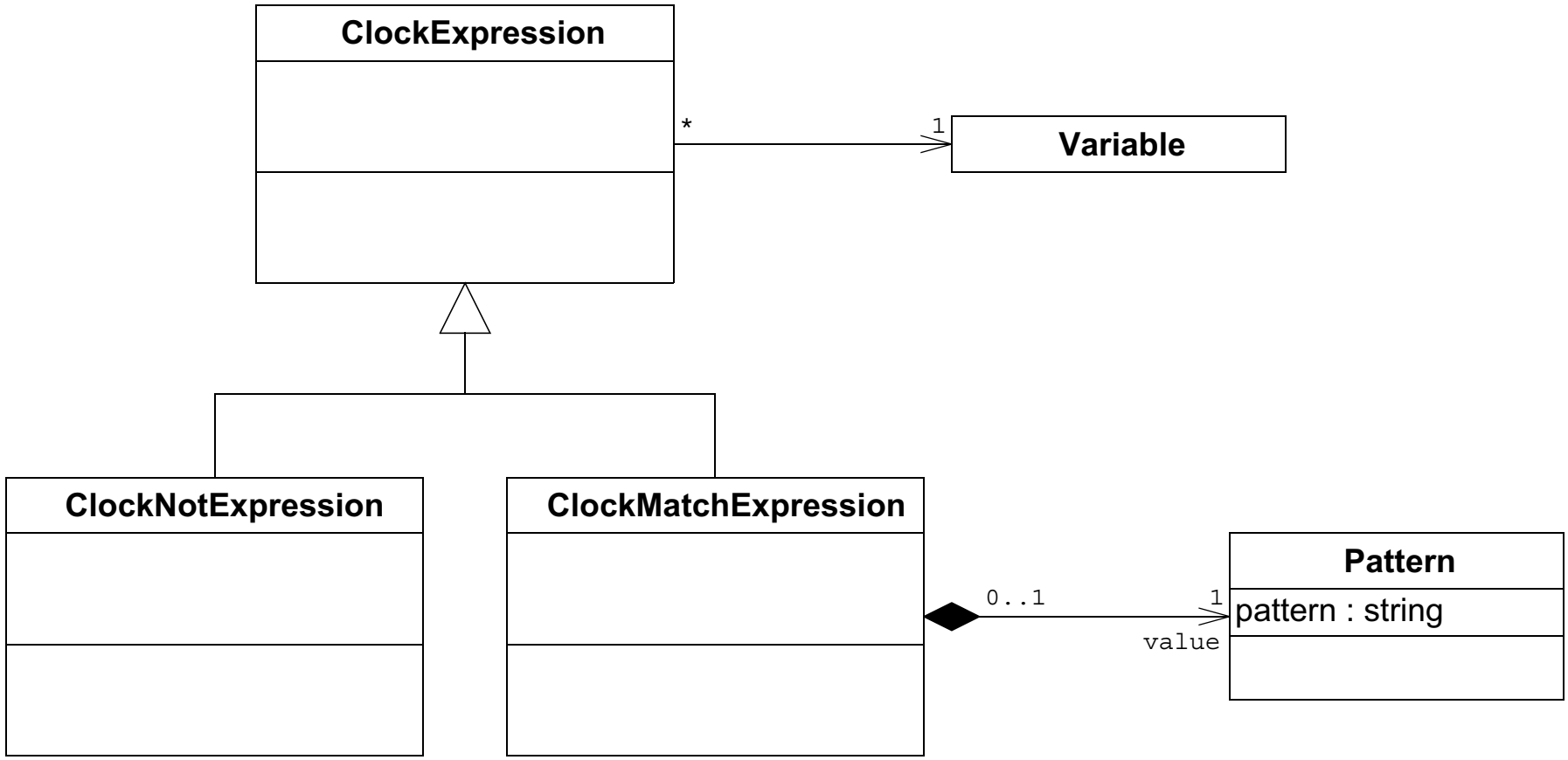
Activate Block



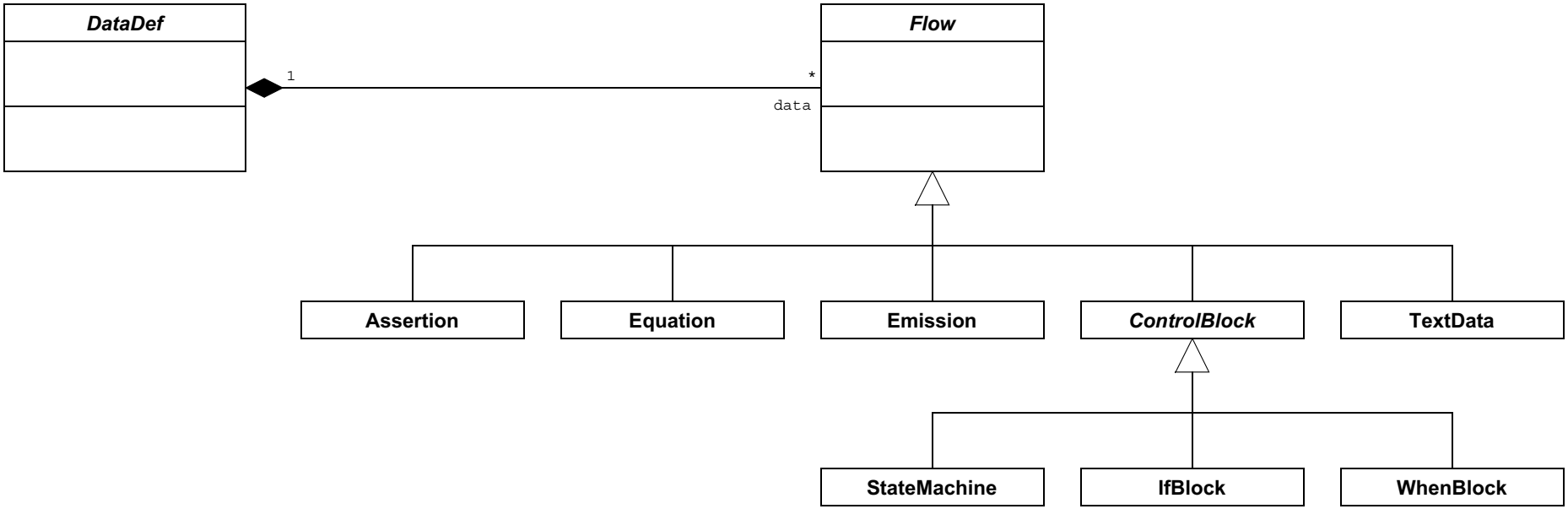
Call Expression



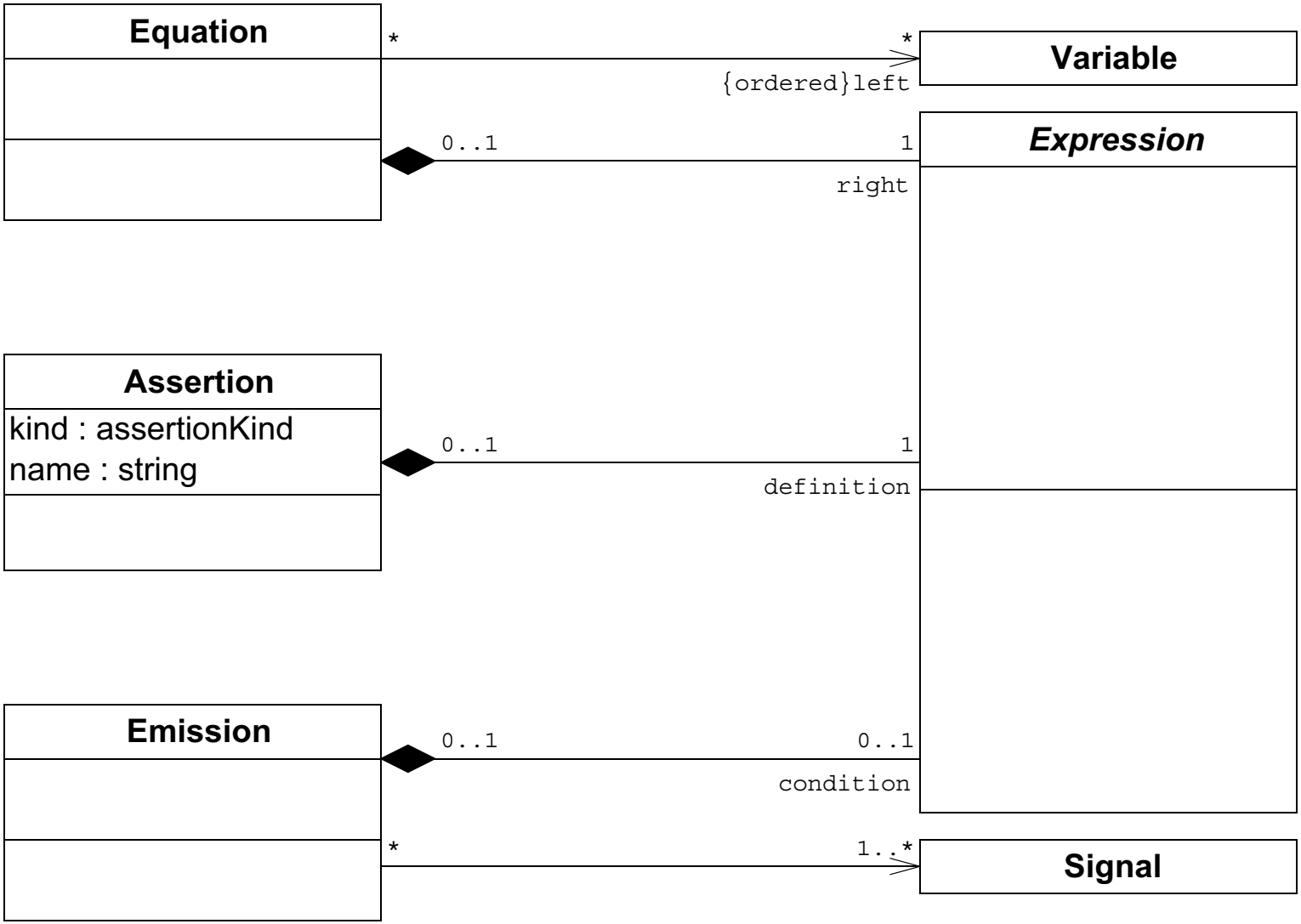
Clocks


ClockExpression
*
1
Variable
ClockNotExpression
ClockMatchExpression
0..1
1
value
Pattern
pattern : string

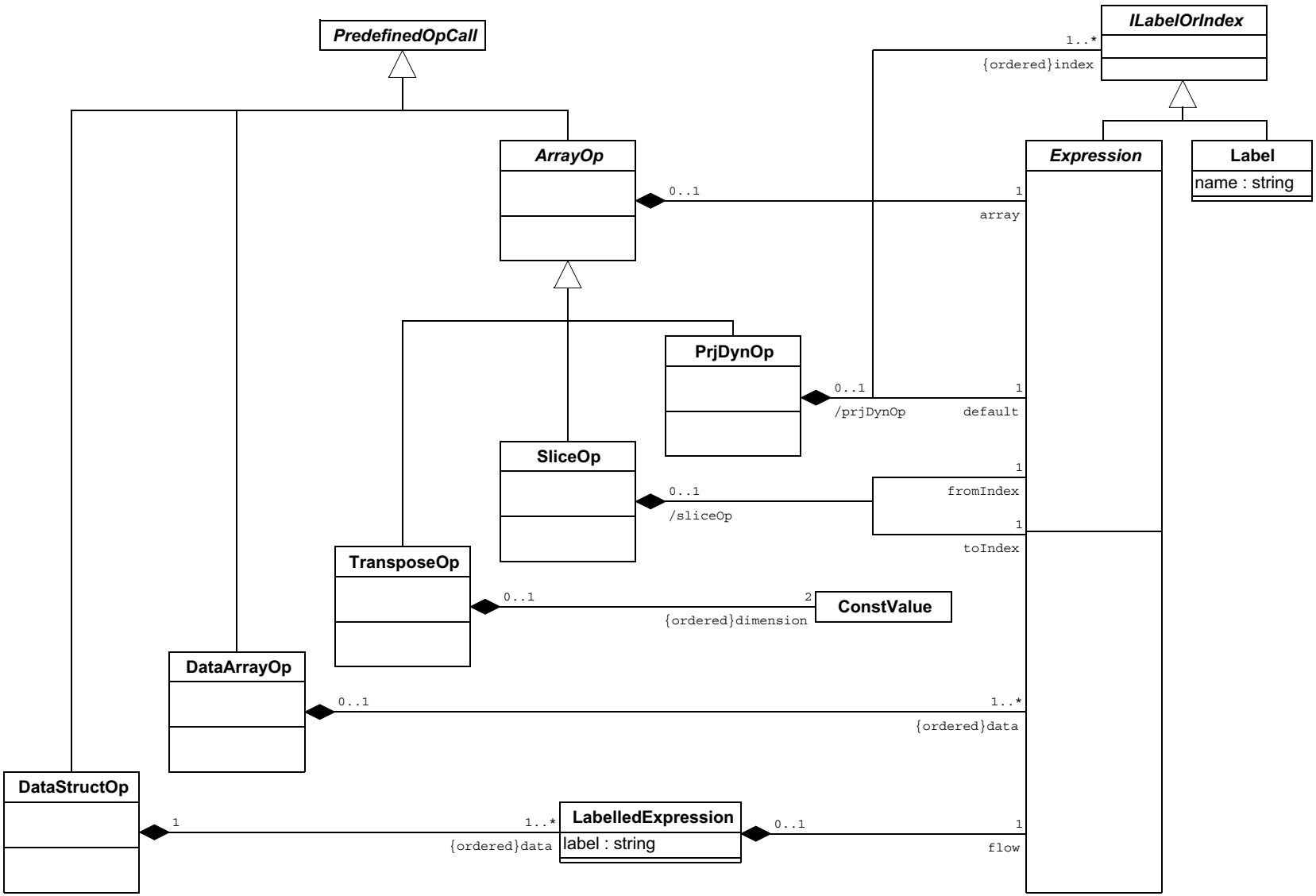
Data Definition



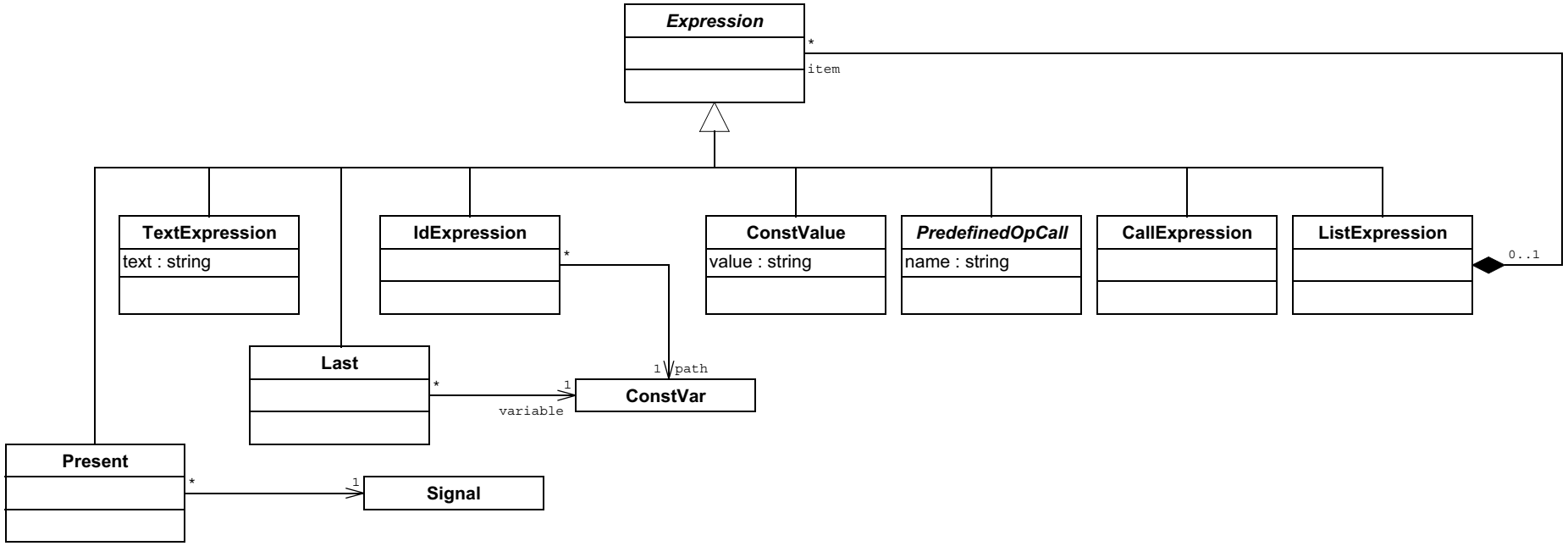
Data Flow


Equation
*
*
Variable
{ordered}left
0..1
1
Expression
right
Assertion
kind : assertionKind
name : string
0..1
1
definition
Emission
0..1
0..1
condition
*
1..*
Signal

