A Proposal of a Dynamic Core for UML Metamodelling with MML
April 2001

G. Reggio - E. Astesiano
DISI Università di Genova - Italy
{reggio,astes}@disi.unige.it

Abstract

We present an initial proposal for an extension of MML, the basic language of [1], by adding a dynamic core and a basic visual notation to represent behaviour. Then we give some hints on how to use this extension for metamodelling UML covering the behavioural-dynamic aspects. The presentation will follow the style of [1].

This report has to be considered complementary to [1].

1 Introduction

We present an initial proposal for a dynamic extension of MML, the basic language of [1], apt to cover the dynamic/behavioural aspects of an object-oriented notation.

In this report we use MML, as introduced in [1], to present its dynamic extension, and try to follow as much as possible the approach of [1] to the metamodelling; we will explicit mention and motivate anytime we depart from it, using the special notation **Differences with [1]**..

In Fig. 1 we presents the packages with their relationships that we introduce in this report, together with those already present in [1] that we reuse. The last two packages (written in Italic) are under preparation, and will consist of the rephrasing following the metamodelling technique of the precise definition of statemachines of [2] (*StateMachine*) and of the definition of the use cases proposed by Perdita Stevens in [3] based on labelled transition systems.

Differences with [1] *In this report we prefer to use a more standard terminology, thus we call the first two parts of each package:* Abstract Syntax *and* Semantic Domains , *instead of* Model *and* Instances*; other possible names for them may be* Syntactic Structure *and* Meanings *or* Semantic Meanings.

Since this is a very initial report, we have not considered the part about the concrete syntax of the various packages.

To make the presentation of the various packages more compact and readable we introduce some “macros” reported in Appendix A; following a UML style we present them as

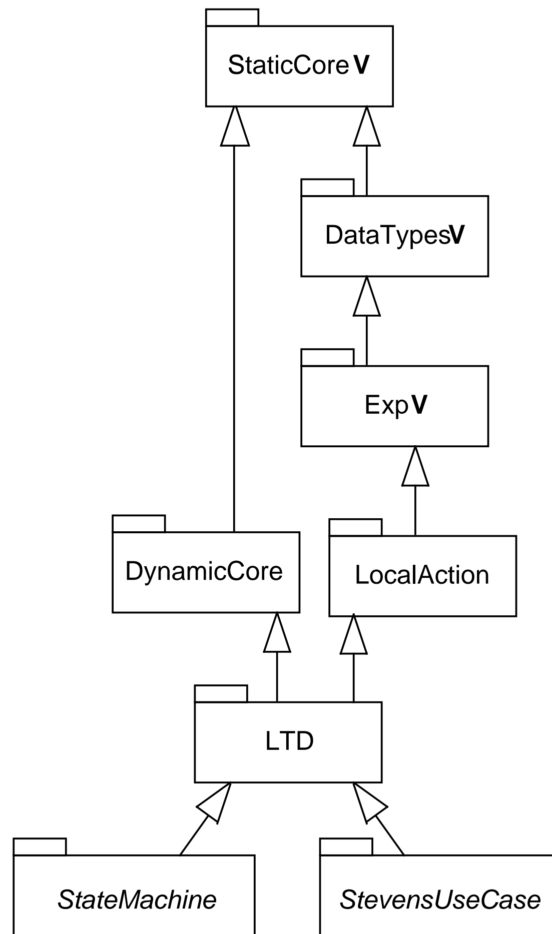


Figure 1: Components of the "dynamic" MML

stereotypes, but they are really shortcuts for constraints following particular patterns that are frequently repeated.

In this initial proposal