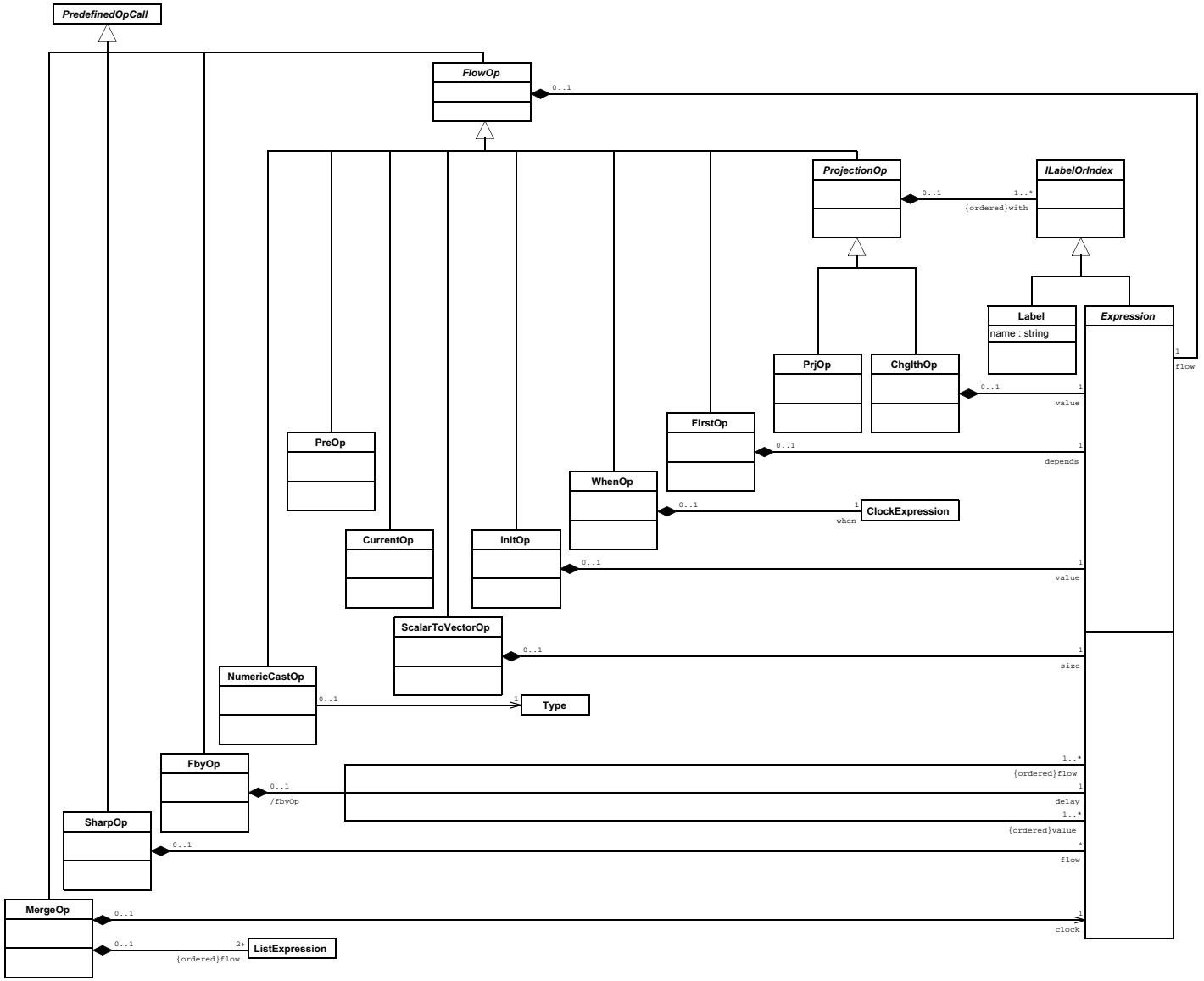
Data Operators



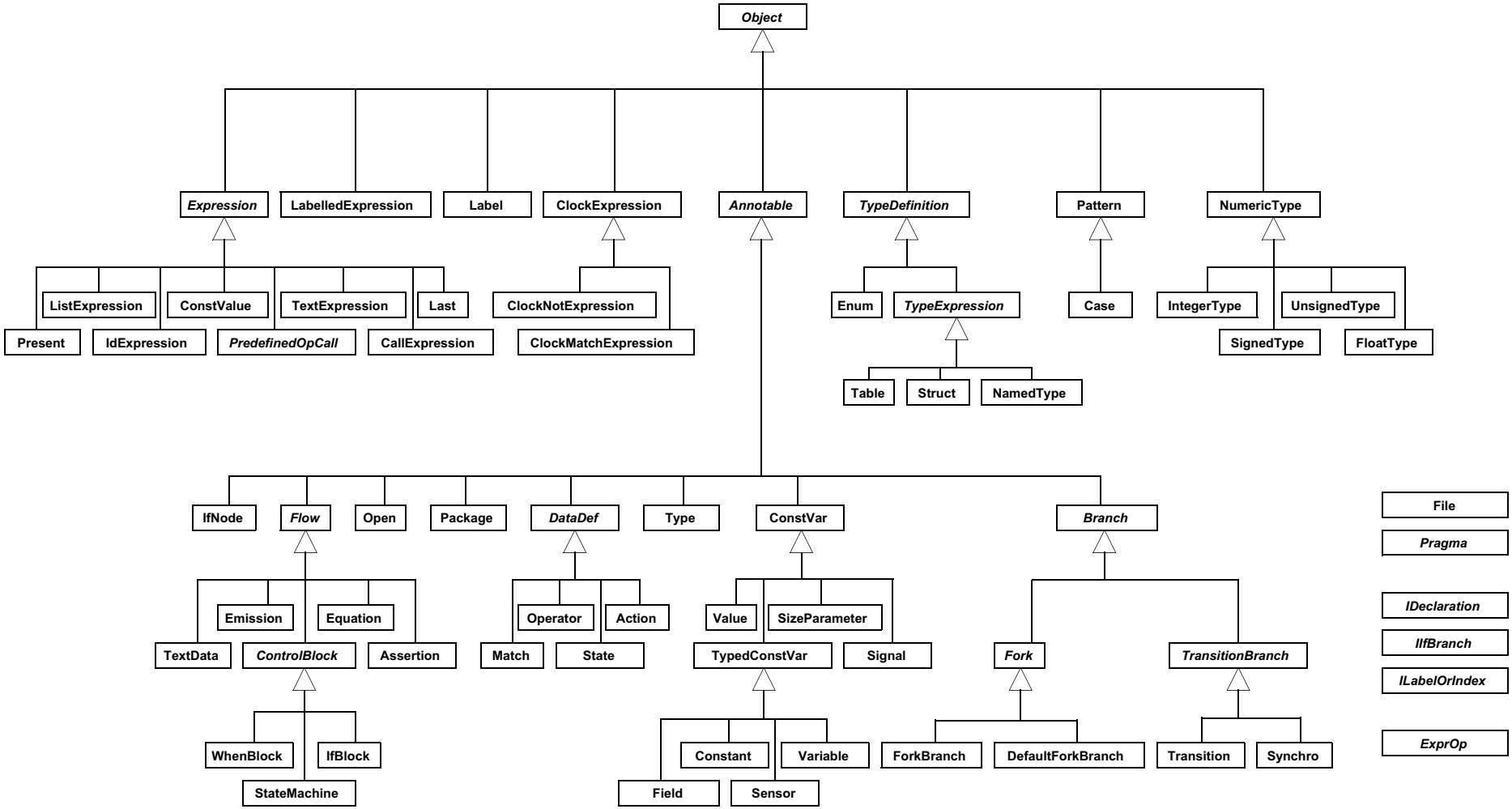
Expressions



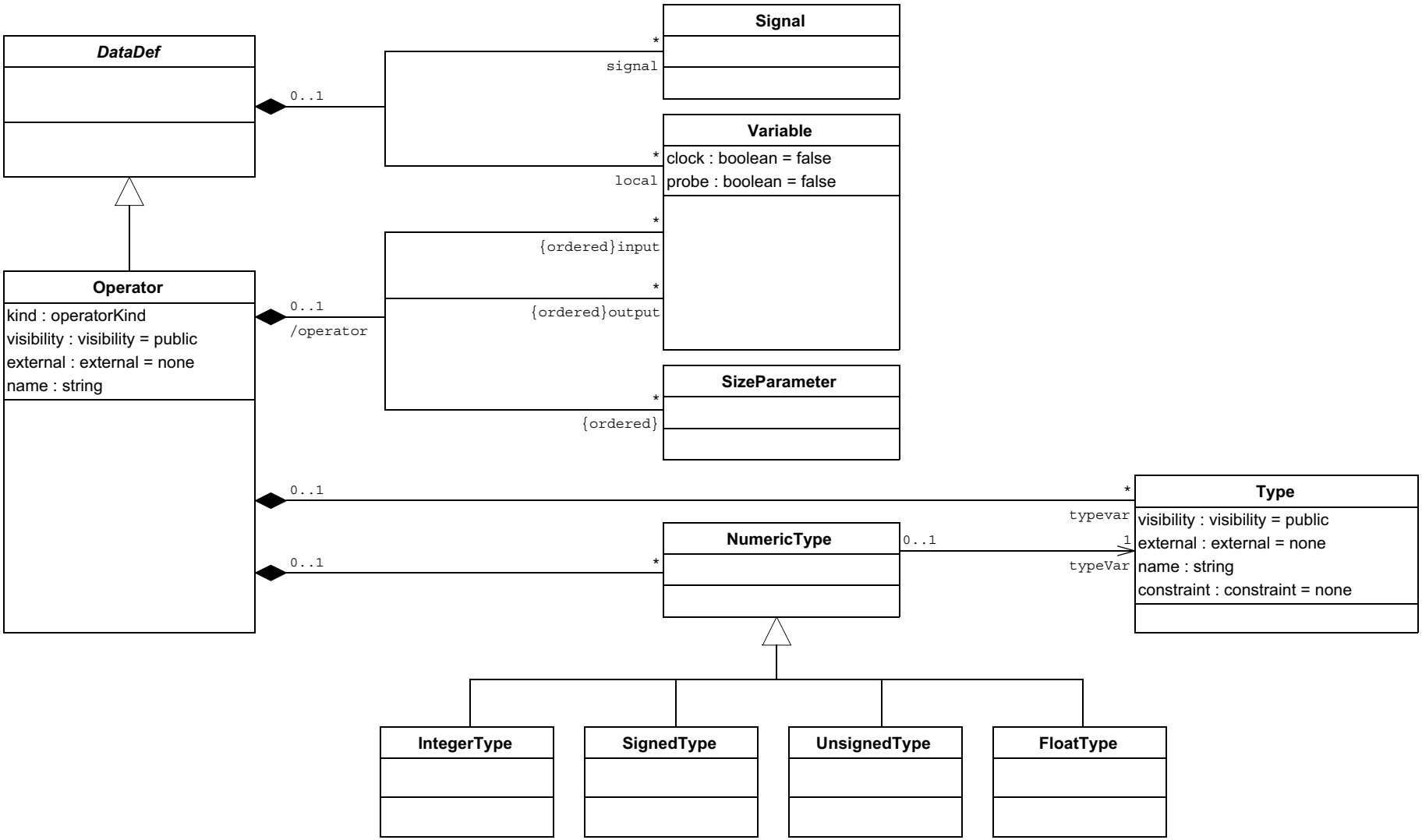
Flow Operators



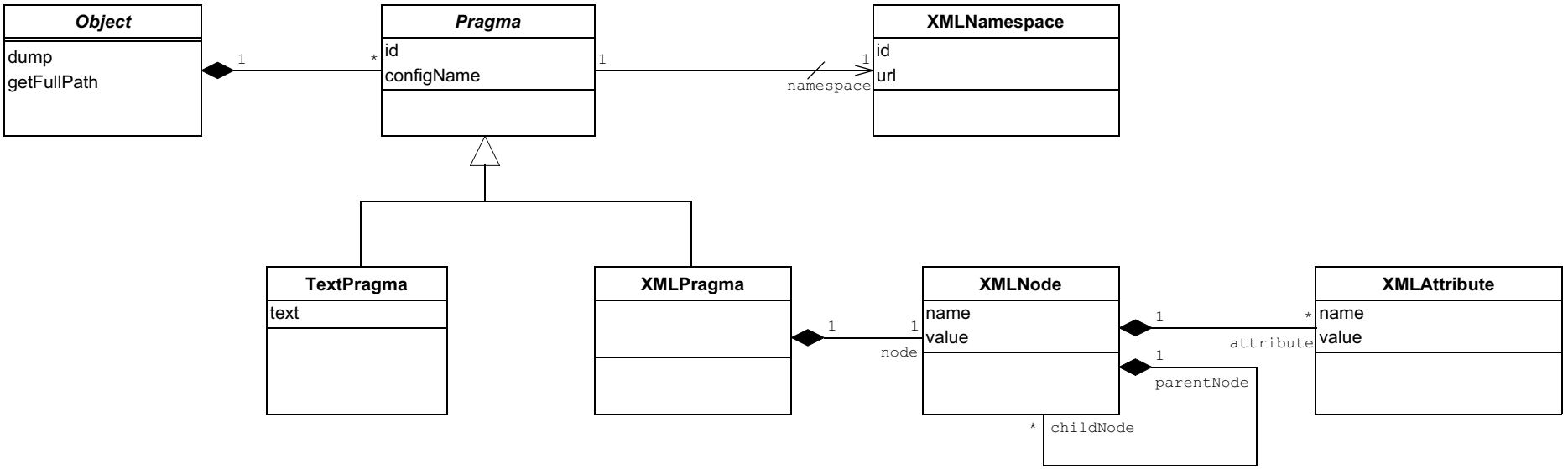
Inheritance



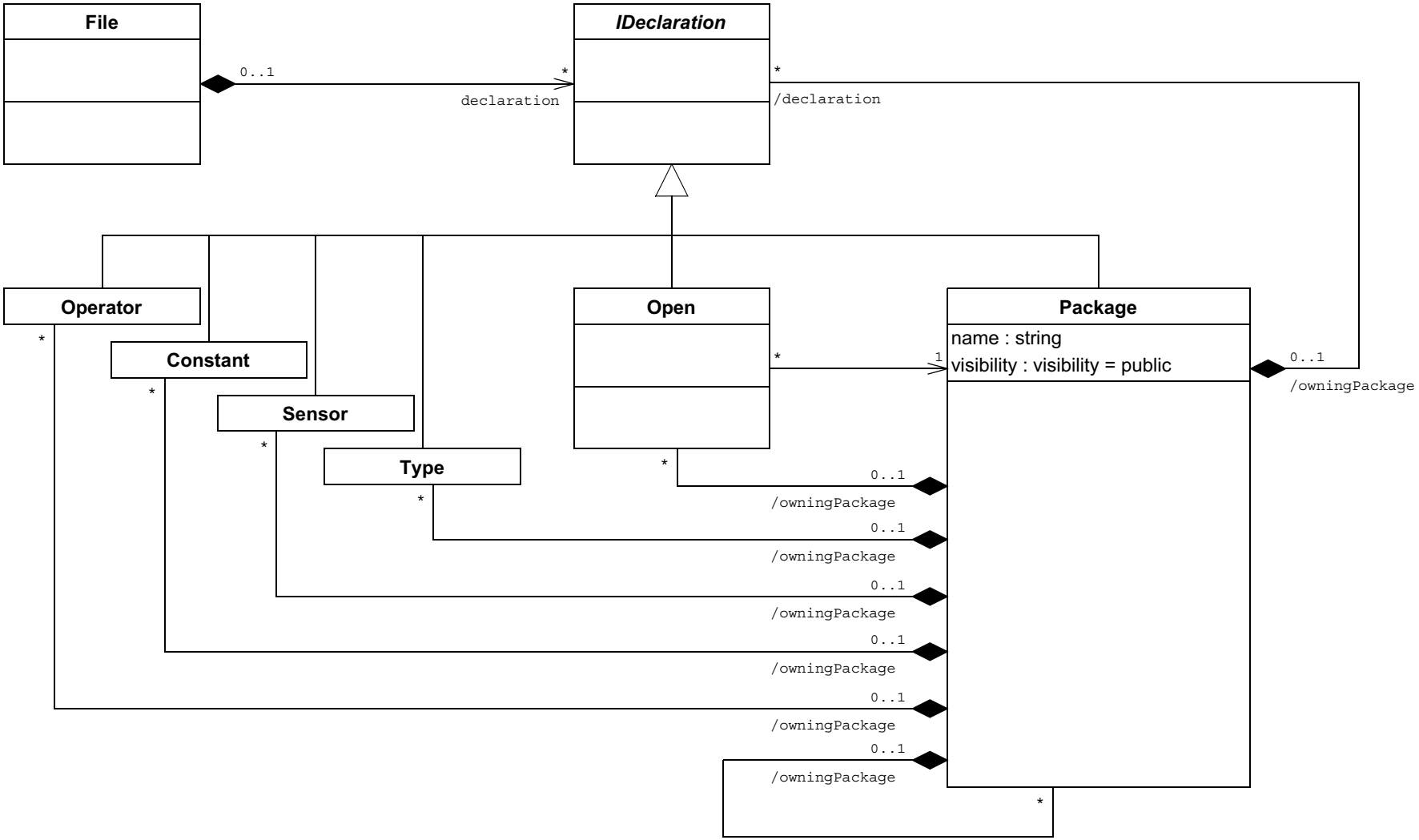
Operators


DataDef
Signal
*
signal
0..1
Variable
clock : boolean = false
probe : boolean = false
*
local
*
{ordered}input
Operator
kind : operatorKind
visibility : visibility = public
external : external = none
name : string
0..1
/operator
*
{ordered}output
SizeParameter
*
{ordered}
0..1
*
typevar
Type
visibility : visibility = public
external : external = none
name : string
constraint : constraint = none
NumericType
0..1
1
typeVar
0..1
*
IntegerType
SignedType
UnsignedType
FloatType

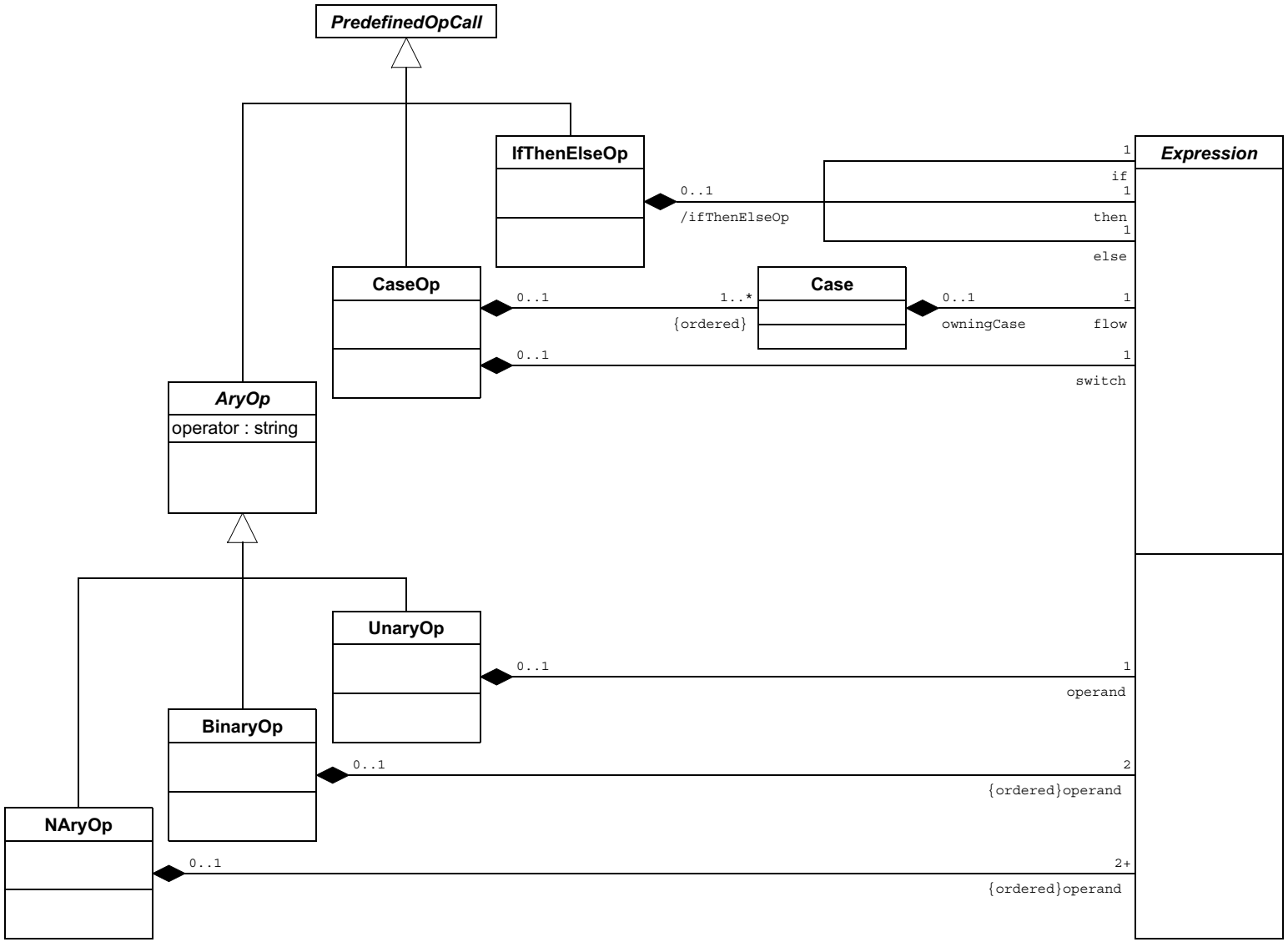
Pragmas


Object
dump
getFullPath
Pragma
id
configName
XMLNamespace
id
url
1
*
1
1
namespace
TextPragma
text
XMLPragma
XMLNode
name
value
XMLAttribute
name
value
1
1
node
1
*
attribute
1
parentNode
*
childNode

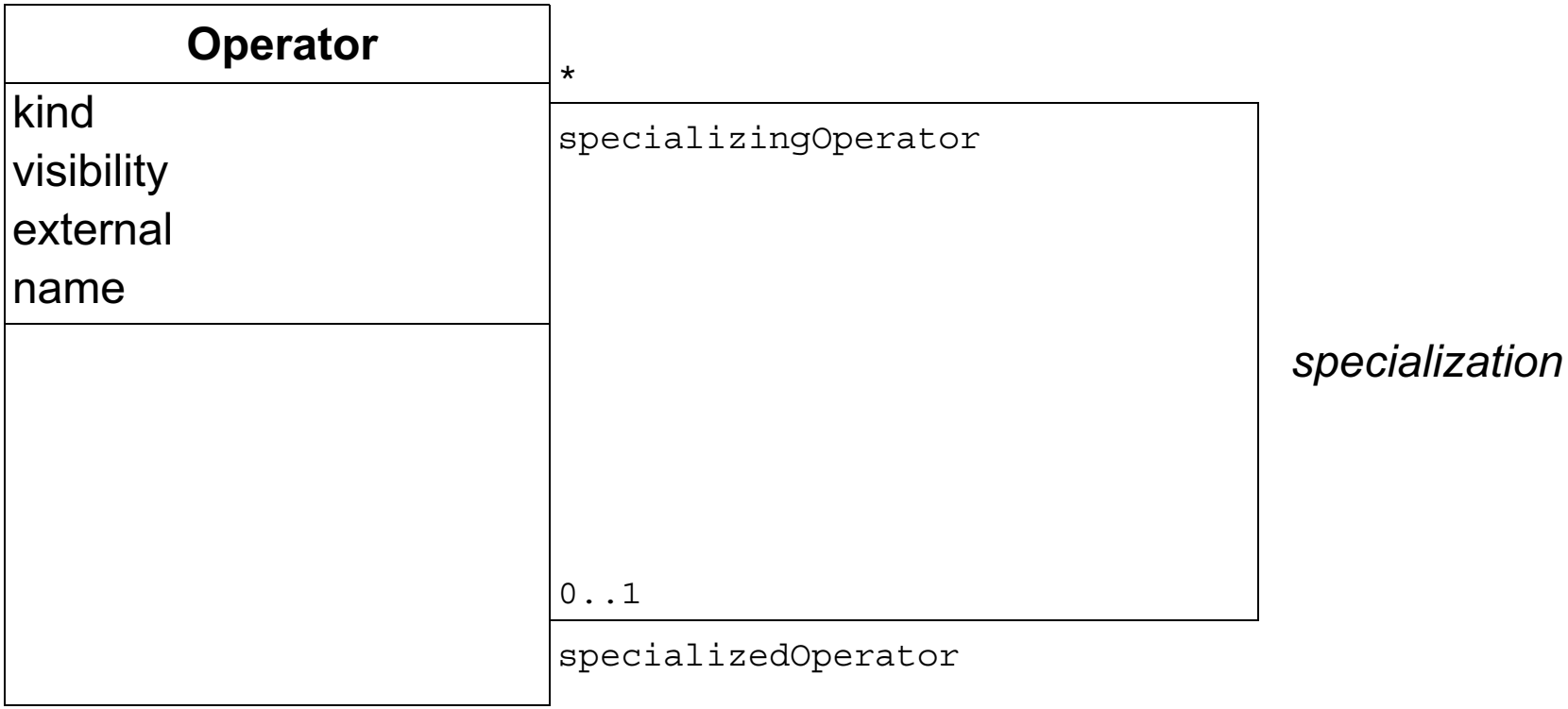
Roots


File
0..1
declaration
*
IDeclaration
*
/declaration
Operator
Constant
Sensor
Type
Open
*
1
Package
name : string
visibility : visibility = public
0..1
/owningPackage
0..1
/owningPackage
0..1
/owningPackage
0..1
/owningPackage
0..1
/owningPackage
0..1
/owningPackage
*

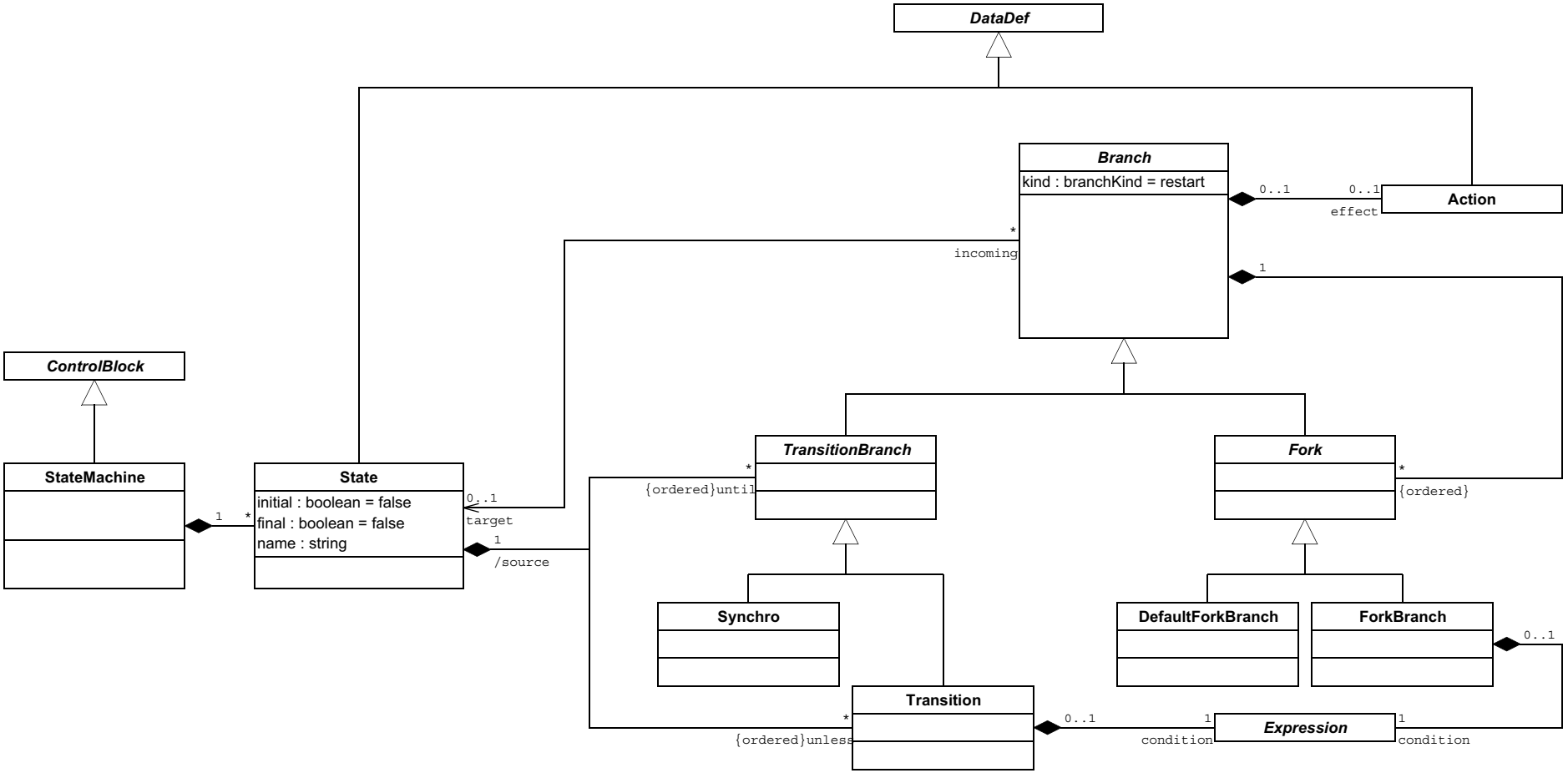
Simple and Control Operators



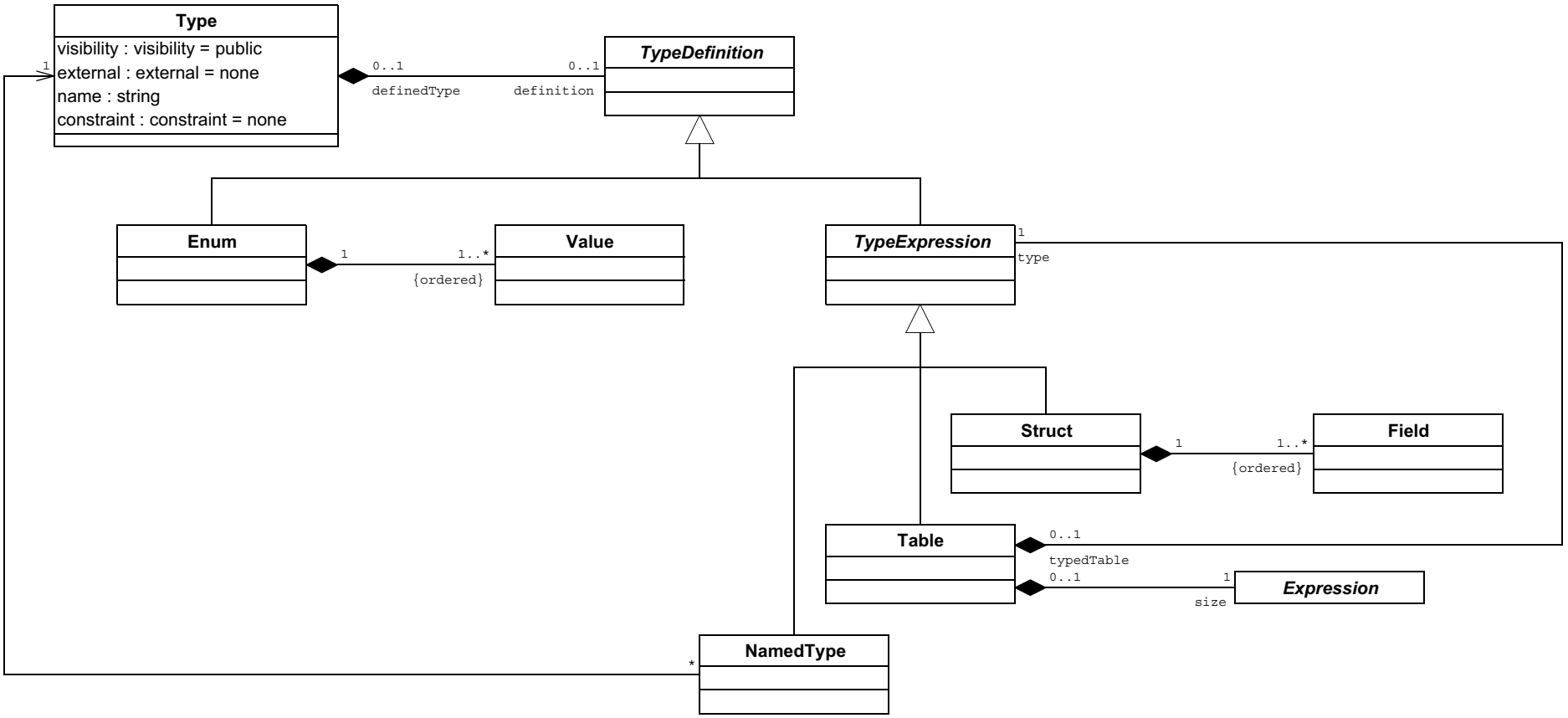
Specialization



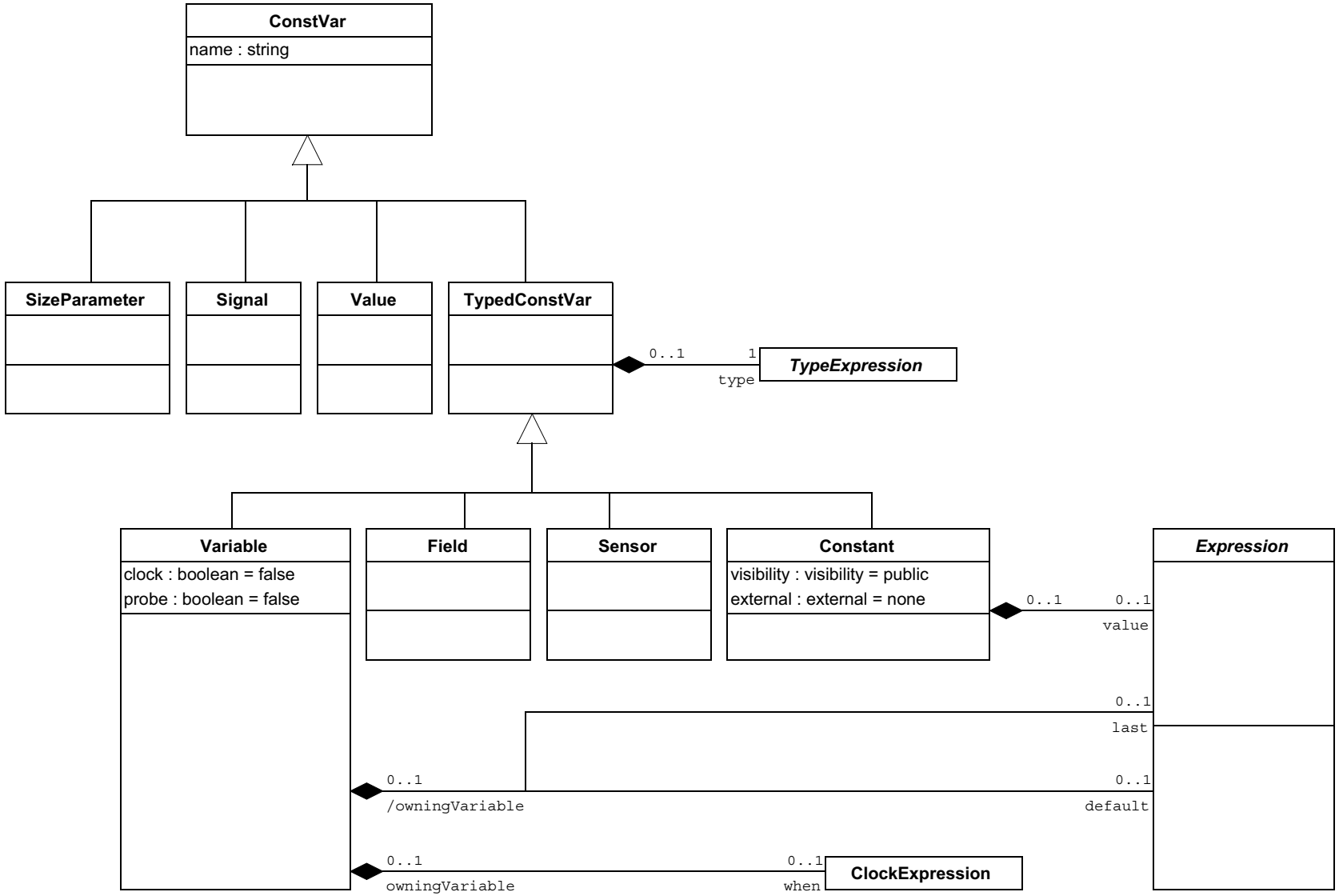
State Machines



Type


Type
visibility : visibility = public
external : external = none
name : string
constraint : constraint = none
1
0..1
definedType
0..1
definition
TypeDefinition
Enum
1
1..*
{ordered}
Value
TypeExpression
1
type
Struct
1
1..*
{ordered}
Field
Table
0..1
typedTable
0..1
1
size
Expression
NamedType
*

Variable

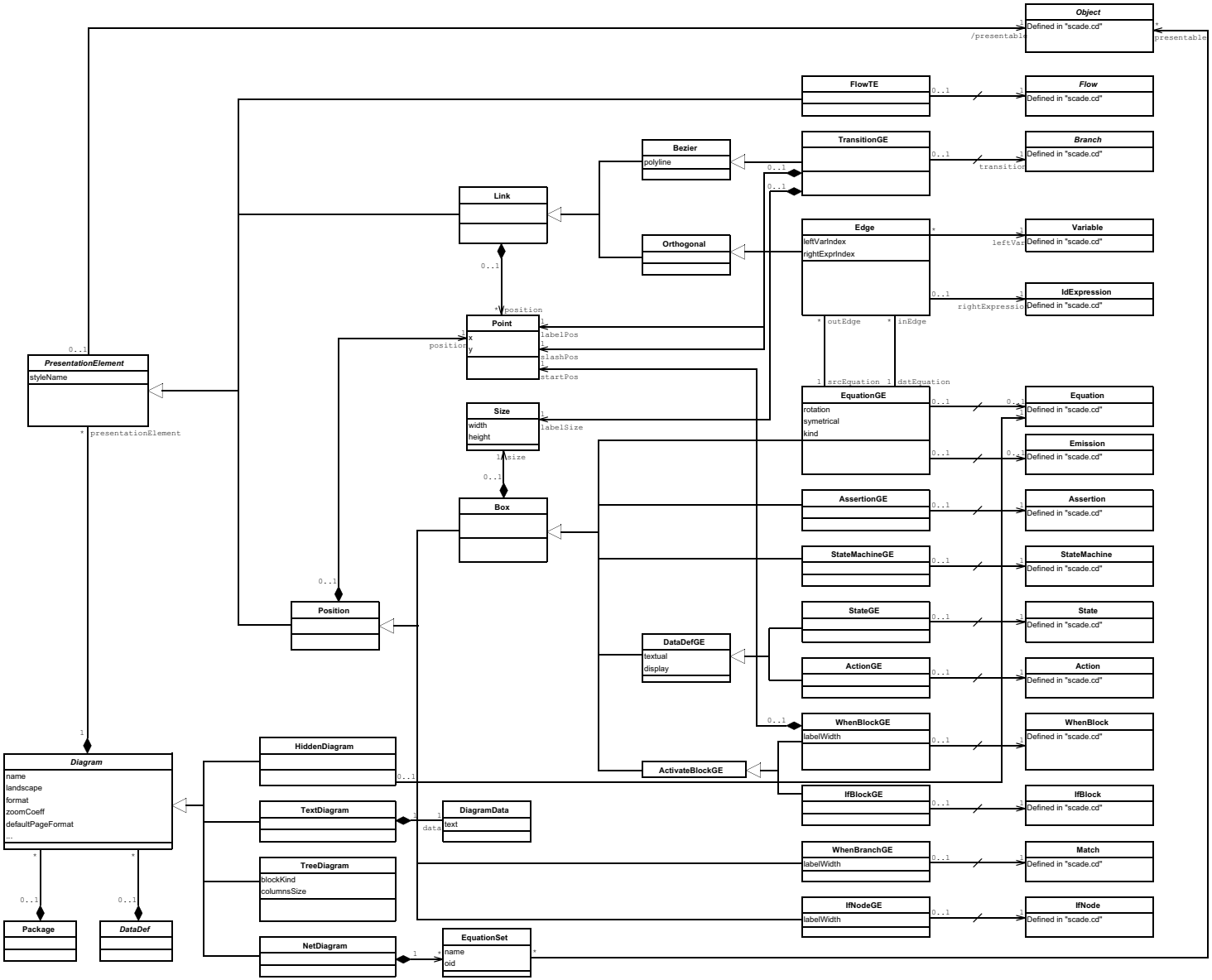

ConstVar
name : string
SizeParameter
Signal
Value
TypedConstVar
0..1
1
type
TypeExpression
Variable
clock : boolean = false
probe : boolean = false
Field
Sensor
Constant
visibility : visibility = public
external : external = none
0..1
0..1
value
Expression
0..1
last
0..1
/owningVariable
0..1
default
0..1
owningVariable
0..1
when
ClockExpression

19 / Scade Graphics Metamodels

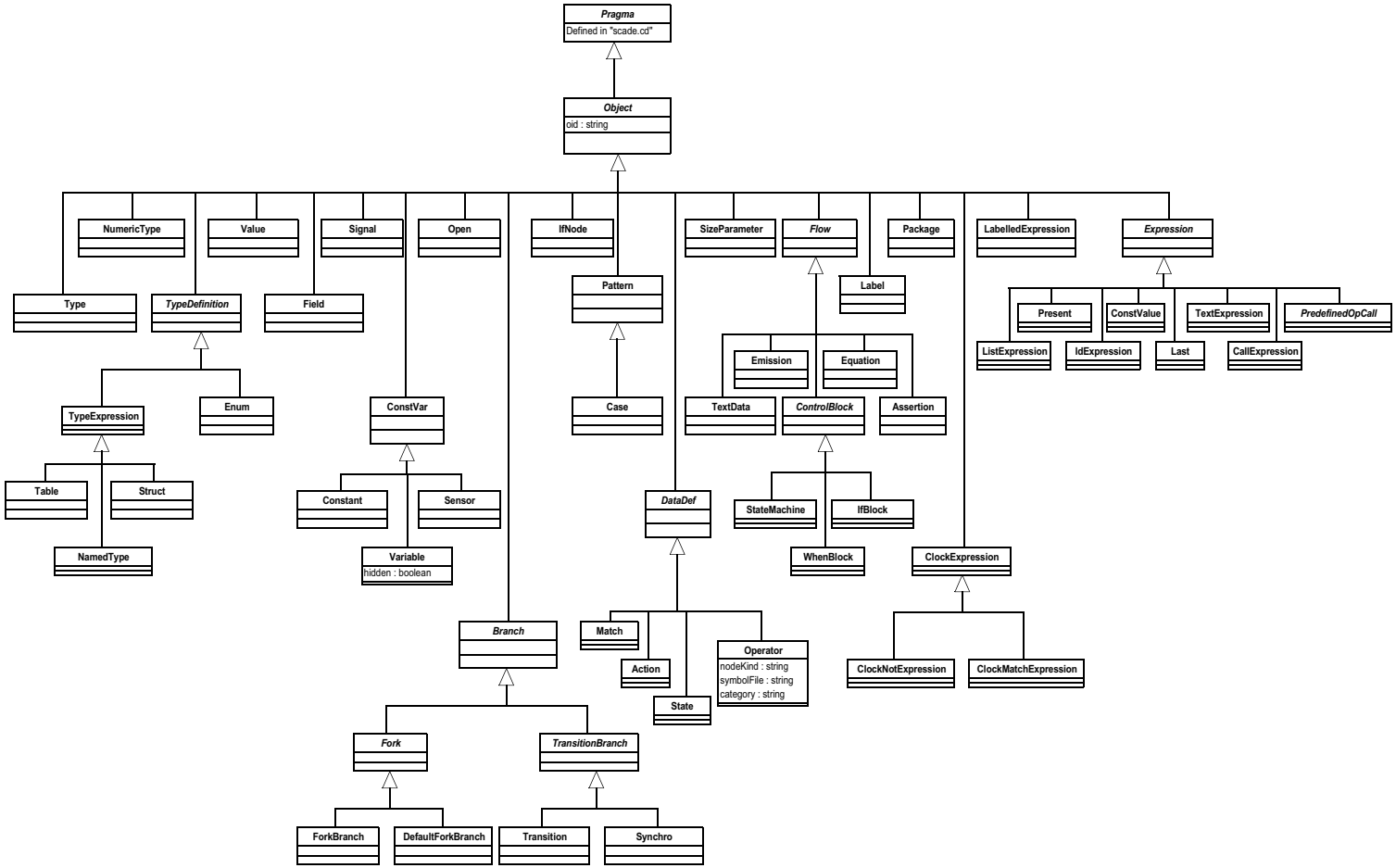
These metamodels present the data structures that give access to the graphic constructs of the Scade language using Java API:

- "Diagrams"
- "Graphical Pragmas"

Diagrams


Object
Defined in "scade.cd"
/presentable
presentable
FlowTE
Flow
Defined in "scade.cd"
Bezier
polyline
TransitionGE
Branch
Defined in "scade.cd"
transition
Link
Orthogonal
Edge
leftVarIndex
rightExprIndex
Variable
Defined in "scade.cd"
leftVar
IdExpression
Defined in "scade.cd"
rightExpression
Point
x
y
position
labelPos
slashPos
startPos
outEdge
inEdge
srcEquation
dstEquation
PresentationElement
styleName
presentationElement
Size
width
height
labelSize
Box
EquationGE
rotation
symetrical
kind
Equation
Defined in "scade.cd"
Emission
Defined in "scade.cd"
AssertionGE
Assertion
Defined in "scade.cd"
StateMachineGE
StateMachine
Defined in "scade.cd"
Position
StateGE
State
Defined in "scade.cd"
DataDefGE
textual
display
ActionGE
Action
Defined in "scade.cd"
WhenBlockGE
labelWidth
WhenBlock
Defined in "scade.cd"
ActivateBlockGE
IfBlockGE
IfBlock
Defined in "scade.cd"
WhenBranchGE
labelWidth
Match
Defined in "scade.cd"
IfNodeGE
labelWidth
IfNode
Defined in "scade.cd"
Diagram
name
landscape
format
zoomCoeff
defaultPageFormat
...
HiddenDiagram
TextDiagram
DiagramData
text
data
TreeDiagram
blockKind
columnsSize
NetDiagram
EquationSet
name
oid
Package
DataDef

Graphical Pragmas



Reference Card

Part 5

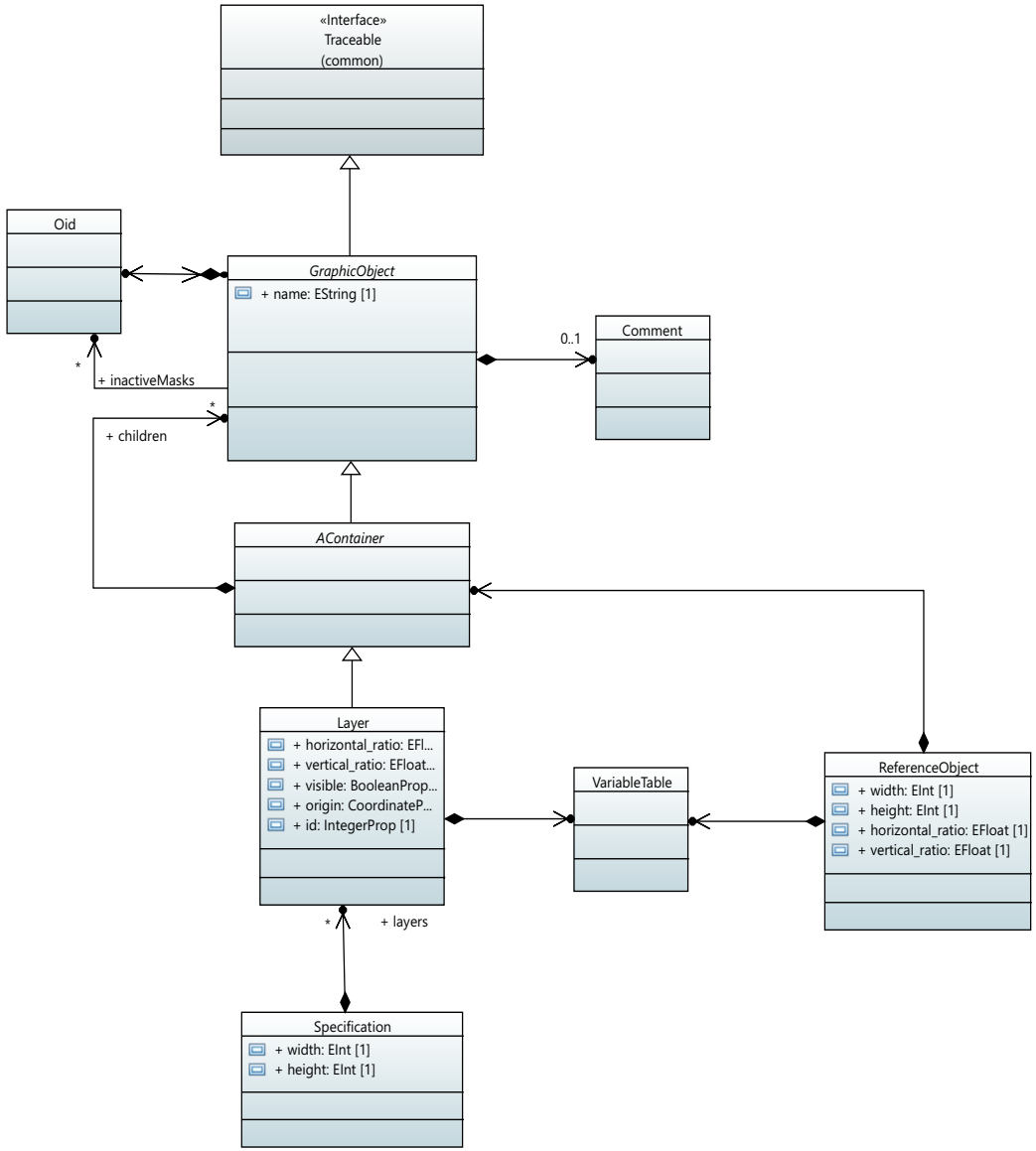
SCADE Display Metamodels

20 / SCADE Display Metamodels

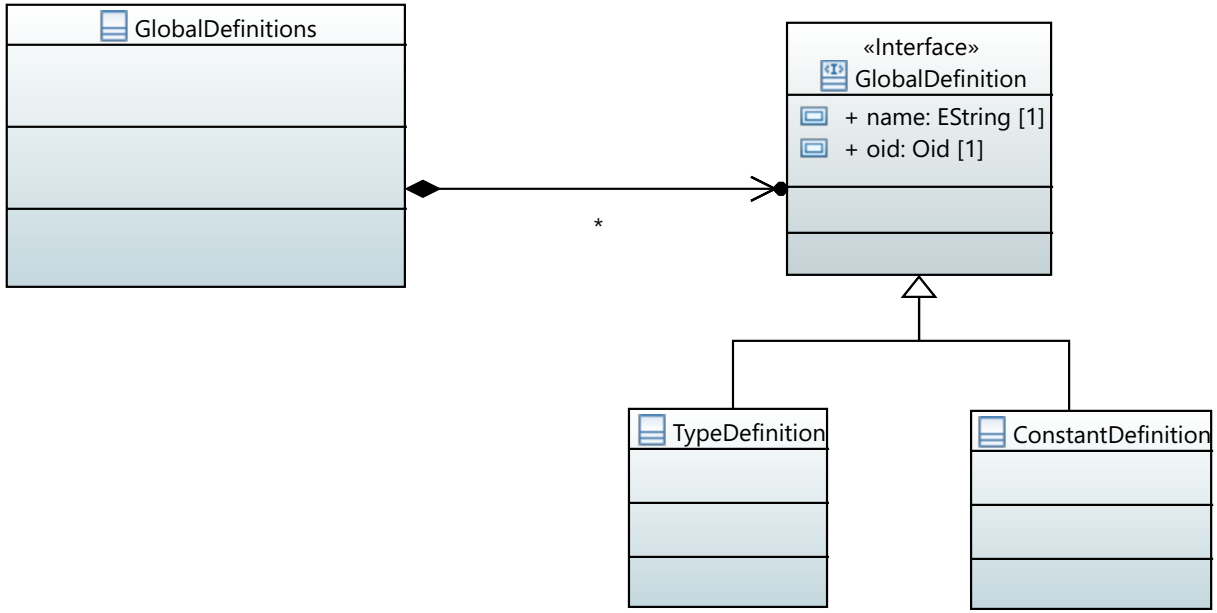
These metamodels present the data structure that give access to SCADE Display models using Python API or Java API:

- "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"

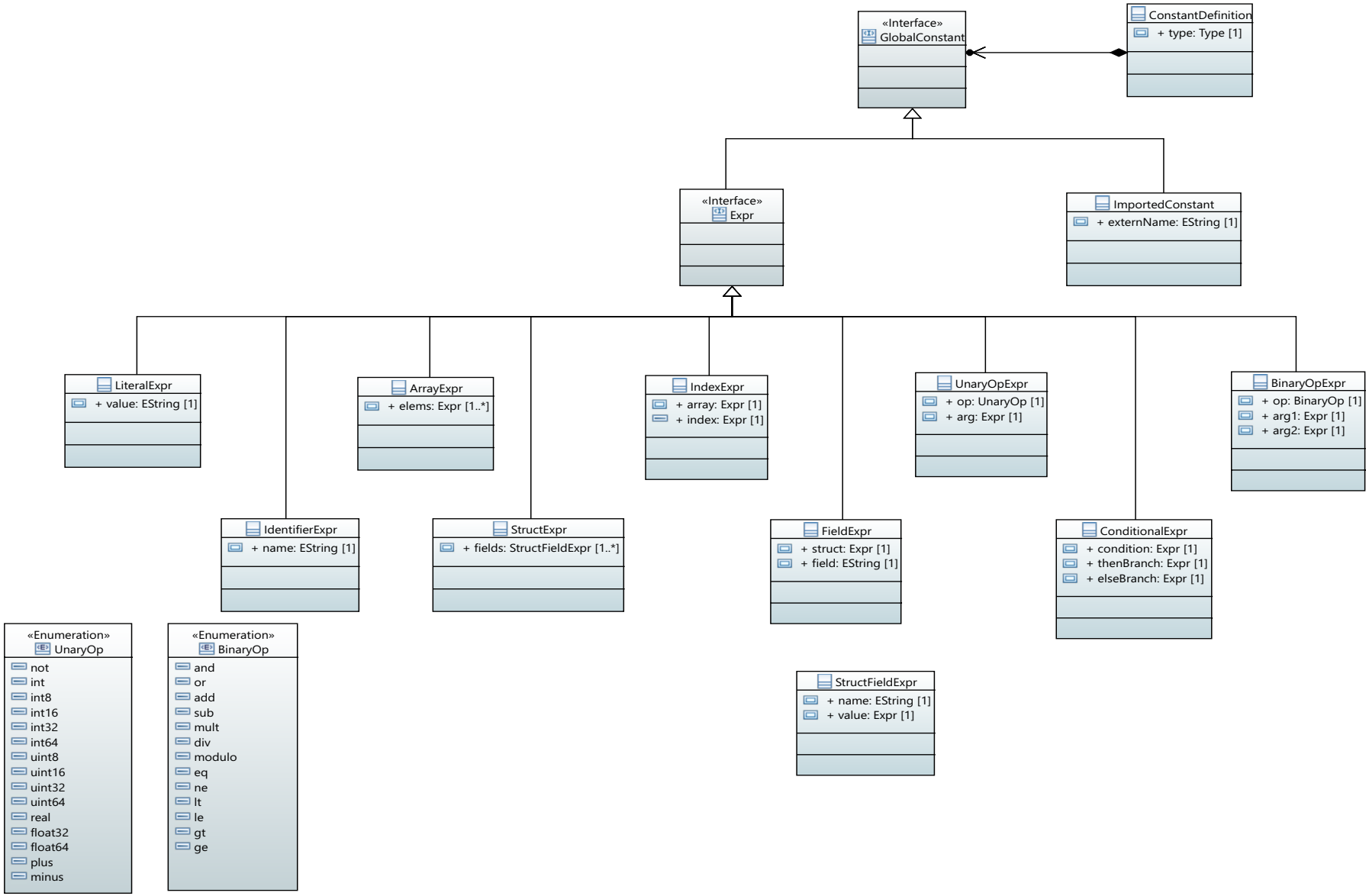
Specification and Reference Object Model



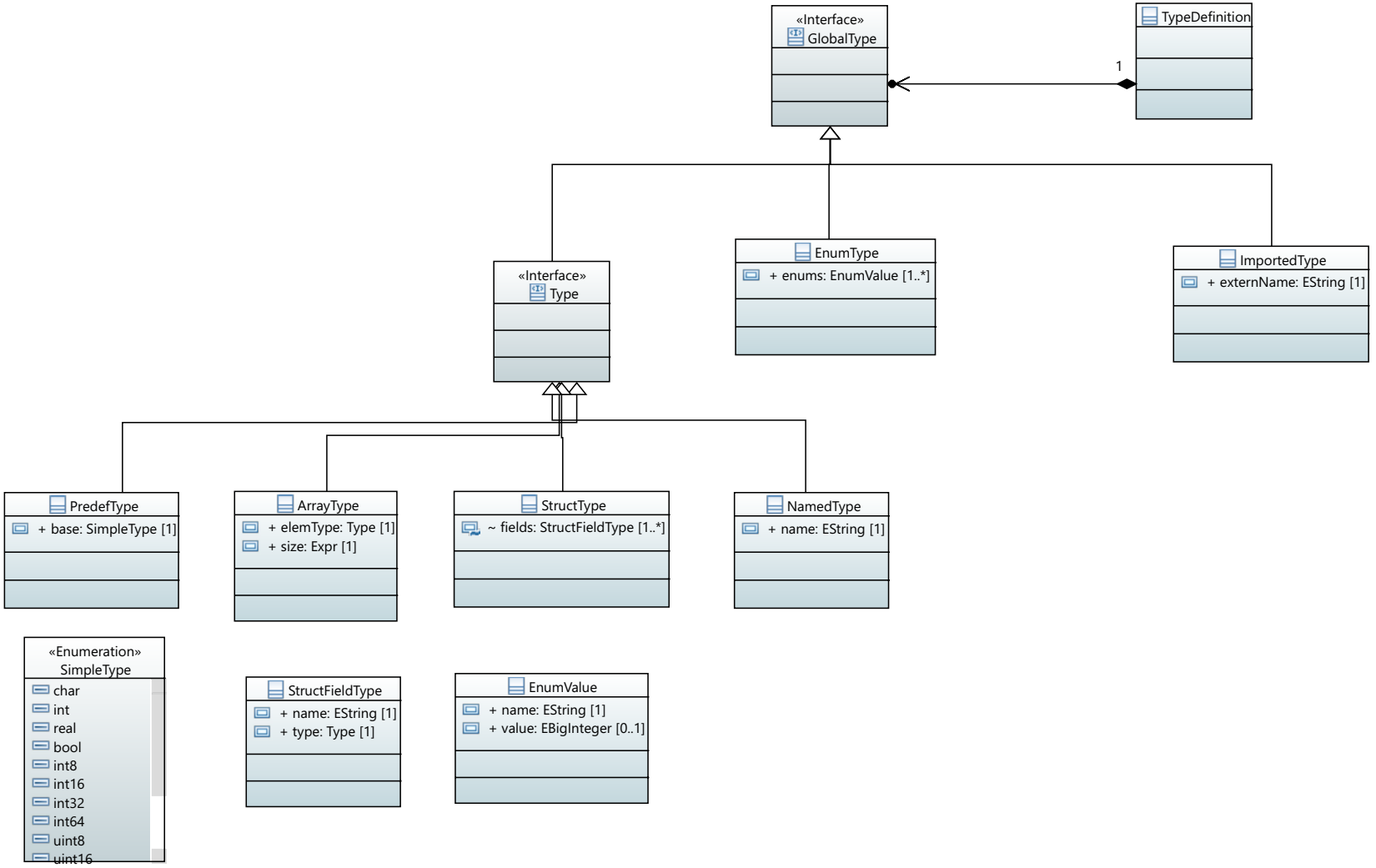
Global Definitions



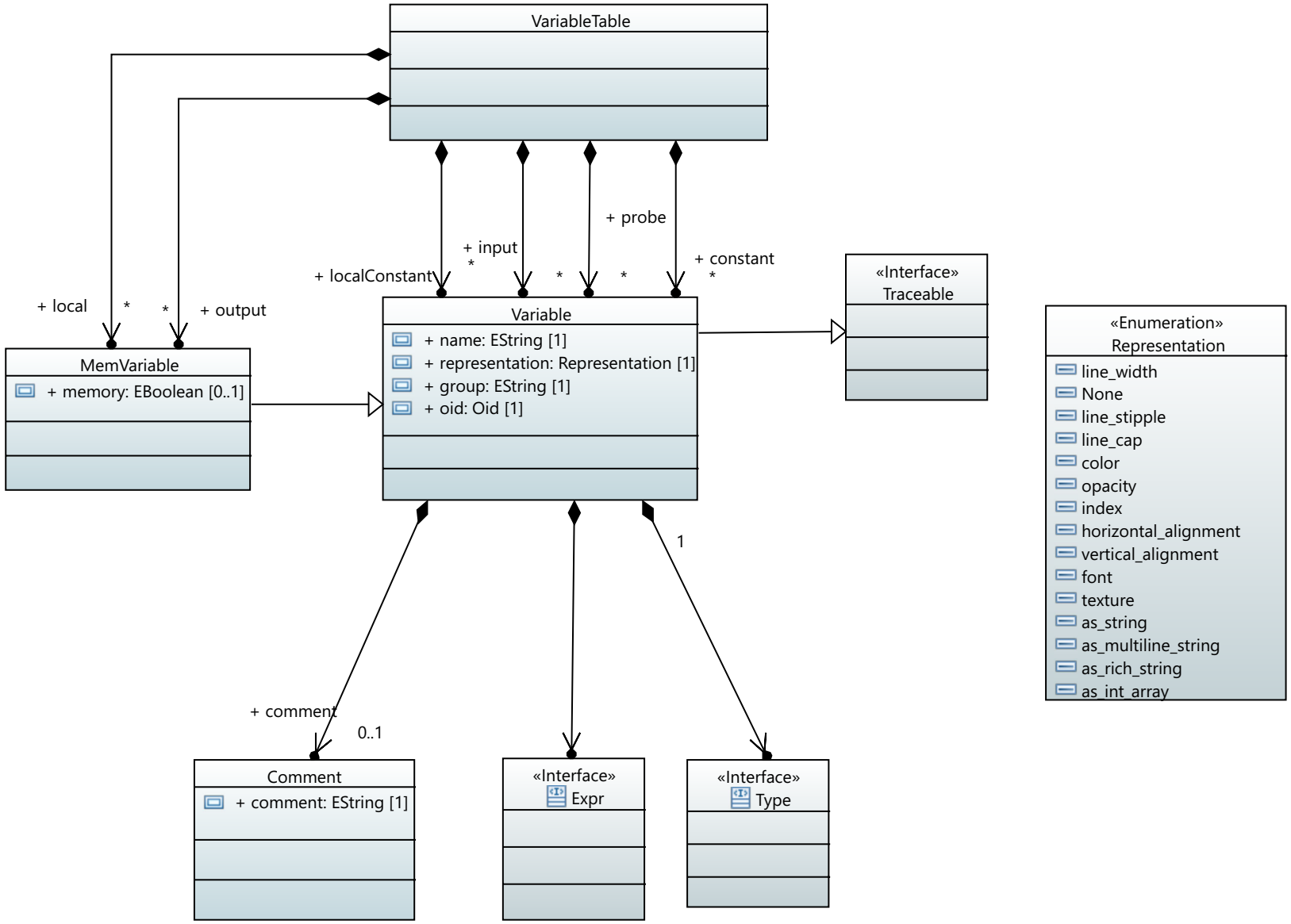
Constants


«Interface»
GlobalConstant
ConstantDefinition
+ type: Type [1]
«Interface»
Expr
ImportedConstant
+ externName: EString [1]
LiteralExpr
+ value: EString [1]
ArrayExpr
+ elems: Expr [1..*]
IndexExpr
+ array: Expr [1]
+ index: Expr [1]
UnaryOpExpr
+ op: UnaryOp [1]
+ arg: Expr [1]
BinaryOpExpr
+ op: BinaryOp [1]
+ arg1: Expr [1]
+ arg2: Expr [1]
IdentifierExpr
+ name: EString [1]
StructExpr
+ fields: StructFieldExpr [1..*]
FieldExpr
+ struct: Expr [1]
+ field: EString [1]
ConditionalExpr
+ condition: Expr [1]
+ thenBranch: Expr [1]
+ elseBranch: Expr [1]
«Enumeration»
UnaryOp
not
int
int8
int16
int32
int64
uint8
uint16
uint32
uint64
real
float32
float64
plus
minus
«Enumeration»
BinaryOp
and
or
add
sub
mult
div
modulo
eq
ne
lt
le
gt
ge
StructFieldExpr
+ name: EString [1]
+ value: Expr [1]

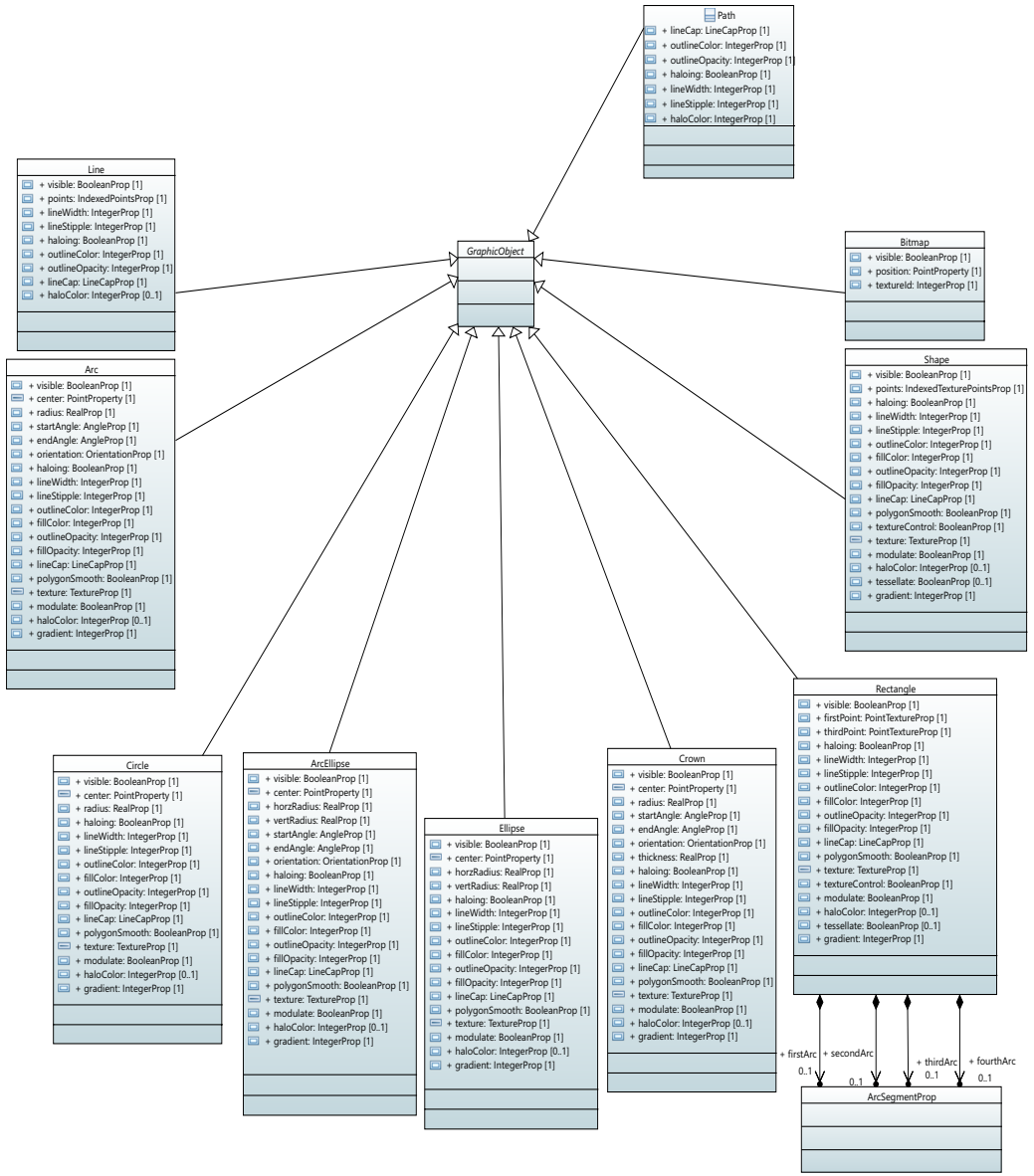
Types


«Interface»
GlobalType
TypeDefinition
1
«Interface»
Type
EnumType
+ enums: EnumValue [1..*]
ImportedType
+ externName: EString [1]
PredefType
+ base: SimpleType [1]
ArrayType
+ elemType: Type [1]
+ size: Expr [1]
StructType
~ fields: StructFieldType [1..*]
NamedType
+ name: EString [1]
«Enumeration»
SimpleType
char
int
real
bool
int8
int16
int32
int64
uint8
uint16
StructFieldType
+ name: EString [1]
+ type: Type [1]
EnumValue
+ name: EString [1]
+ value: EBigInteger [0..1]

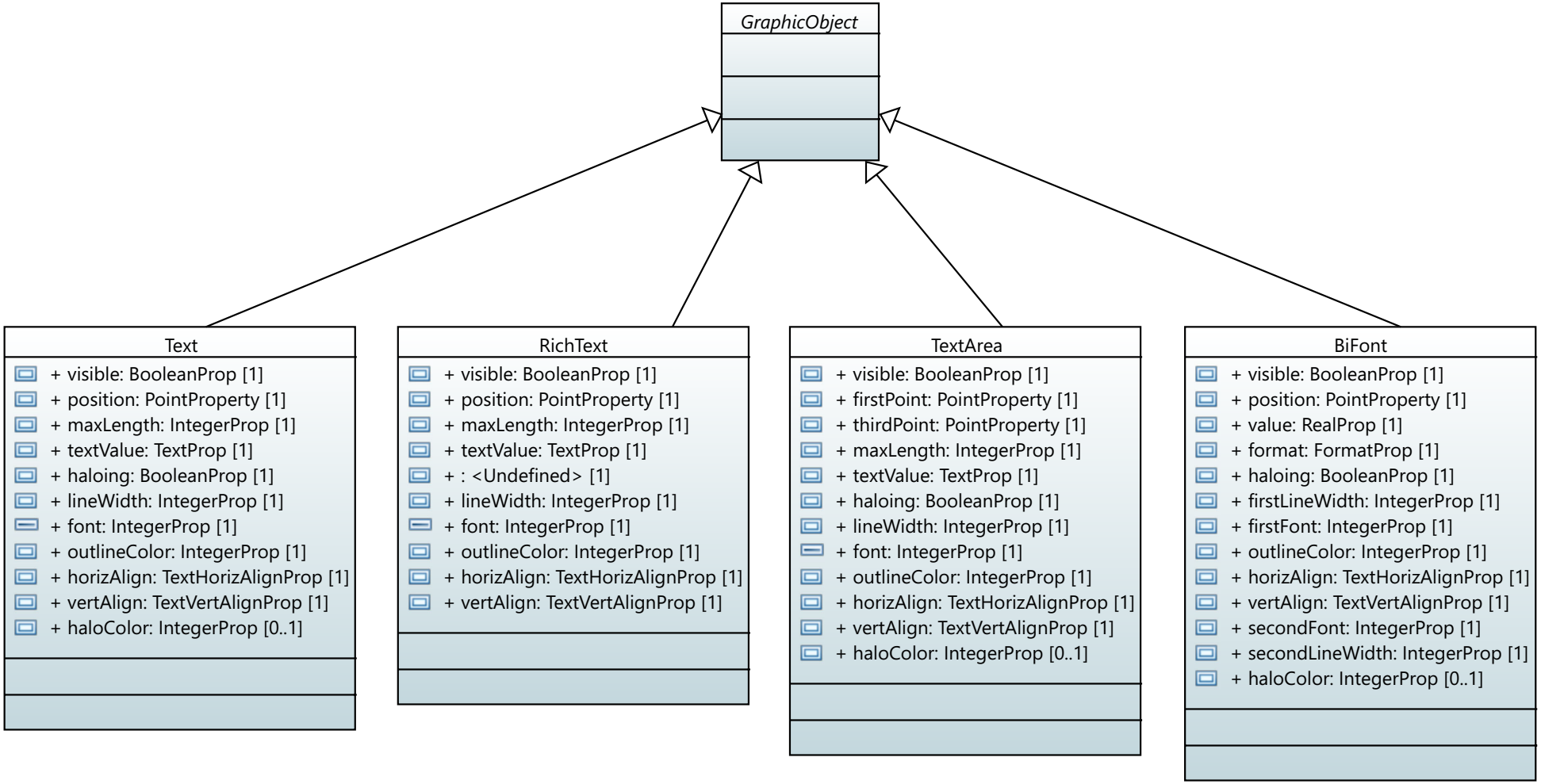
Variable Dictionary



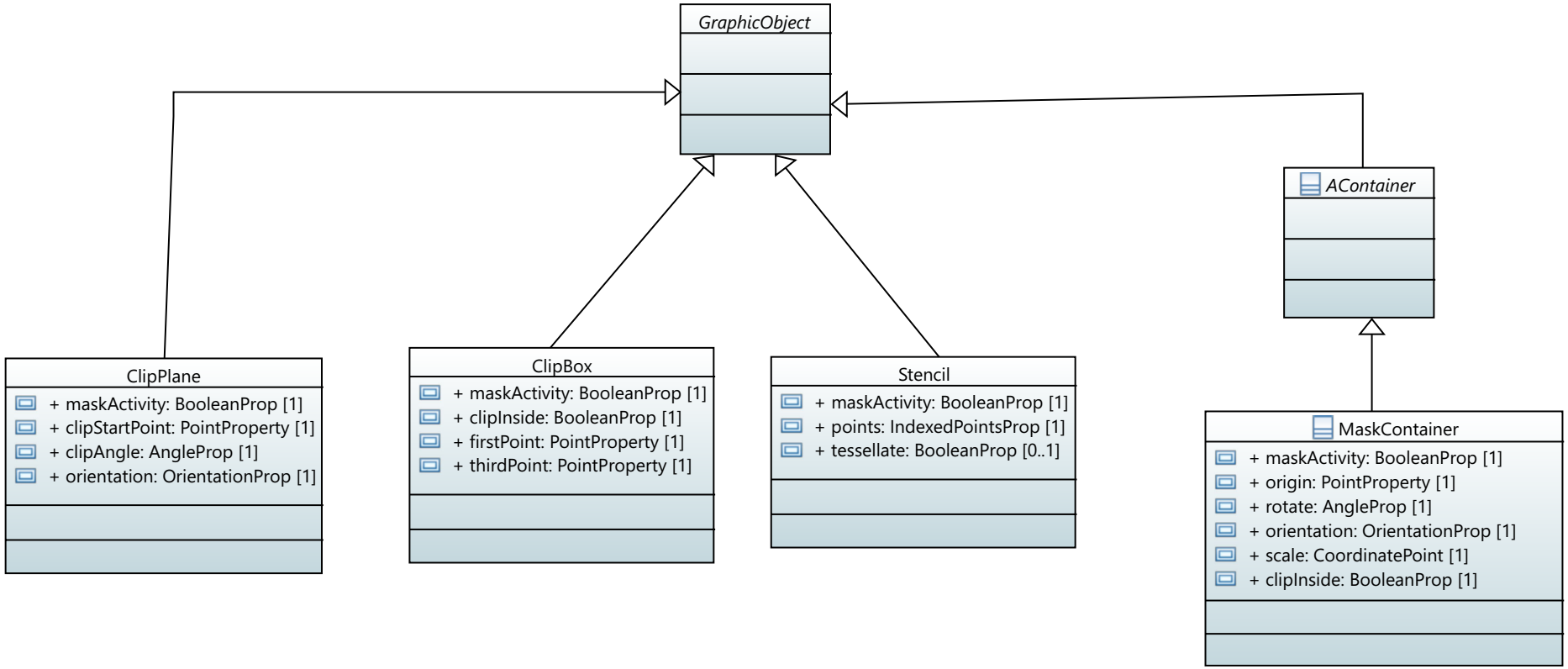
Graphic Primitives



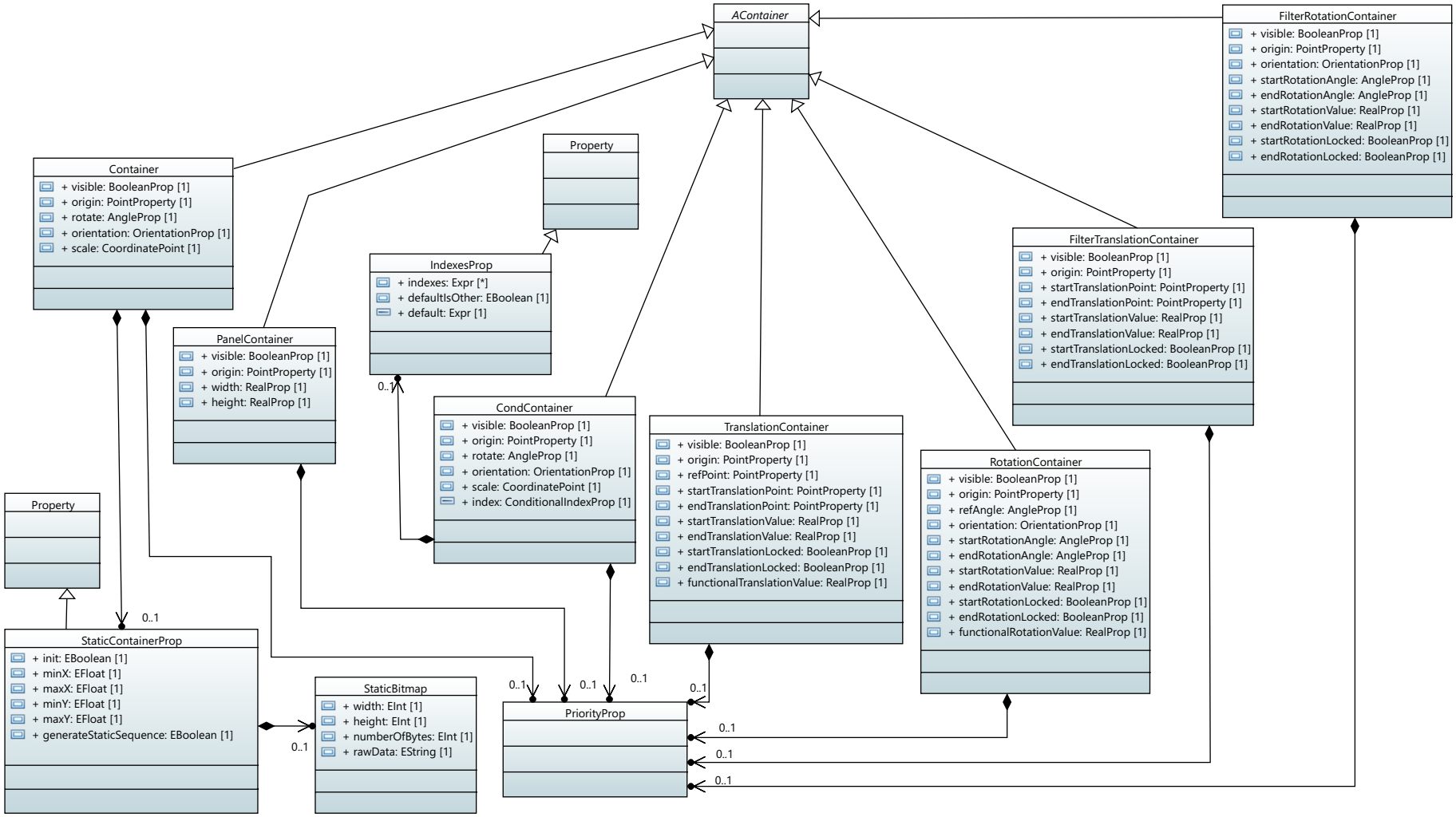
Text Primitives



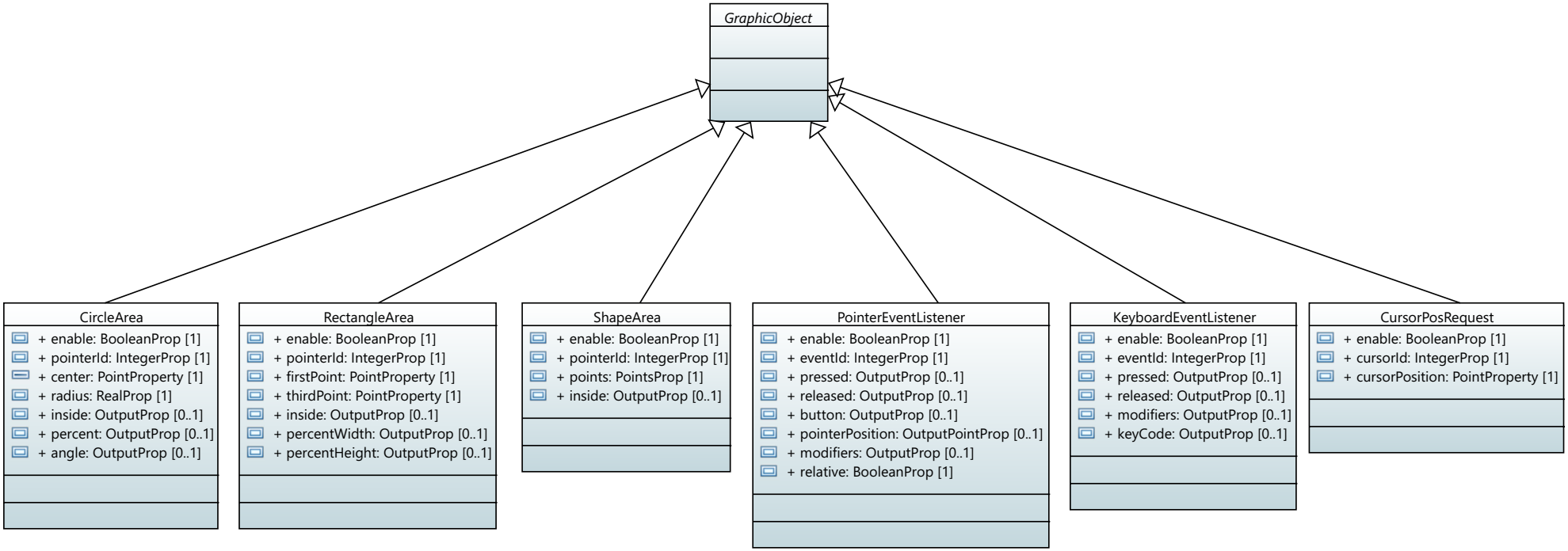
Masks


GraphicObject
AContainer
ClipPlane
+ maskActivity: BooleanProp [1]
+ clipStartPoint: PointProperty [1]
+ clipAngle: AngleProp [1]
+ orientation: OrientationProp [1]
ClipBox
+ maskActivity: BooleanProp [1]
+ clipInside: BooleanProp [1]
+ firstPoint: PointProperty [1]
+ thirdPoint: PointProperty [1]
Stencil
+ maskActivity: BooleanProp [1]
+ points: IndexedPointsProp [1]
+ tessellate: BooleanProp [0..1]
MaskContainer
+ maskActivity: BooleanProp [1]
+ origin: PointProperty [1]
+ rotate: AngleProp [1]
+ orientation: OrientationProp [1]
+ scale: CoordinatePoint [1]
+ clipInside: BooleanProp [1]

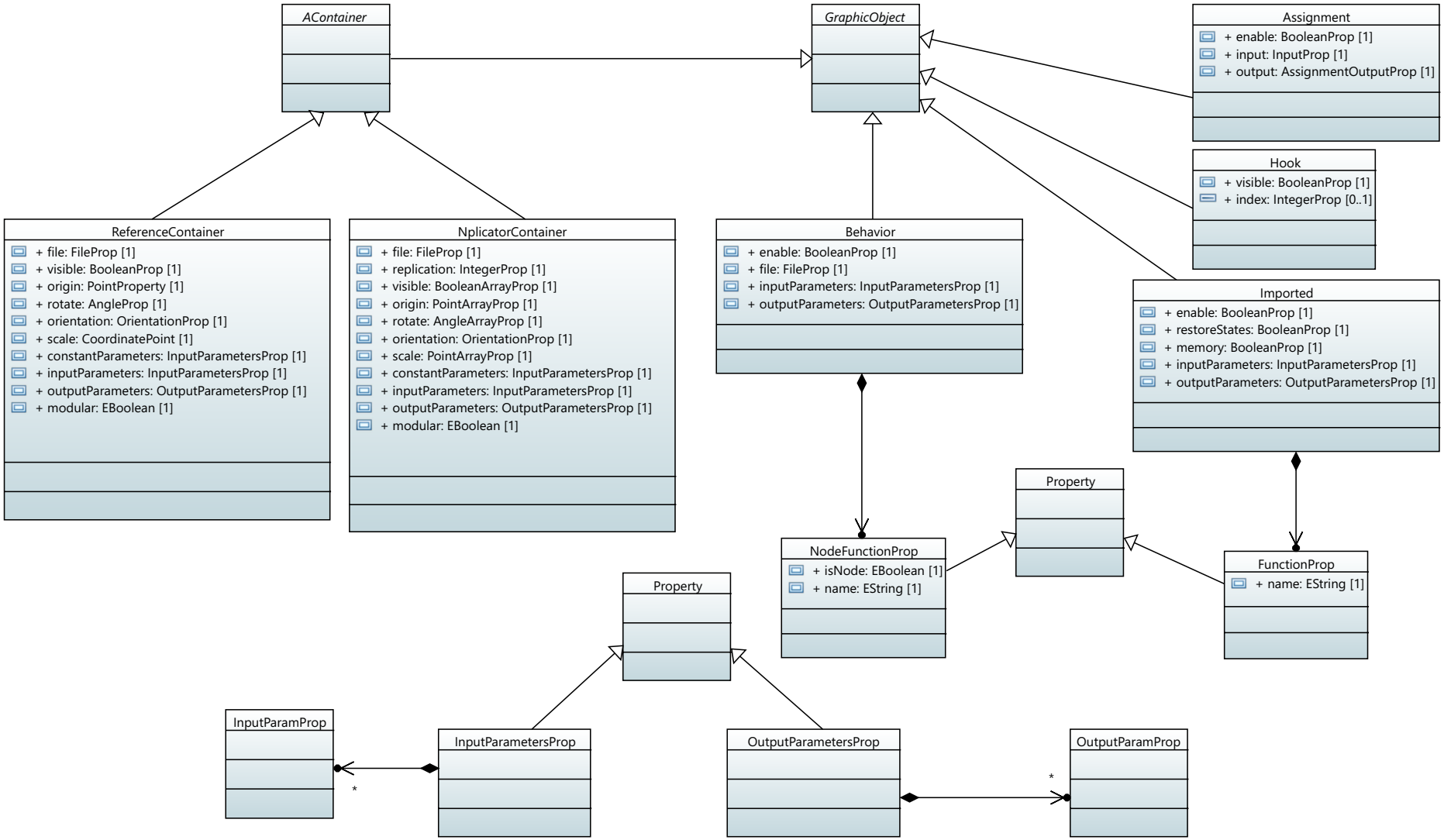
Containers


AContainer
Container
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ rotate: AngleProp [1]
+ orientation: OrientationProp [1]
+ scale: CoordinatePoint [1]
Property
IndexesProp
+ indexes: Expr [*]
+ defaultIsOther: EBoolean [1]
+ default: Expr [1]
FilterRotationContainer
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ orientation: OrientationProp [1]
+ startRotationAngle: AngleProp [1]
+ endRotationAngle: AngleProp [1]
+ startRotationValue: RealProp [1]
+ endRotationValue: RealProp [1]
+ startRotationLocked: BooleanProp [1]
+ endRotationLocked: BooleanProp [1]
FilterTranslationContainer
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ startTranslationPoint: PointProperty [1]
+ endTranslationPoint: PointProperty [1]
+ startTranslationValue: RealProp [1]
+ endTranslationValue: RealProp [1]
+ startTranslationLocked: BooleanProp [1]
+ endTranslationLocked: BooleanProp [1]
PanelContainer
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ width: RealProp [1]
+ height: RealProp [1]
CondContainer
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ rotate: AngleProp [1]
+ orientation: OrientationProp [1]
+ scale: CoordinatePoint [1]
+ index: ConditionalIndexProp [1]
TranslationContainer
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ refPoint: PointProperty [1]
+ startTranslationPoint: PointProperty [1]
+ endTranslationPoint: PointProperty [1]
+ startTranslationValue: RealProp [1]
+ endTranslationValue: RealProp [1]
+ startTranslationLocked: BooleanProp [1]
+ endTranslationLocked: BooleanProp [1]
+ functionalTranslationValue: RealProp [1]
RotationContainer
+ visible: BooleanProp [1]
+ origin: PointProperty [1]
+ refAngle: AngleProp [1]
+ orientation: OrientationProp [1]
+ startRotationAngle: AngleProp [1]
+ endRotationAngle: AngleProp [1]
+ startRotationValue: RealProp [1]
+ endRotationValue: RealProp [1]
+ startRotationLocked: BooleanProp [1]
+ endRotationLocked: BooleanProp [1]
+ functionalRotationValue: RealProp [1]
Property
StaticContainerProp
+ init: EBoolean [1]
+ minX: EFloat [1]
+ maxX: EFloat [1]
+ minY: EFloat [1]
+ maxY: EFloat [1]
+ generateStaticSequence: EBoolean [1]
StaticBitmap
+ width: EInt [1]
+ height: EInt [1]
+ numberOfBytes: EInt [1]
+ rawData: EString [1]
PriorityProp
0..1

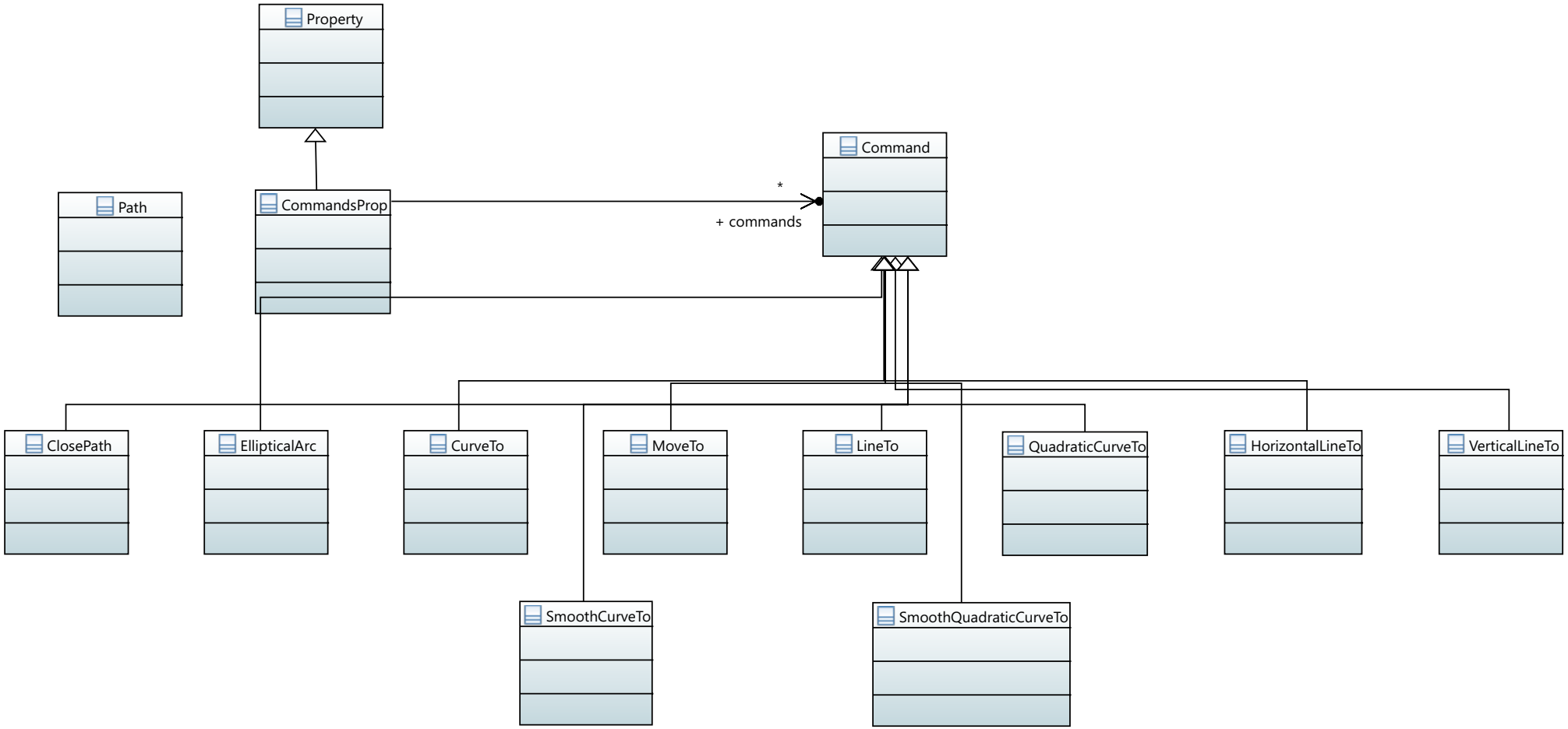
Interactive Primitives



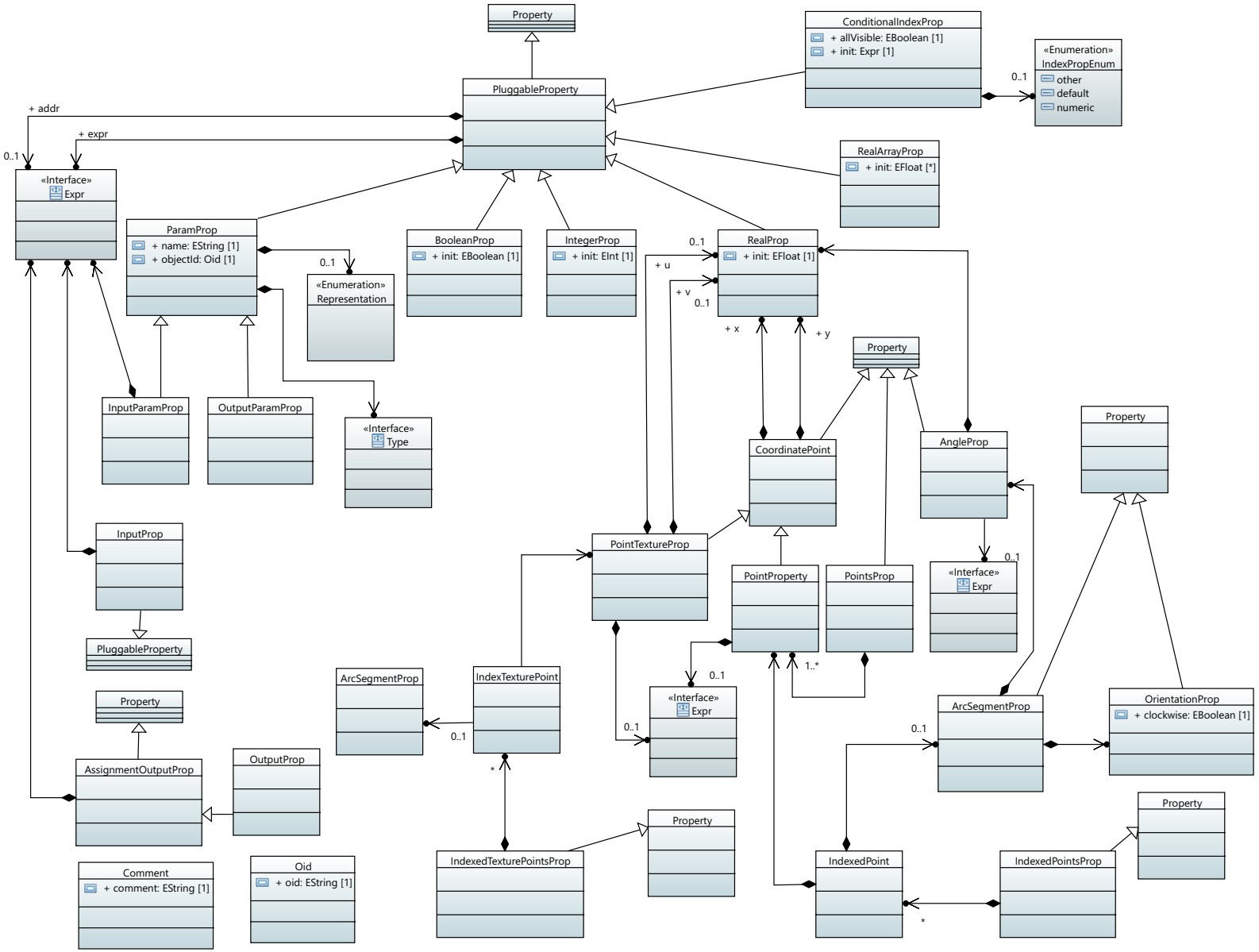
Reference Primitives



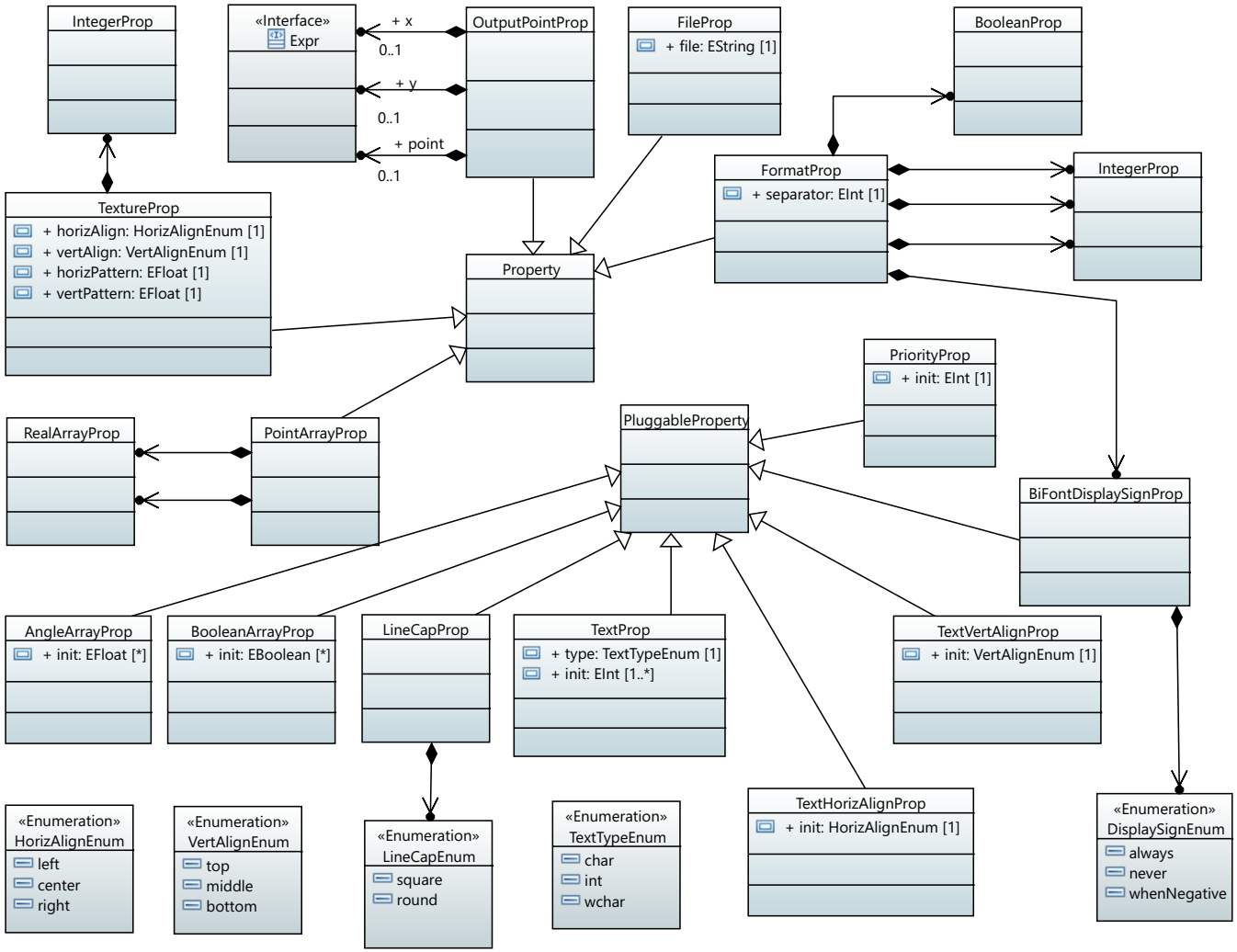
Path Primitives



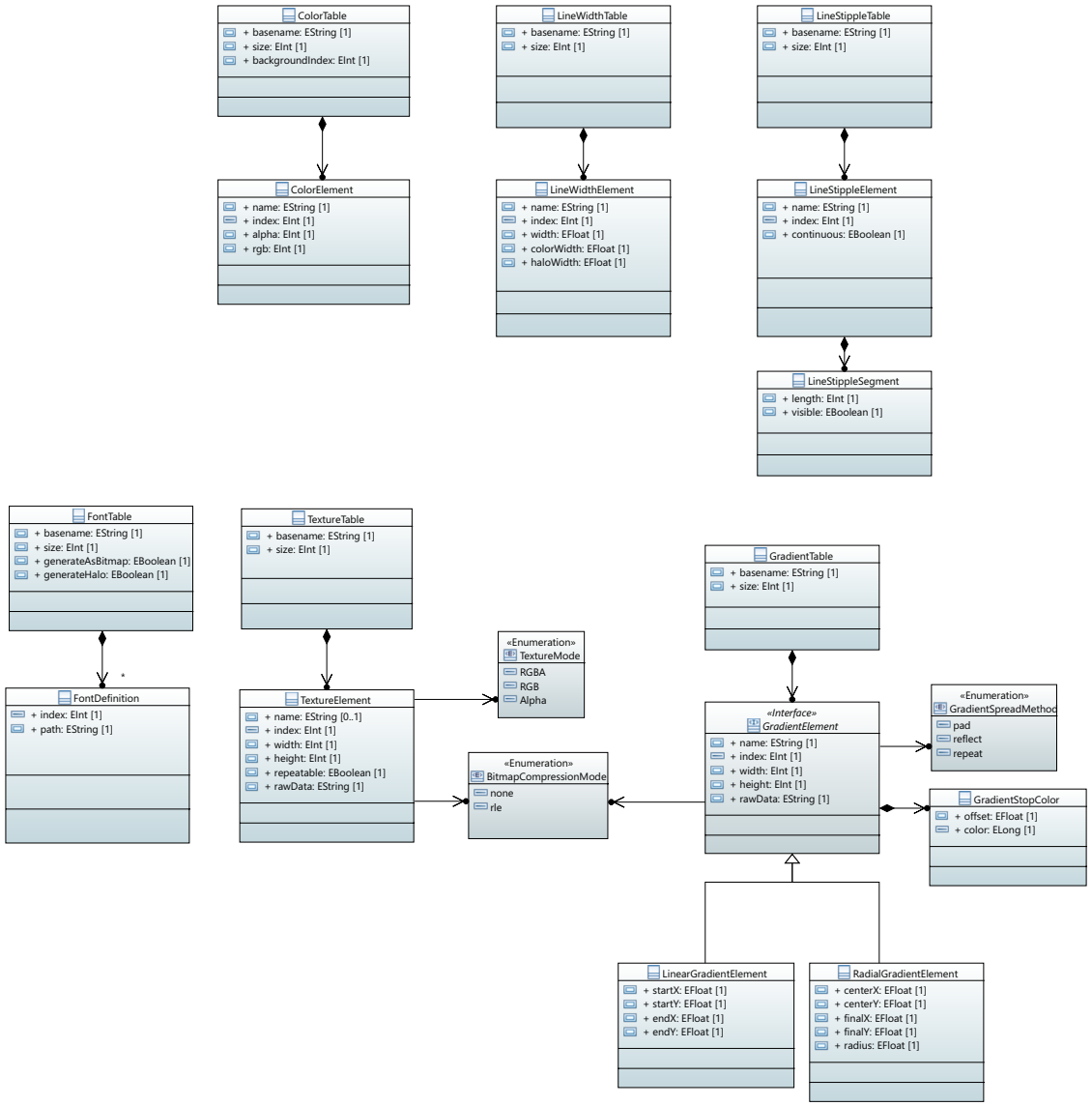
Properties - 1



Properties - 2



Resource Tables

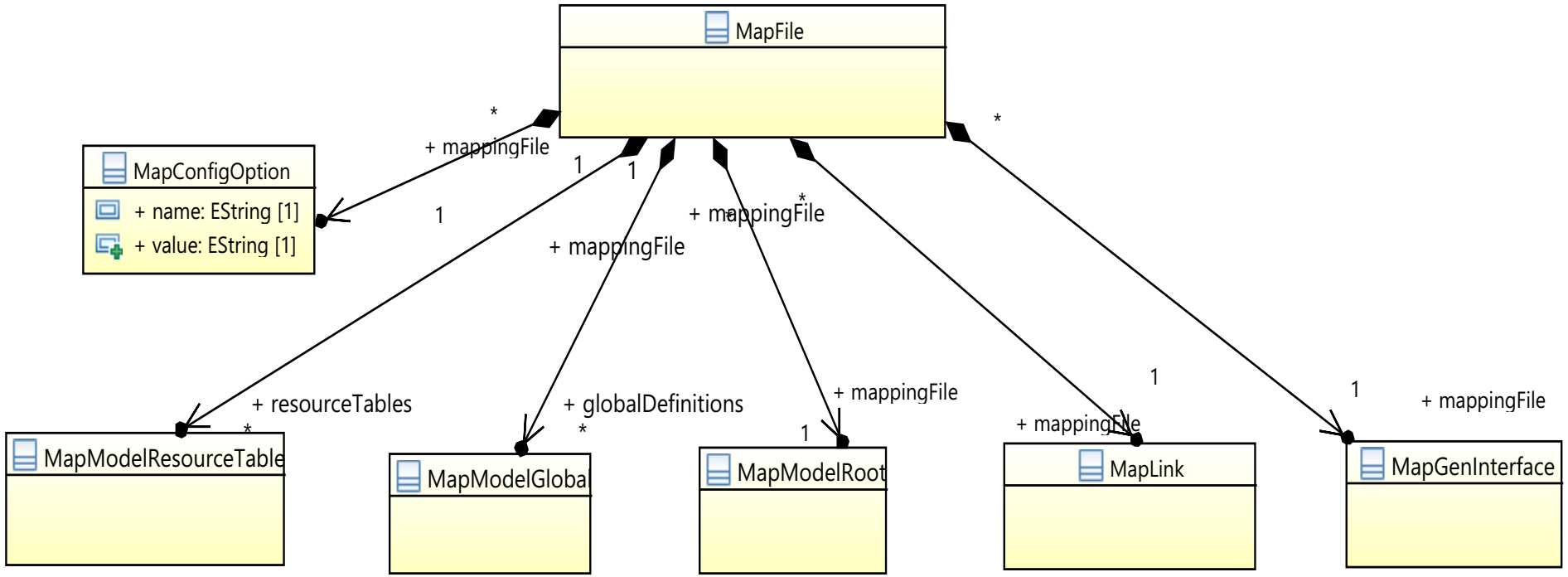


21 / SCADE Display Mapping Files Metamodels

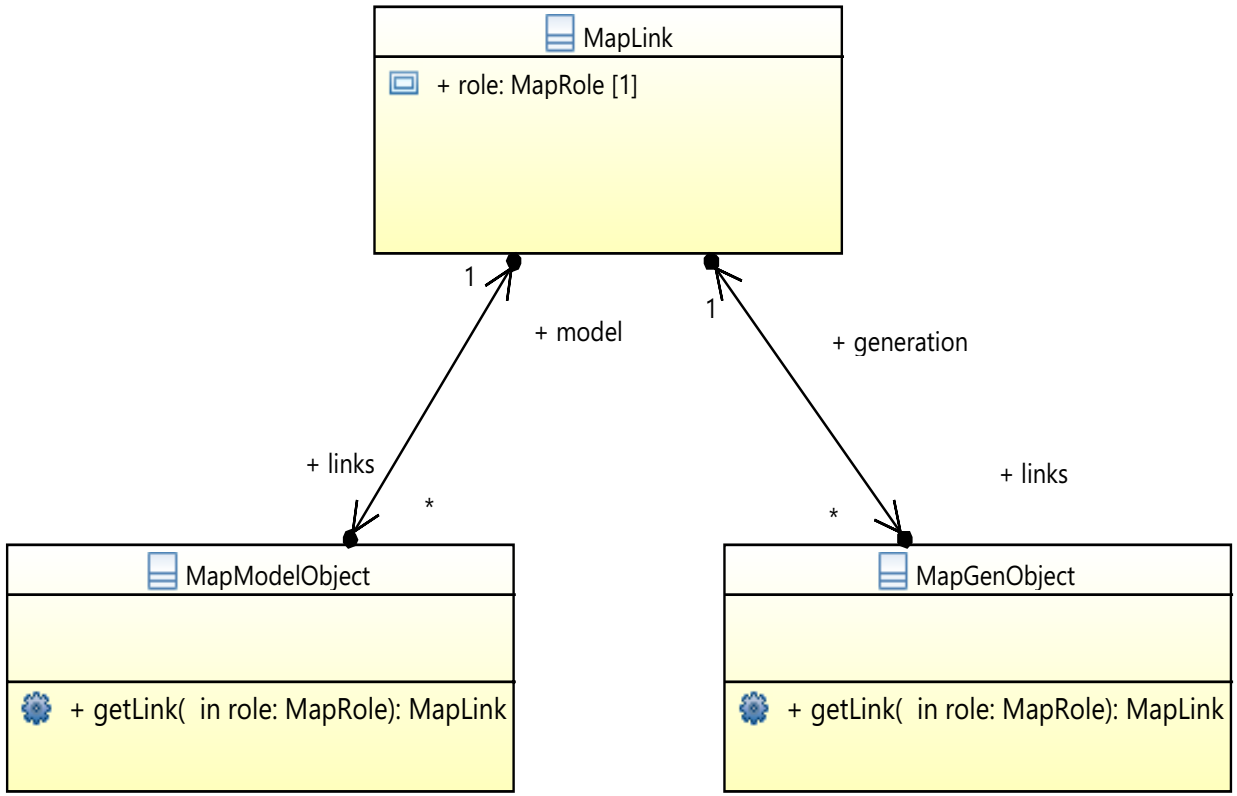
These metamodels present the data structure that give access to generated code through SCADE Display KCG Mapping Files using Python API or Java API:

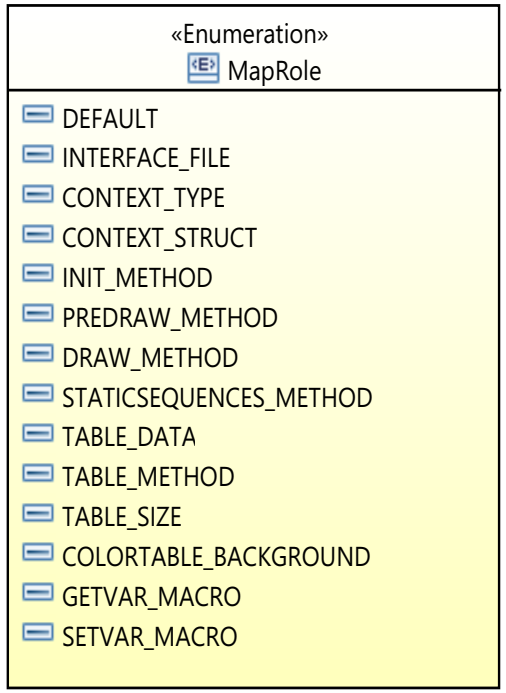
- "Mapping File Entry Points"
- "Mapping File Links (between Generate and Model)"
- "Mapping Model"
- "Mapping File Generate Model"

Mapping File Entry Points

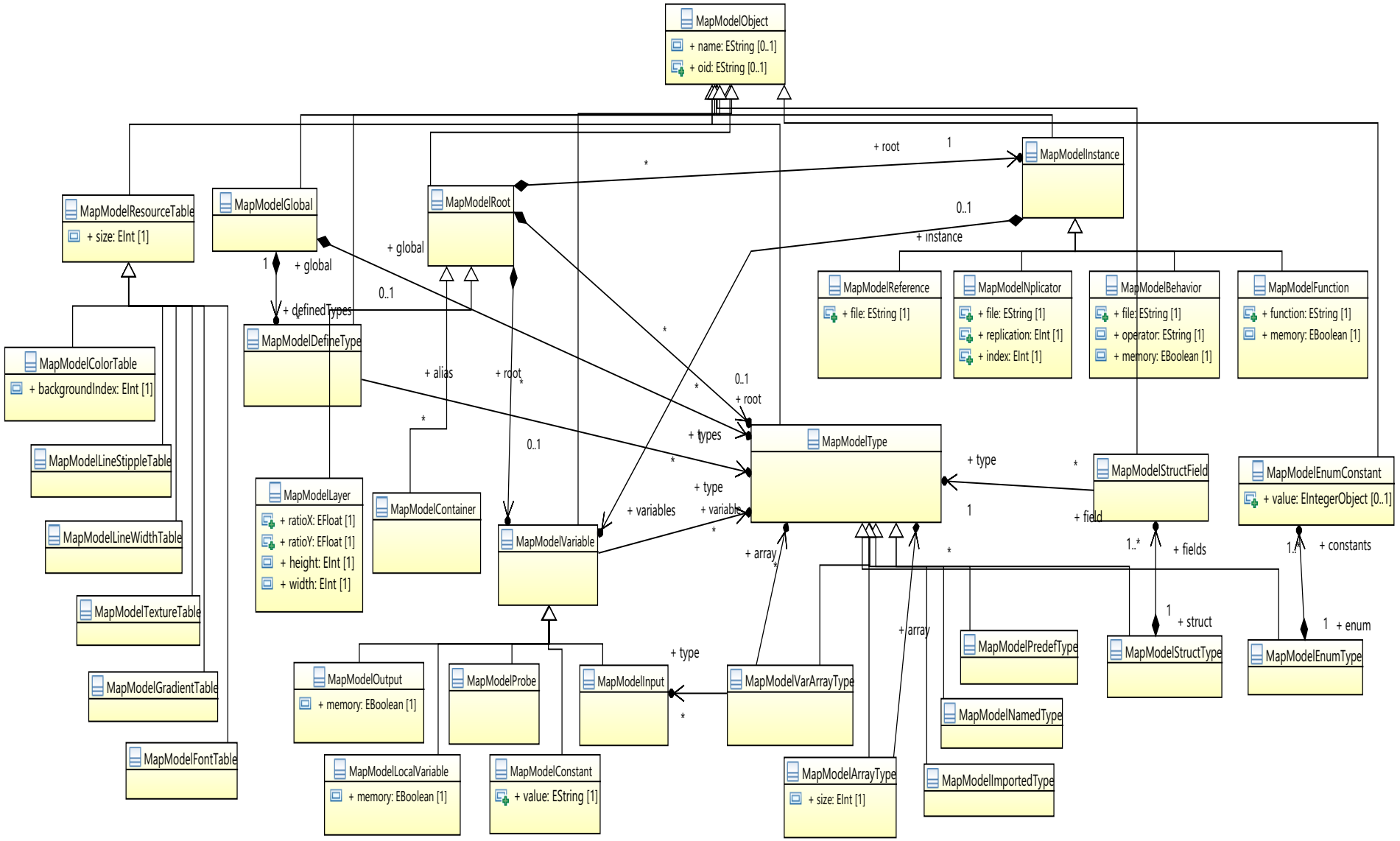


Mapping File Links (between Generate and Model)

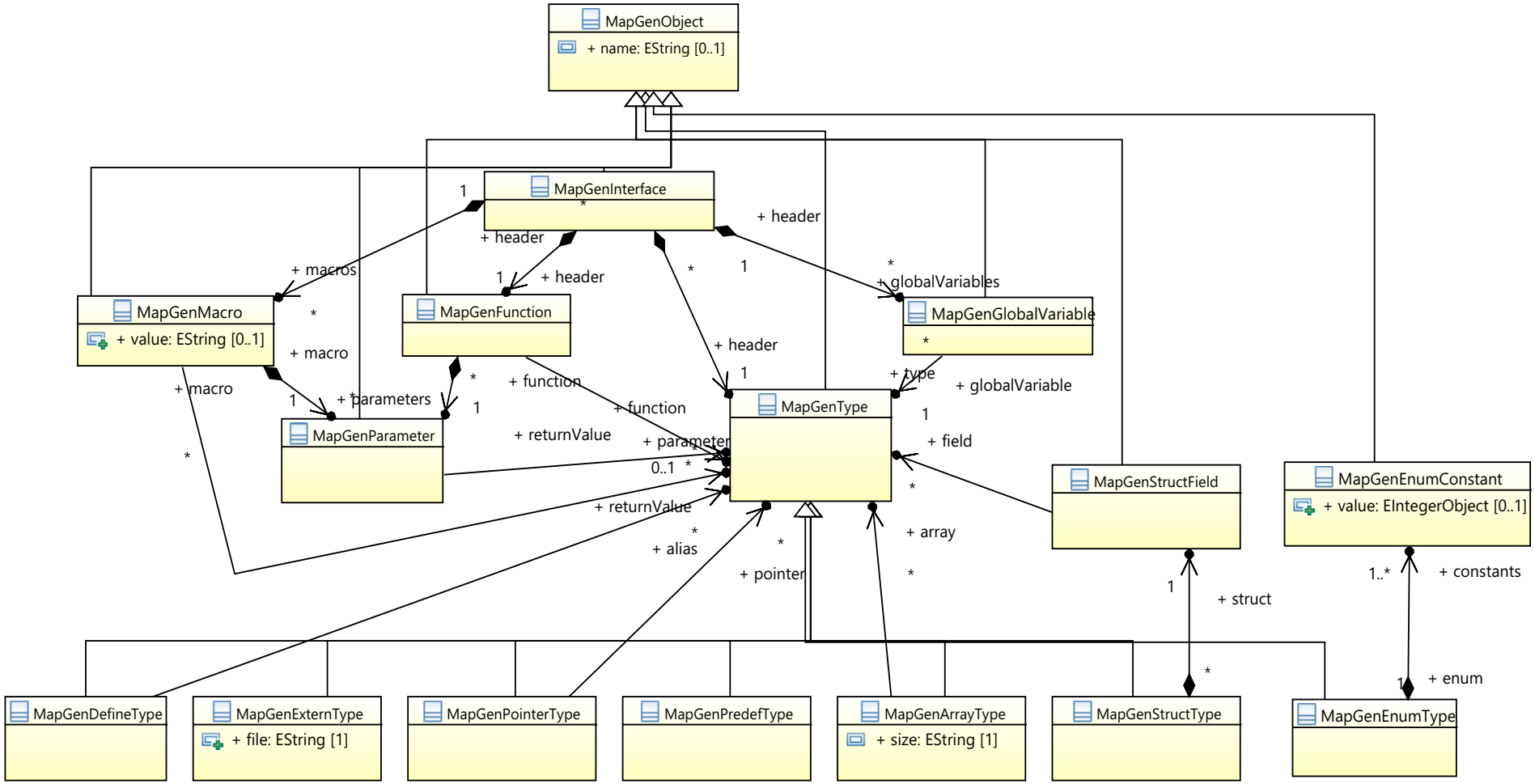




Mapping Model



Mapping File Generate Model



Reference Card

Part 6

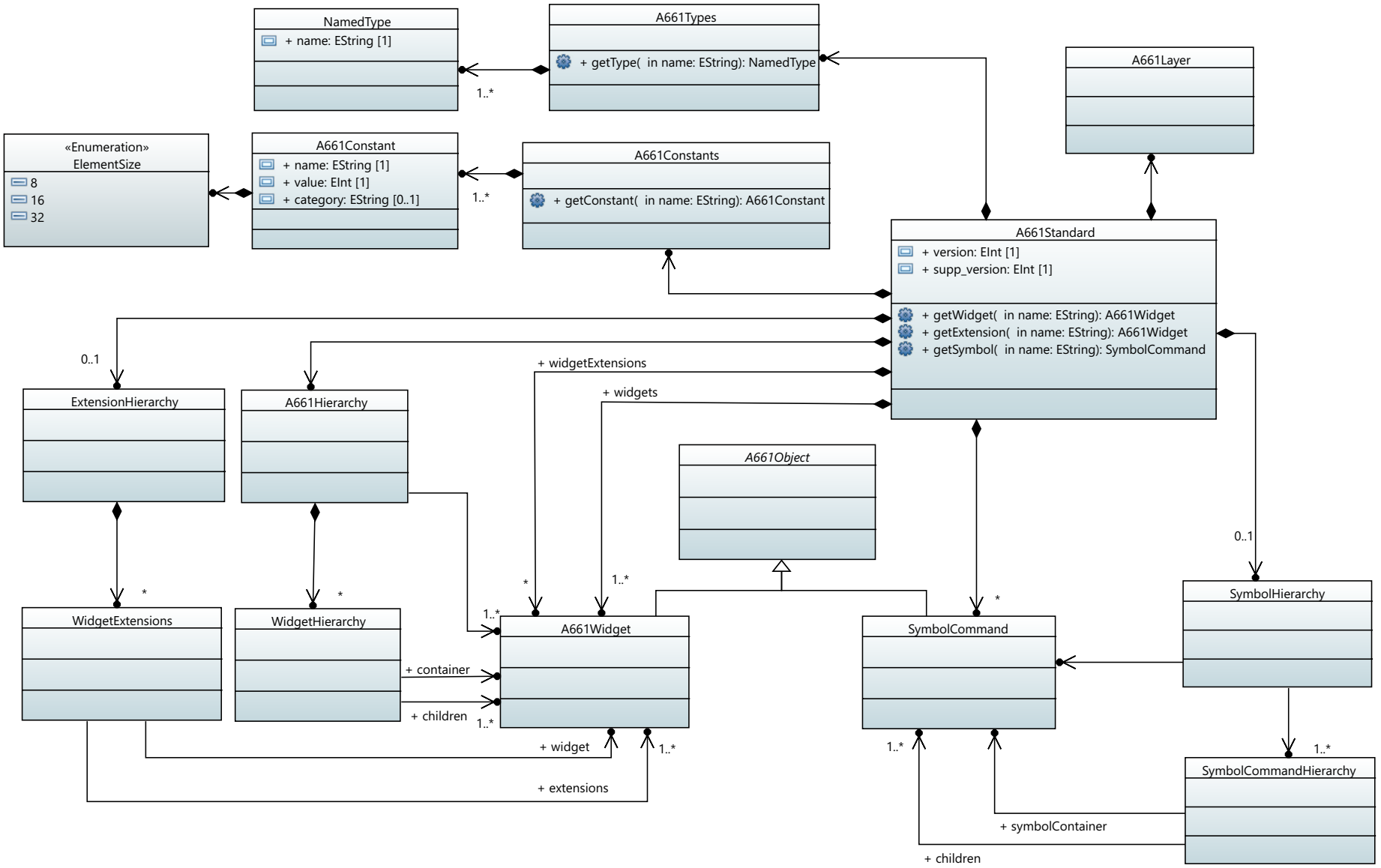
SCADE UA Page Creator Metamodels

22 / SCADE UA Page Creator Metamodels

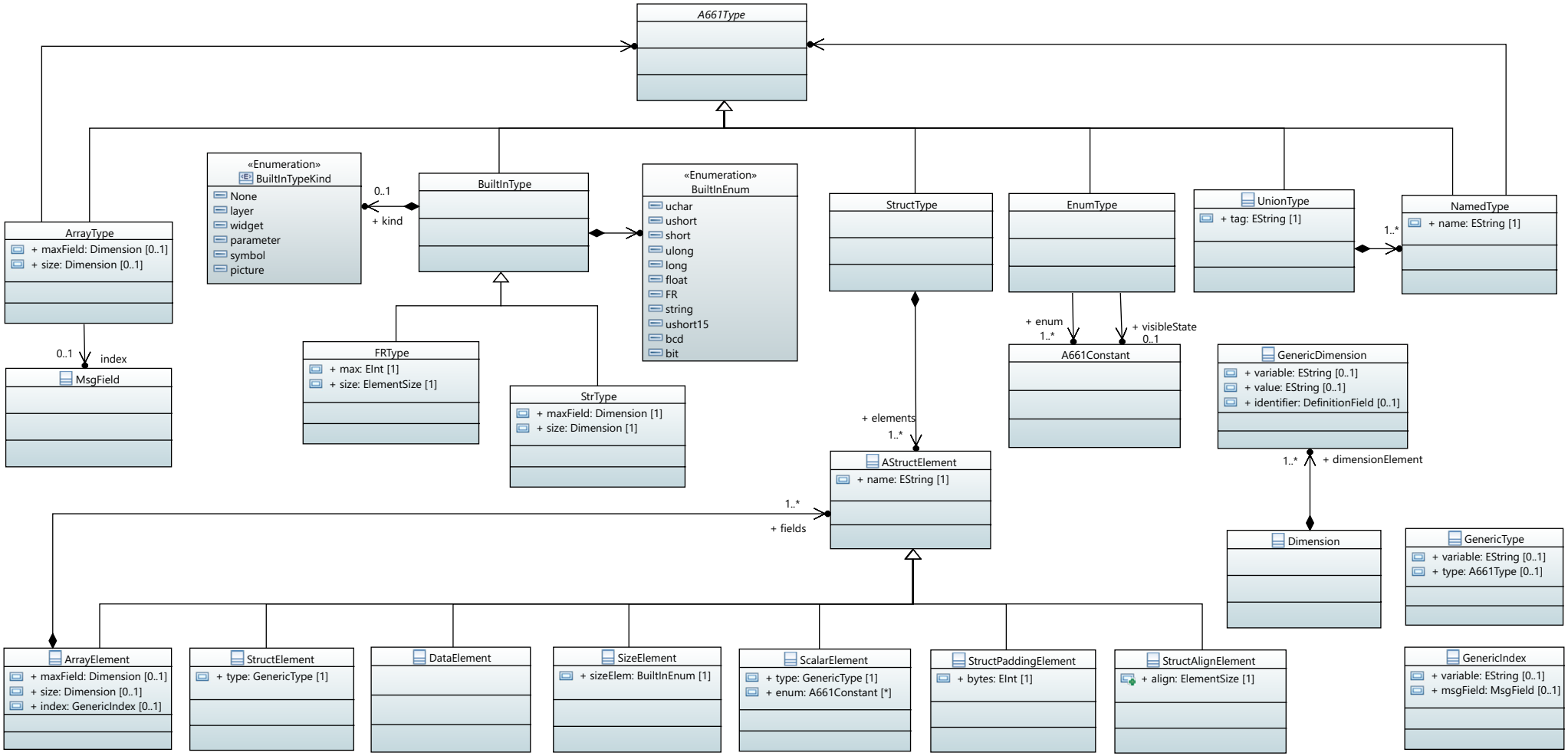
These metamodels present the data structure that give access to SCADE UA Page Creator models using Python API or Java API:

- "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"

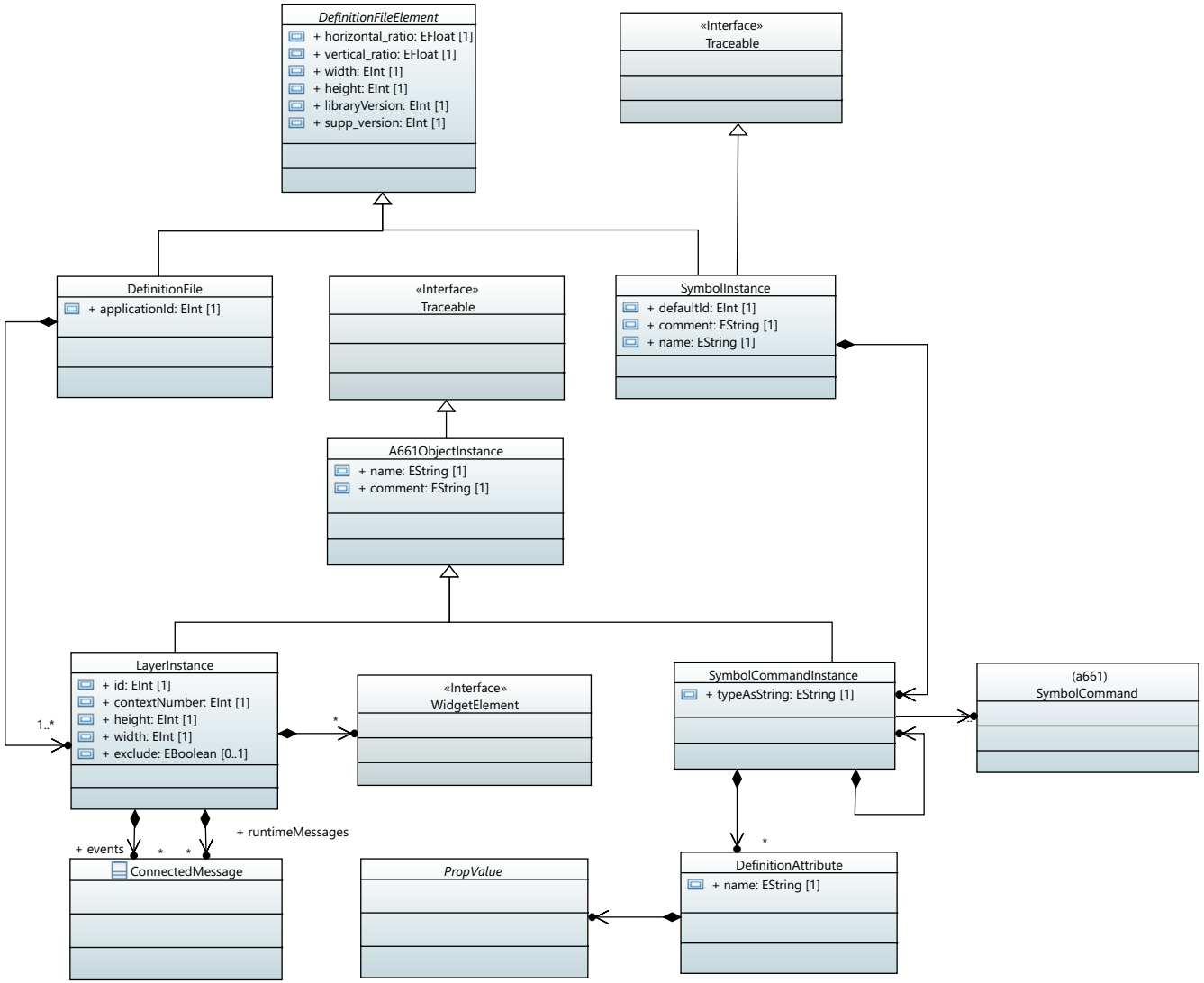
ARINC 661 Standard Model



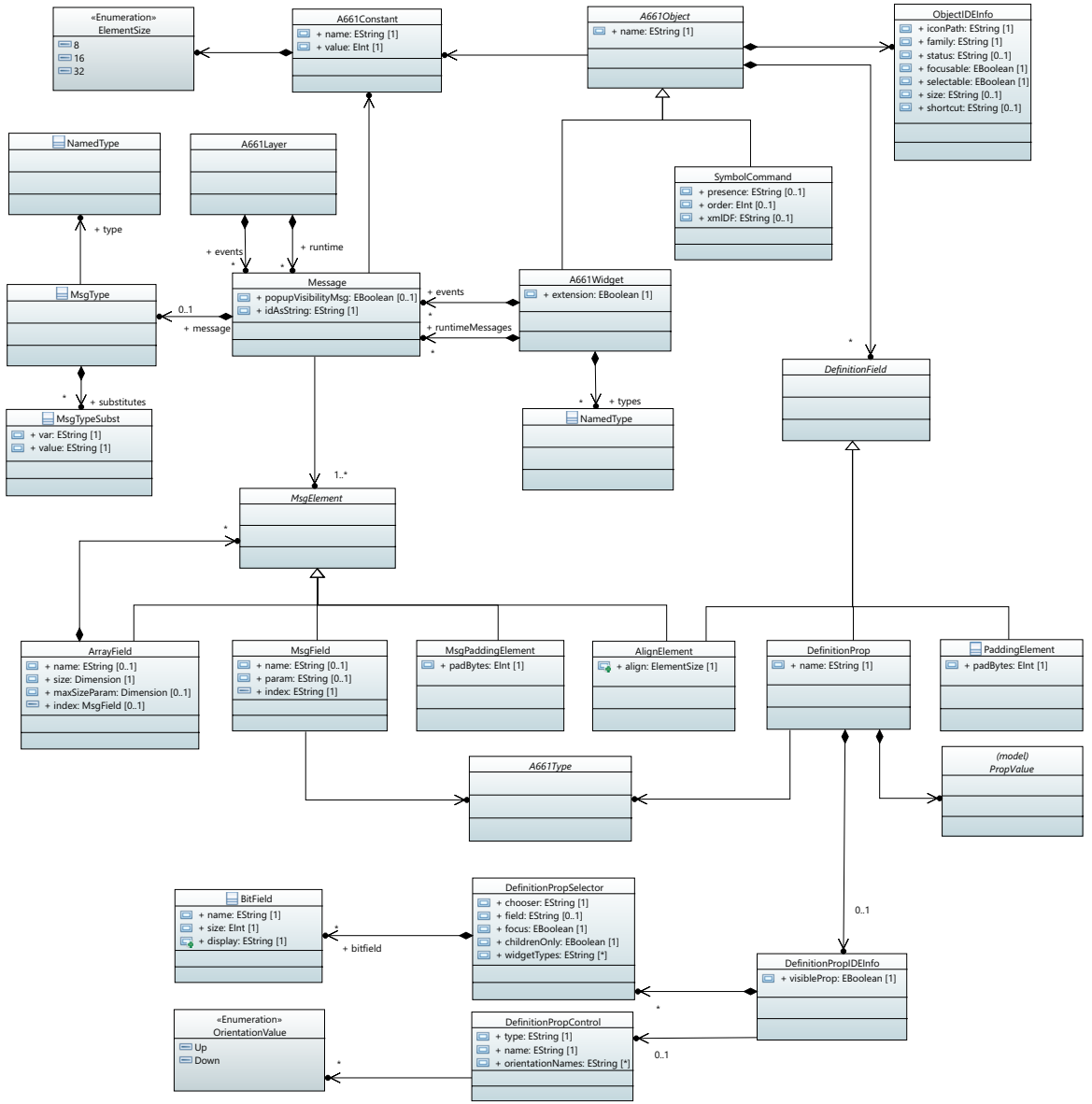
A661 Types


A661Type
«Enumeration»
BuiltInTypeKind
None
layer
widget
parameter
symbol
picture
0..1
+ kind
BuiltInType
«Enumeration»
BuiltInEnum
uchar
ushort
short
ulong
long
float
FR
string
ushort15
bcd
bit
StructType
EnumType
UnionType
+ tag: EString [1]
NamedType
+ name: EString [1]
1..*
ArrayType
+ maxField: Dimension [0..1]
+ size: Dimension [0..1]
0..1
index
MsgField
FRType
+ max: EInt [1]
+ size: ElementSize [1]
StrType
+ maxField: Dimension [1]
+ size: Dimension [1]
+ enum
1..*
+ visibleState
0..1
A661Constant
GenericDimension
+ variable: EString [0..1]
+ value: EString [0..1]
+ identifier: DefinitionField [0..1]
+ elements
1..*
AStructElement
+ name: EString [1]
1..*
+ dimensionElement
1..*
+ fields
Dimension
GenericType
+ variable: EString [0..1]
+ type: A661Type [0..1]
ArrayElement
+ maxField: Dimension [0..1]
+ size: Dimension [0..1]
+ index: GenericIndex [0..1]
StructElement
+ type: GenericType [1]
DataElement
SizeElement
+ sizeElem: BuiltInEnum [1]
ScalarElement
+ type: GenericType [1]
+ enum: A661Constant [*]
StructPaddingElement
+ bytes: EInt [1]
StructAlignElement
+ align: ElementSize [1]
GenericIndex
+ variable: EString [0..1]
+ msgField: MsgField [0..1]

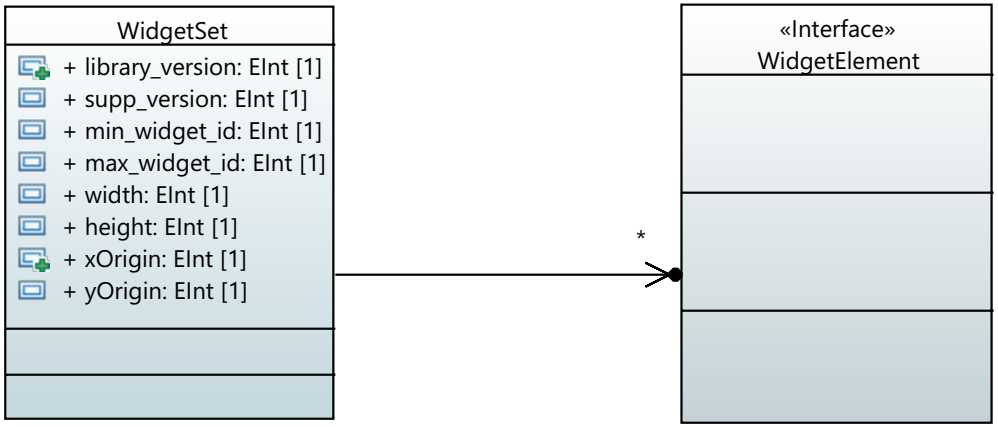
A661 Model


DefinitionFileElement
+ horizontal_ratio: EFloat [1]
+ vertical_ratio: EFloat [1]
+ width: EInt [1]
+ height: EInt [1]
+ libraryVersion: EInt [1]
+ supp_version: EInt [1]
«Interface»
Traceable
DefinitionFile
+ applicationId: EInt [1]
«Interface»
Traceable
SymbolInstance
+ defaultId: EInt [1]
+ comment: EString [1]
+ name: EString [1]
A661ObjectInstance
+ name: EString [1]
+ comment: EString [1]
LayerInstance
+ id: EInt [1]
+ contextNumber: EInt [1]
+ height: EInt [1]
+ width: EInt [1]
+ exclude: EBoolean [0..1]
1..*
*
«Interface»
WidgetElement
SymbolCommandInstance
+ typeAsString: EString [1]
(a661)
SymbolCommand
+ events
+ runtimeMessages
ConnectedMessage
PropValue
DefinitionAttribute
+ name: EString [1]

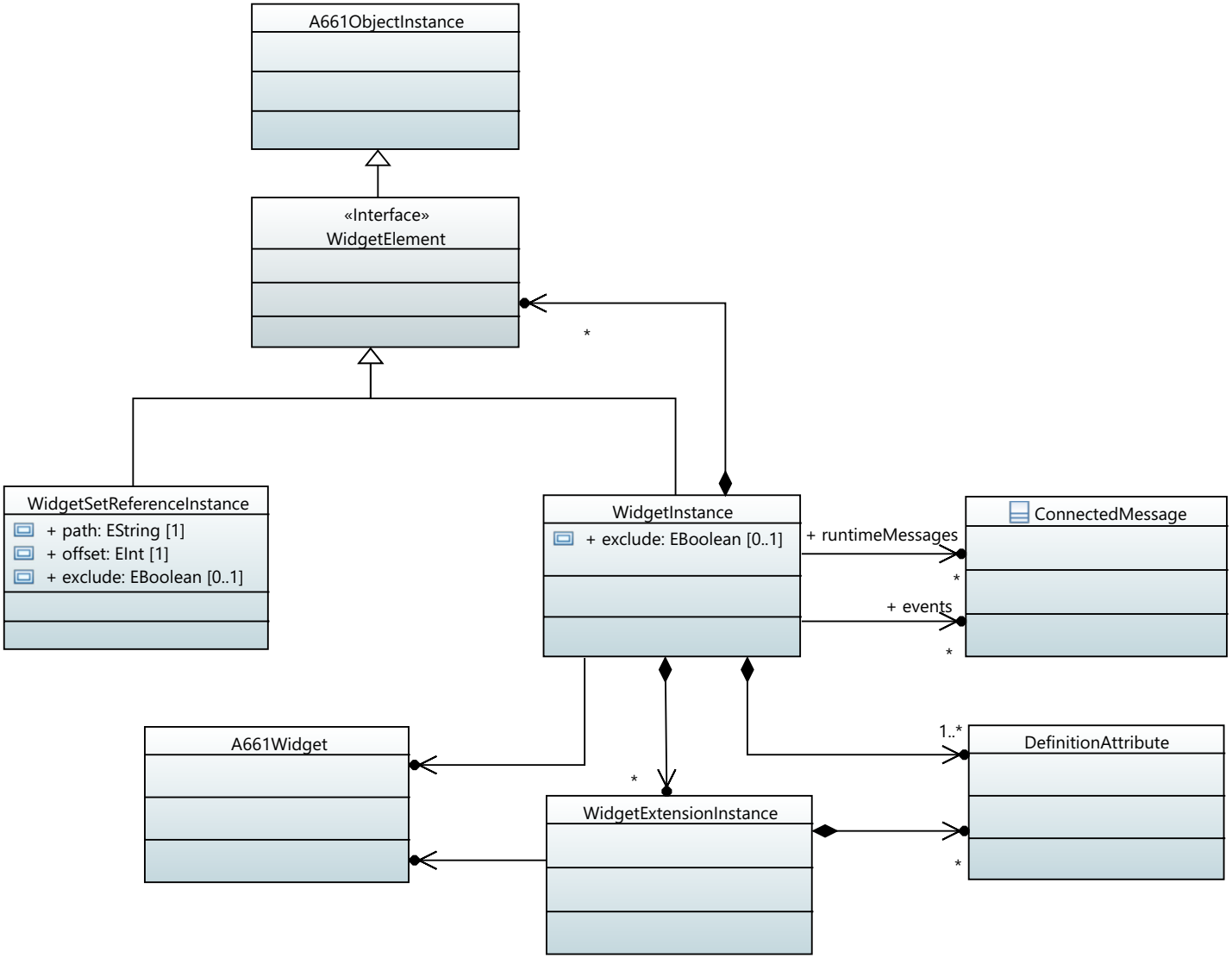
Widgets


«Enumeration»
ElementSize
8
16
32
A661Constant
+ name: EString [1]
+ value: EInt [1]
A661Object
+ name: EString [1]
ObjectIDEInfo
+ iconPath: EString [1]
+ family: EString [1]
+ status: EString [0..1]
+ focusable: EBoolean [1]
+ selectable: EBoolean [1]
+ size: EString [0..1]
+ shortcut: EString [0..1]
NamedType
A661Layer
SymbolCommand
+ presence: EString [0..1]
+ order: EInt [0..1]
+ xmlDF: EString [0..1]
+ type
+ events
+ runtime
MsgType
0..1
+ message
Message
+ popupVisibilityMsg: EBoolean [0..1]
+ idAsString: EString [1]
+ events
+ runtimeMessages
A661Widget
+ extension: EBoolean [1]
DefinitionField
+ substitutes
MsgTypeSubst
+ var: EString [1]
+ value: EString [1]
+ types
NamedType
1..*
MsgElement
ArrayField
+ name: EString [0..1]
+ size: Dimension [1]
+ maxSizeParam: Dimension [0..1]
+ index: MsgField [0..1]
MsgField
+ name: EString [0..1]
+ param: EString [0..1]
+ index: EString [1]
MsgPaddingElement
+ padBytes: EInt [1]
AlignElement
+ align: ElementSize [1]
DefinitionProp
+ name: EString [1]
PaddingElement
+ padBytes: EInt [1]
A661Type
(model)
PropValue
BitField
+ name: EString [1]
+ size: EInt [1]
+ display: EString [1]
+ bitfield
DefinitionPropSelector
+ chooser: EString [1]
+ field: EString [0..1]
+ focus: EBoolean [1]
+ childrenOnly: EBoolean [1]
+ widgetTypes: EString [*]
0..1
DefinitionPropIDEInfo
+ visibleProp: EBoolean [1]
«Enumeration»
OrientationValue
Up
Down
DefinitionPropControl
+ type: EString [1]
+ name: EString [1]
+ orientationNames: EString [*]
0..1

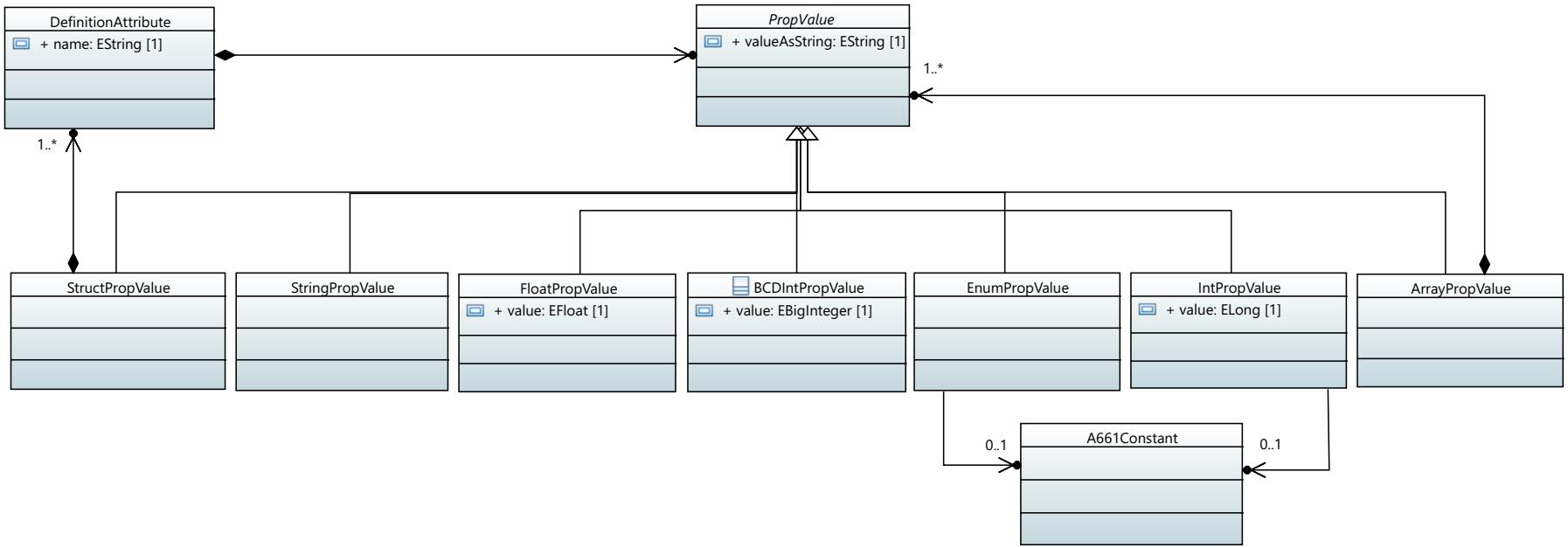
Widget Set


WidgetSet
+ library_version: EInt [1]
+ supp_version: EInt [1]
+ min_widget_id: EInt [1]
+ max_widget_id: EInt [1]
+ width: EInt [1]
+ height: EInt [1]
+ xOrigin: EInt [1]
+ yOrigin: EInt [1]
*
«Interface»
WidgetElement

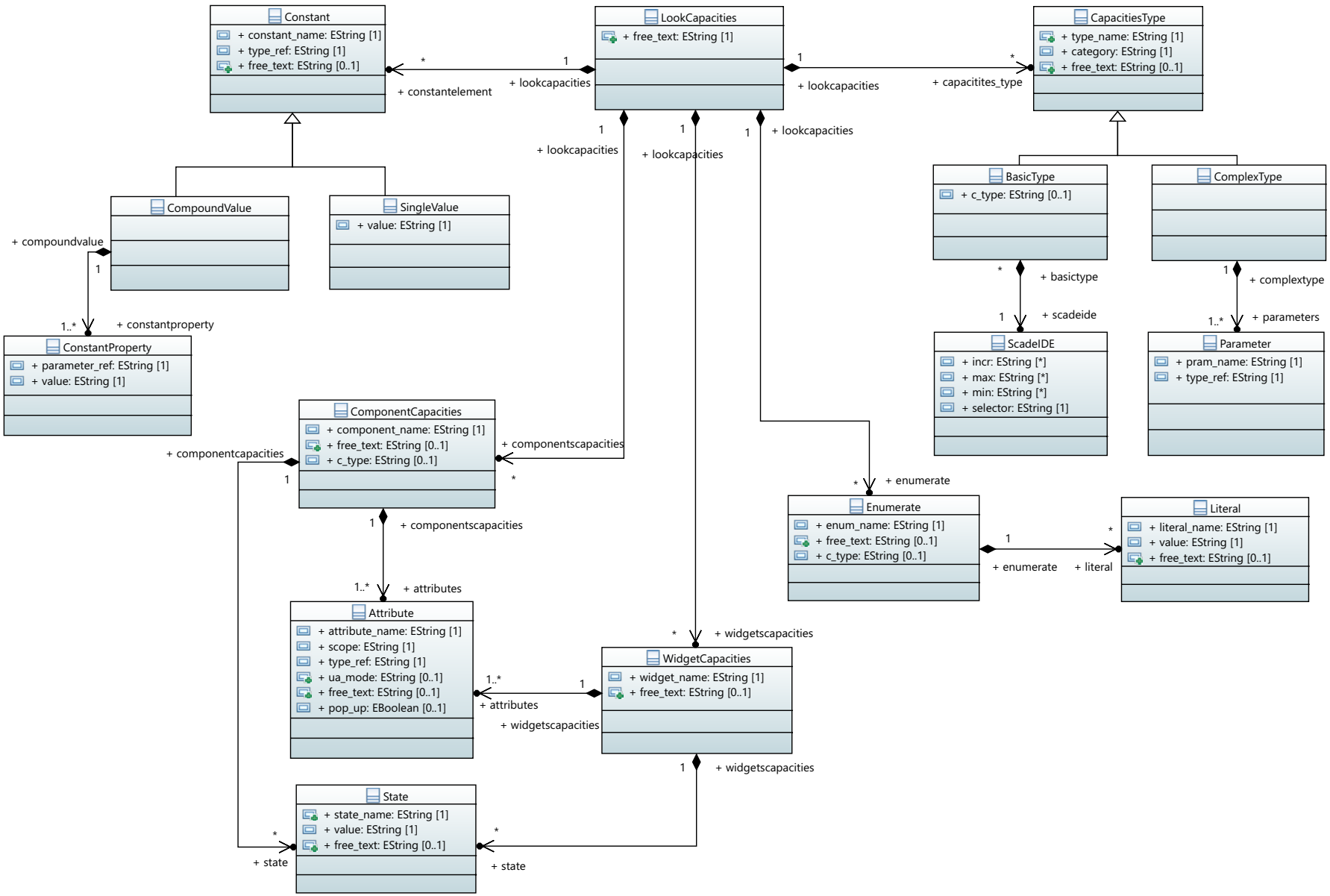
Widget Instance



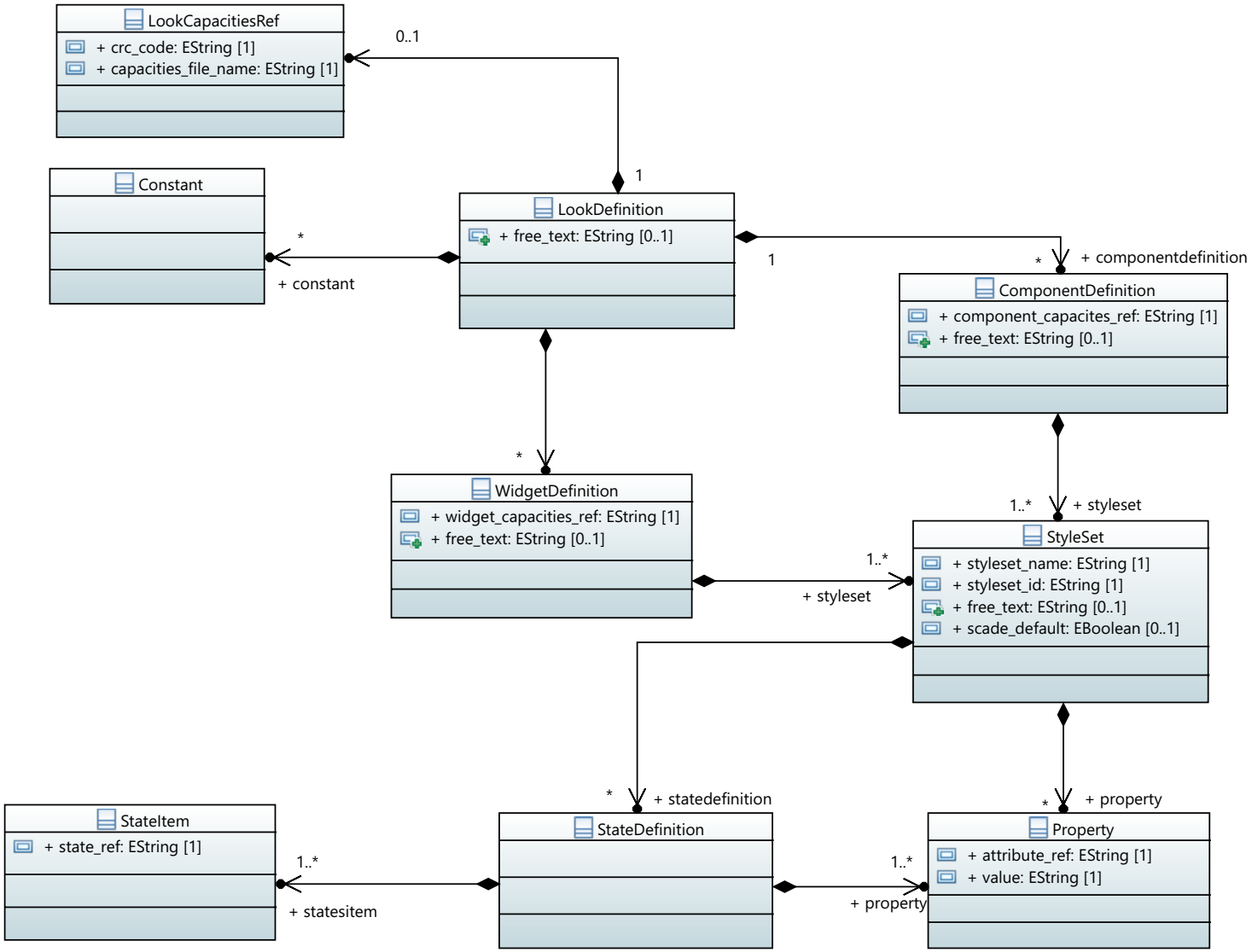
A661 Properties



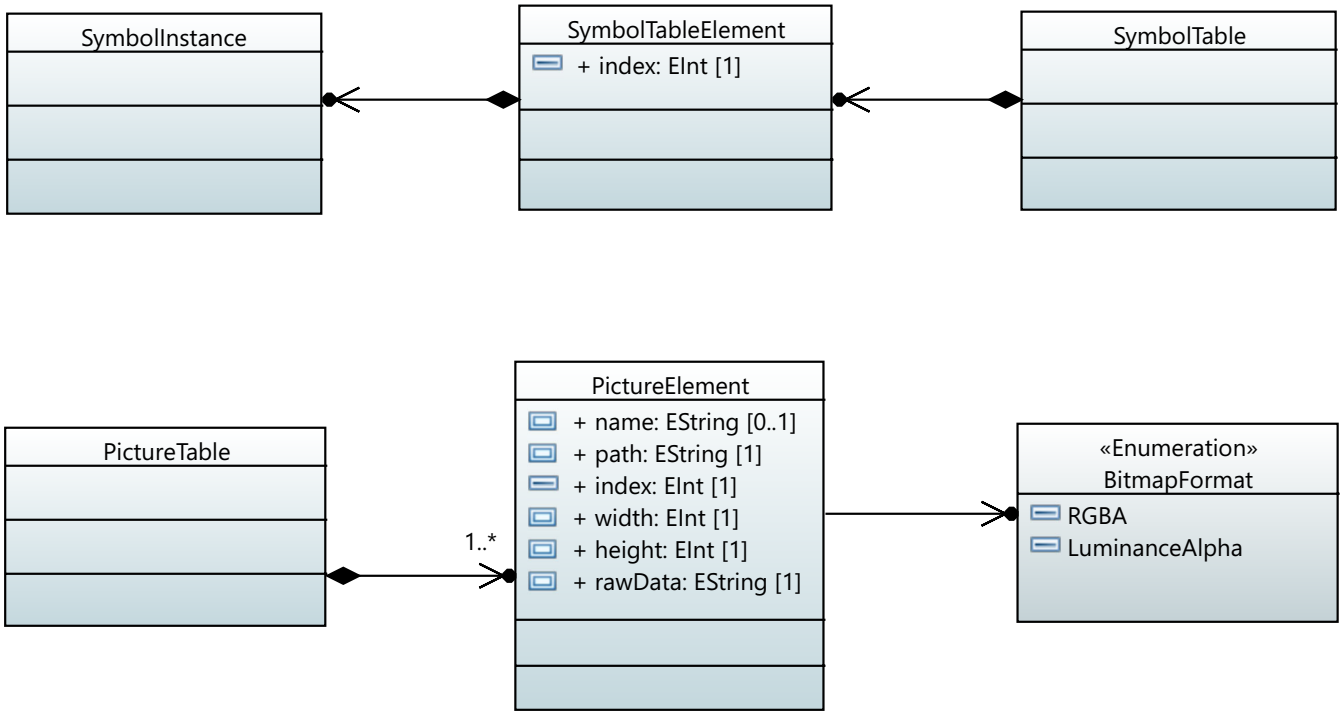
Component and Widget Look Capacities



Component and Widget Look Definitions



Picture and Symbol Tables



Reference Card

Part 7

SCADE Test Metamodels

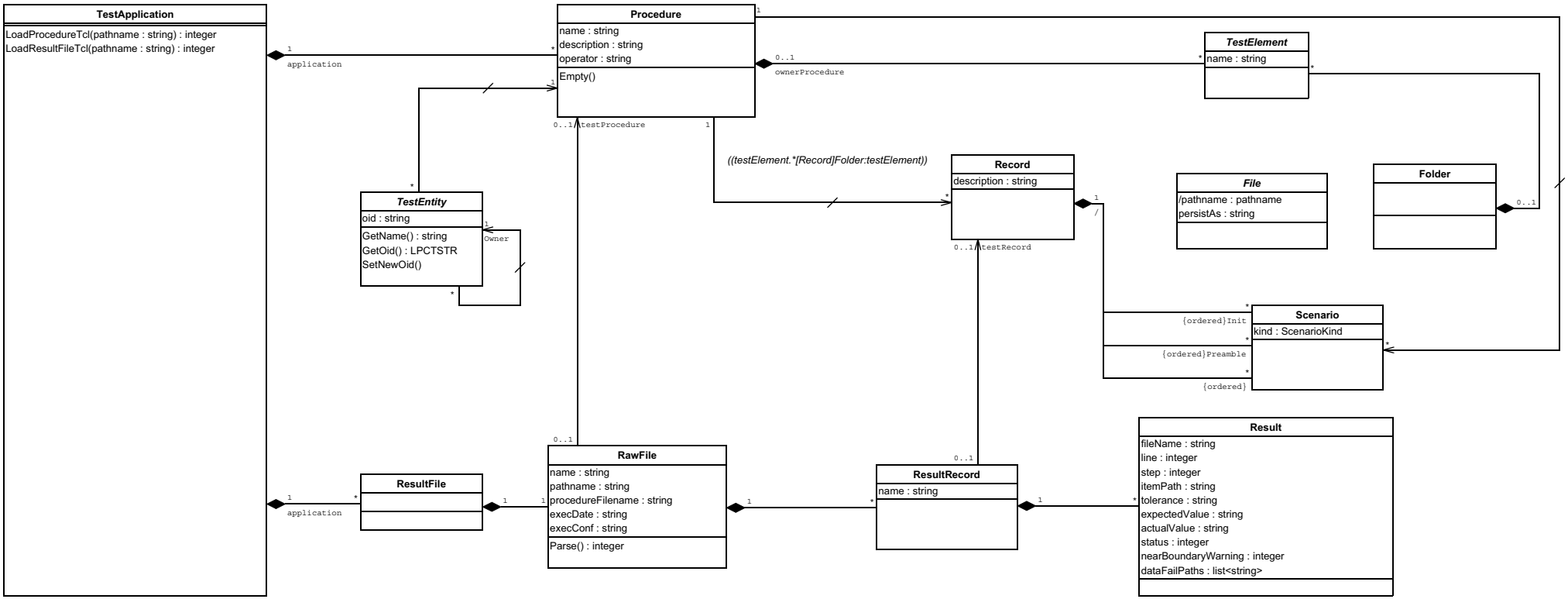
23 /Test Environment for Host Metamodels (Tcl)

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

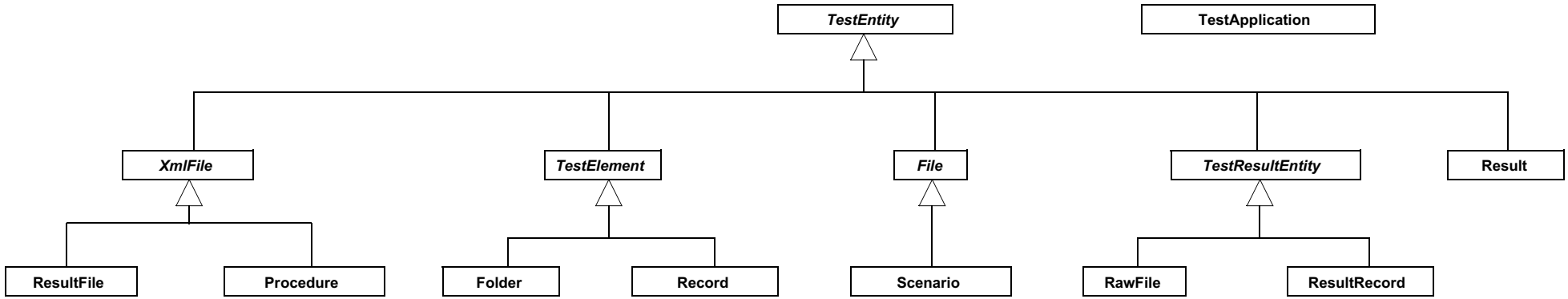
For SCADE Suite model testing:

- "Testing Environment Core (Tcl)"
- "Testing Environment Inheritance (Tcl)"
- "Testing Environment Procedure File (Tcl, Java)"
- "Testing Environment Result File (Tcl)"

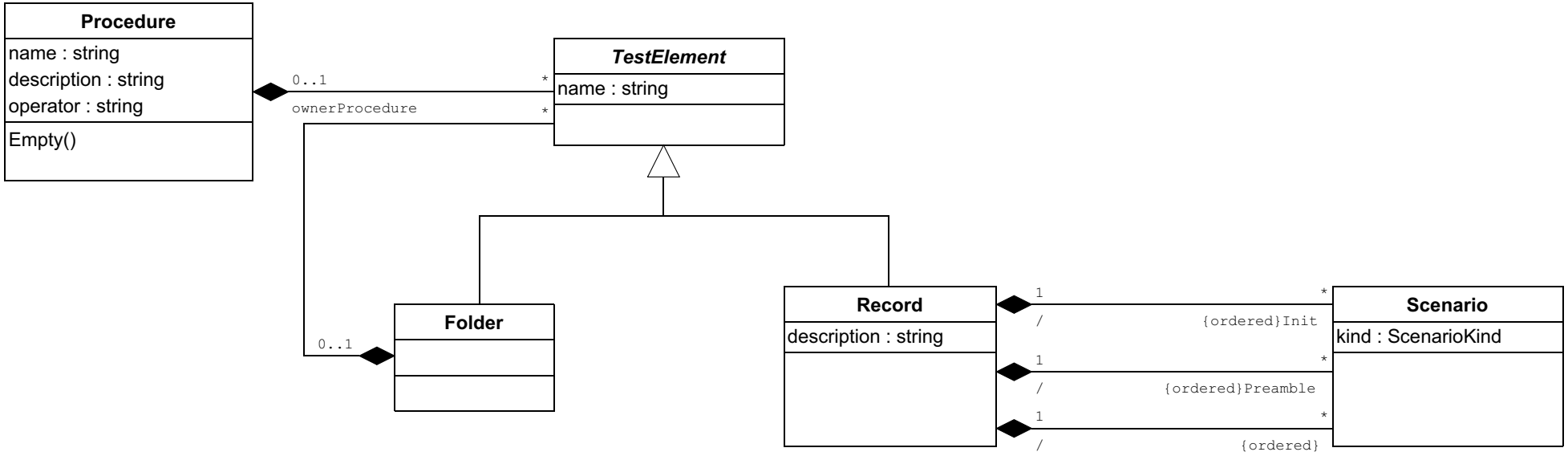
Testing Environment Core (Tcl)



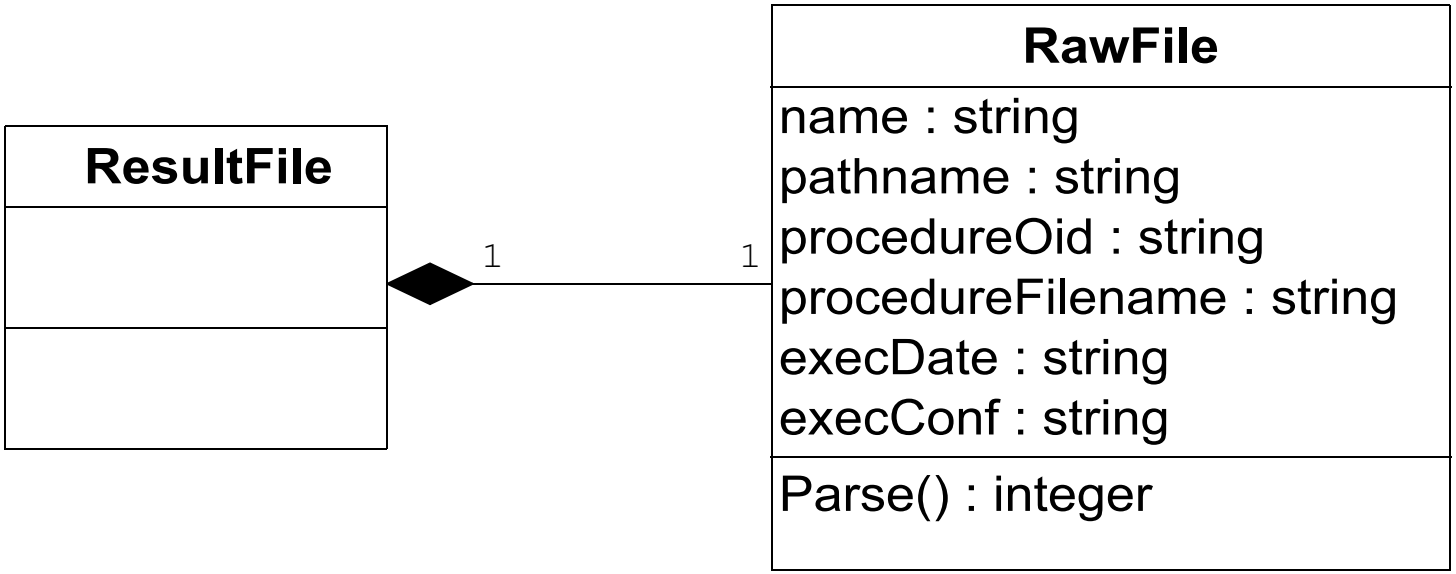
Testing Environment Inheritance (Tcl)



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:

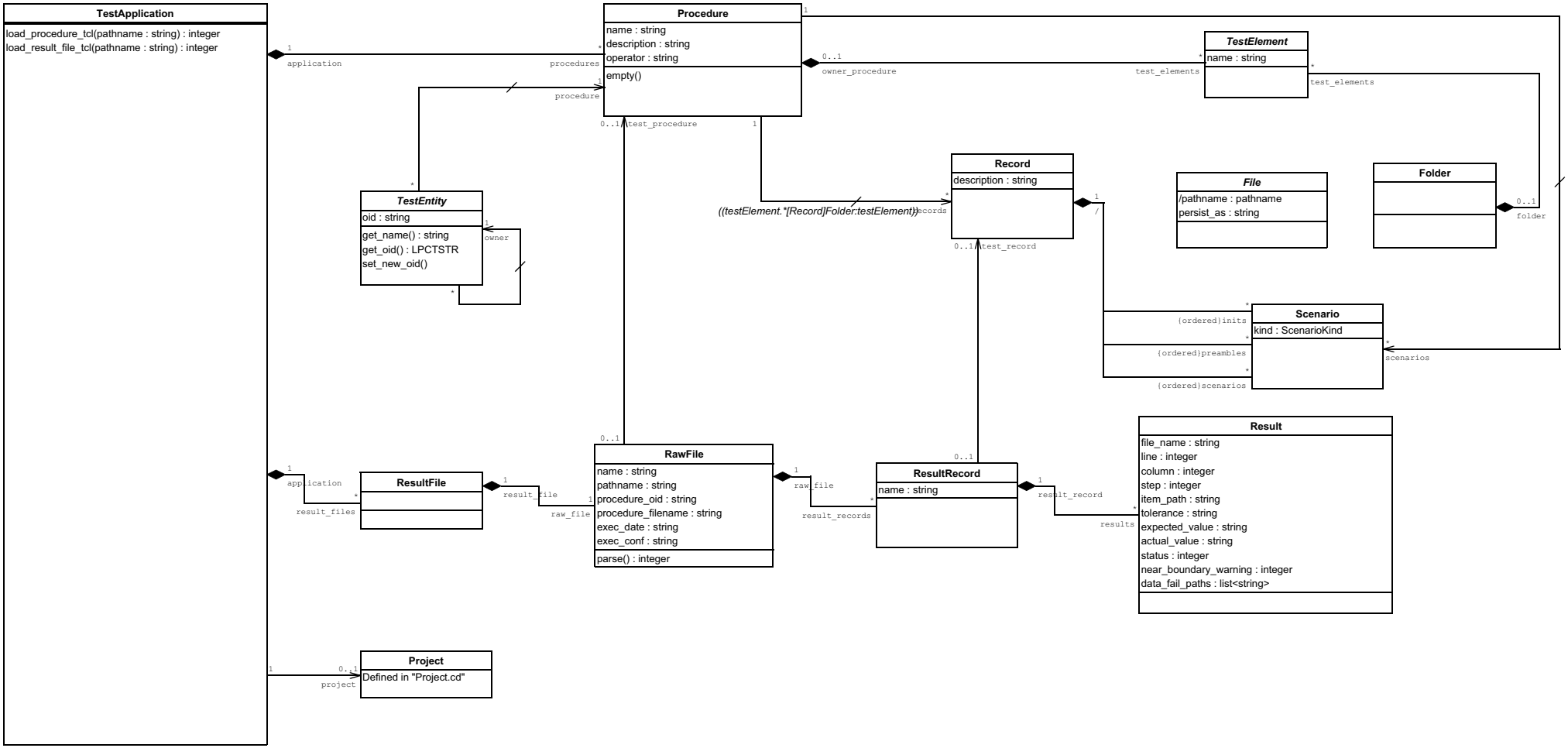
For SCADE Suite model testing:

- "Testing Environment Core (Python)"
- "Testing Environment Inheritance (Python)"
- "Testing Environment Procedure File (Python)"
- "Testing Environment Result File (Python)"

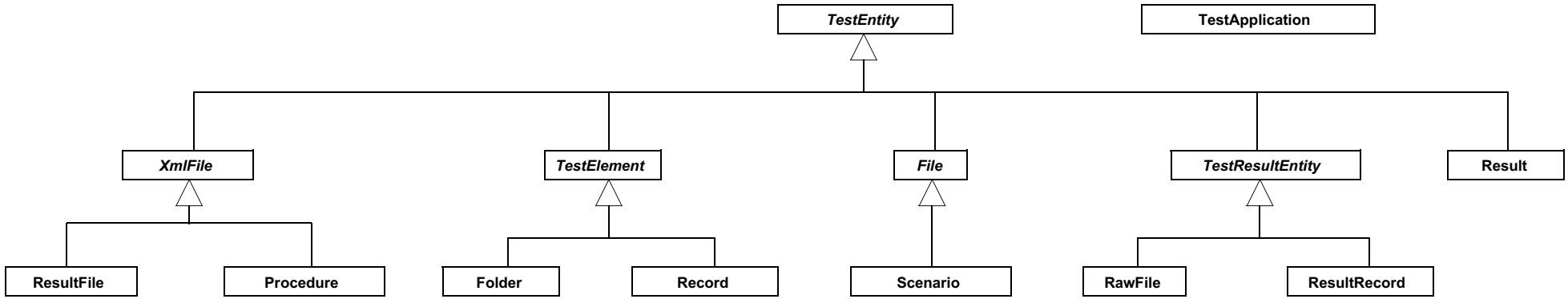
For SCADE Display model testing:

- "Scenario Classes (Python, Java)"

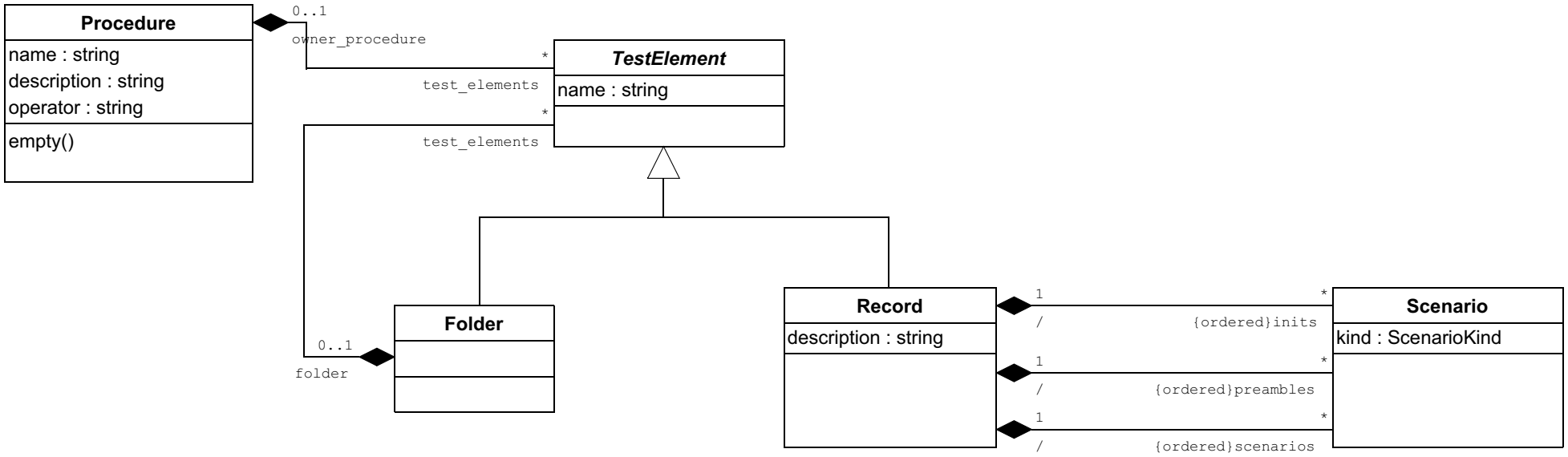
Testing Environment Core (Python)



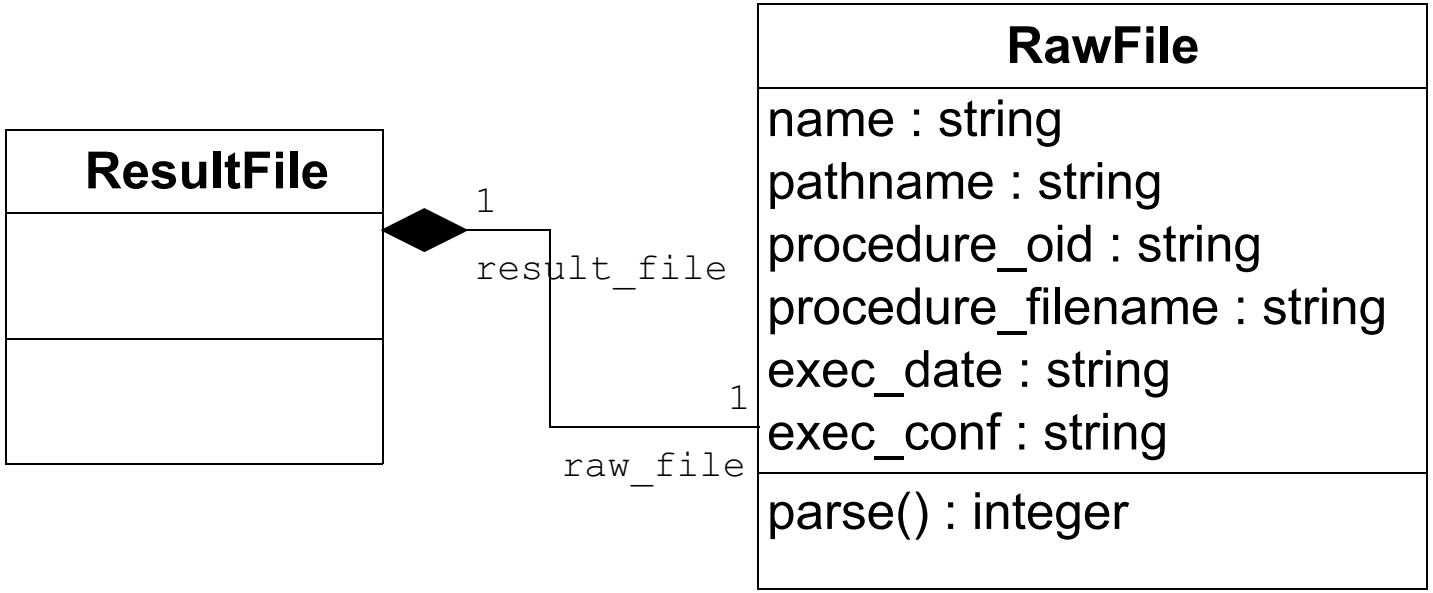
Testing Environment Inheritance (Python)



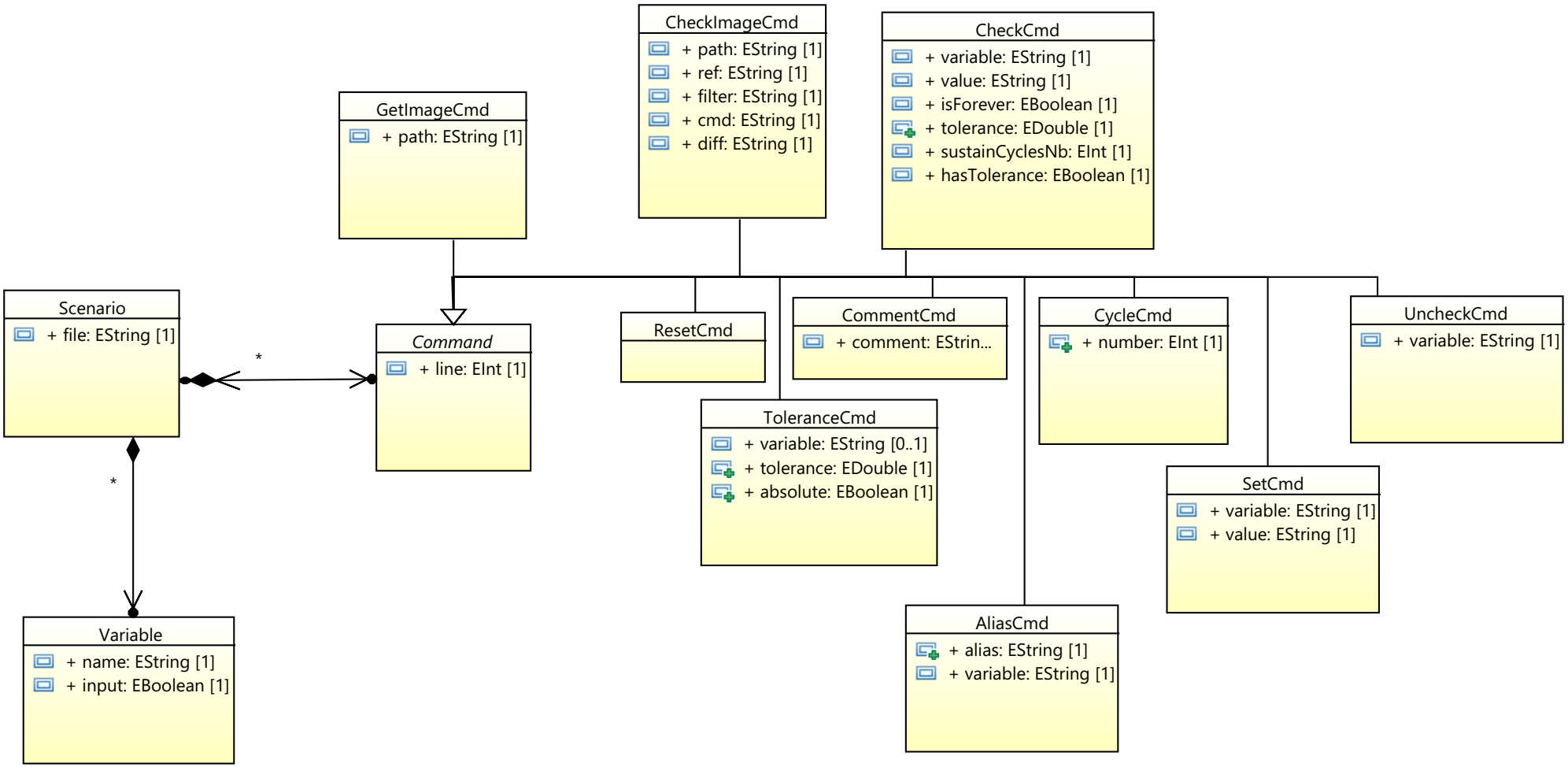
Testing Environment Procedure File (Python)



Testing Environment Result File (Python)



Scenario Classes (Python, Java)



Reference Card

Part 8

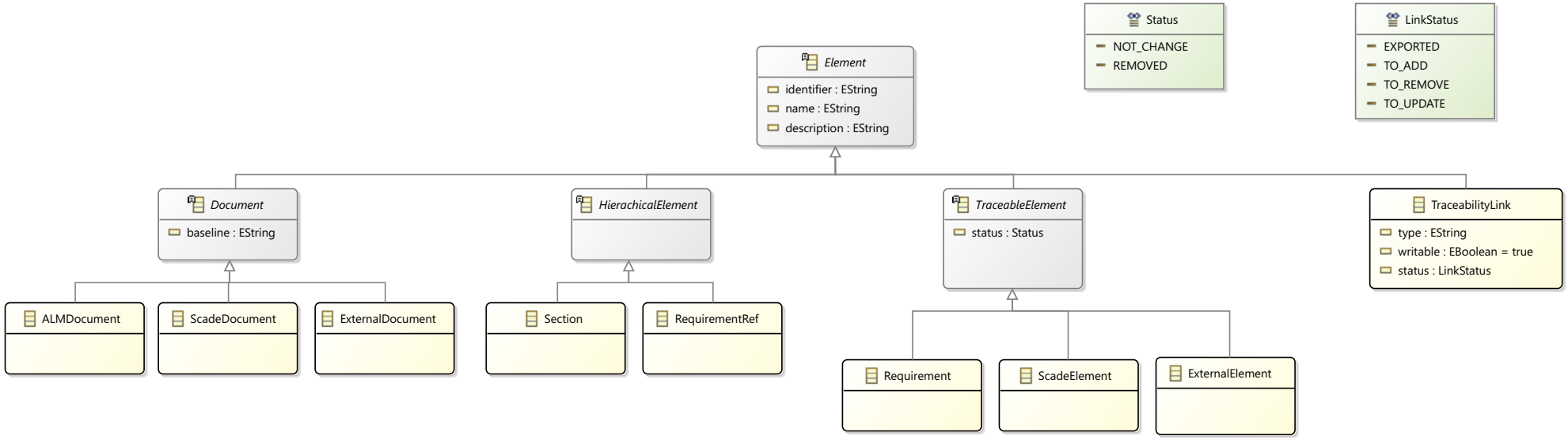
SCADE Traceability Metamodels for Python API

25 / SCADE Traceability Metamodels

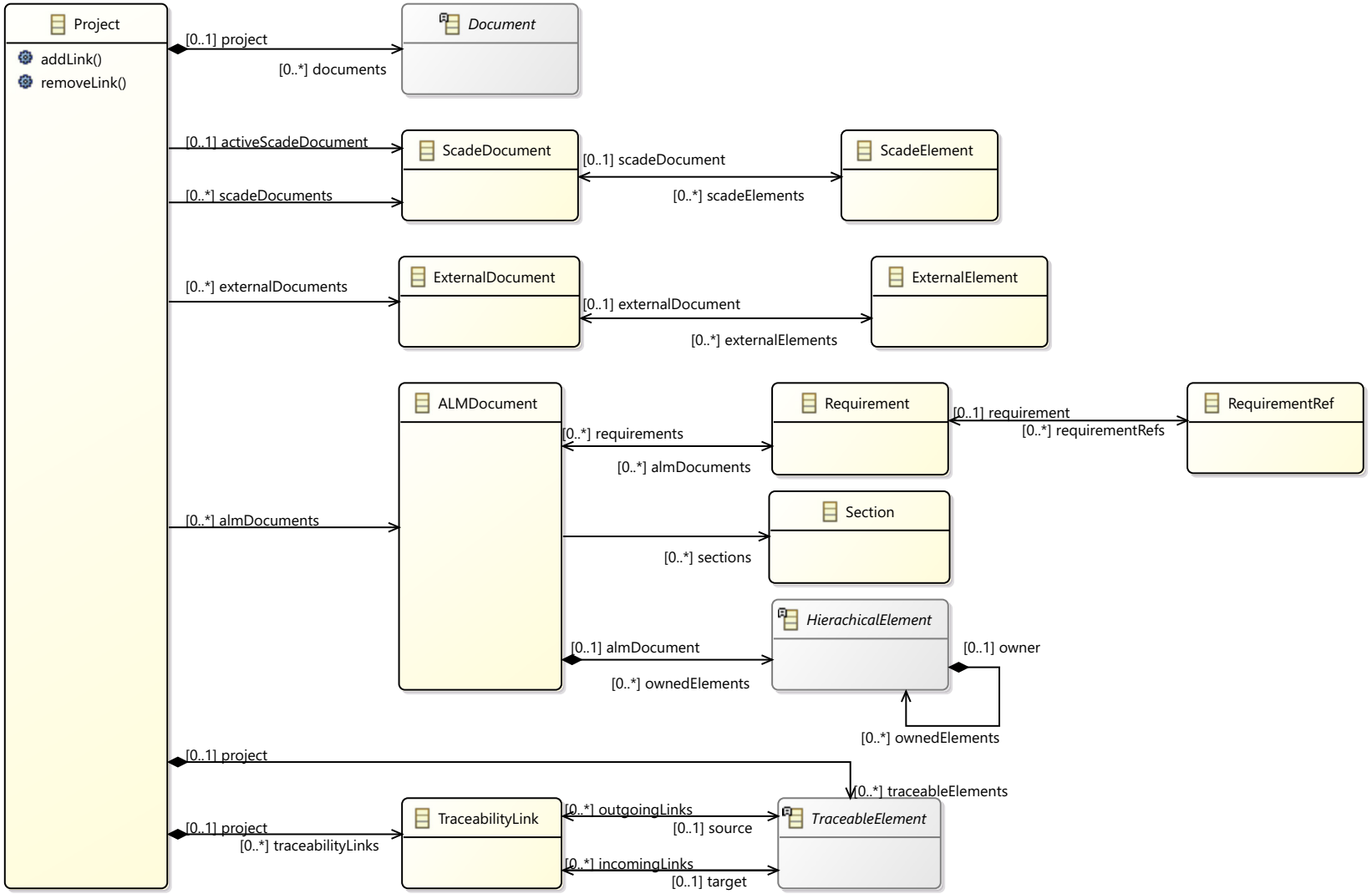
These metamodels present the data structure that give access to traceability data using Python API:

- "Traceability Project Model (inheritance)"
- "Traceability Project Model (composition)"

Traceability Project Model (inheritance)



Traceability Project Model (composition)



Contact Information

Submit questions to Technical Support at
https://customer.ansys.com

Contact one of our Sales representatives at
scade-sales@ansys.com

Direct general questions about SCADE products to
scade-info@ansys.com

Discover the latest news on our products at
www.ansys.com/products/embedded-software

Revision: SCF-API-TRC-25 - 23/04/25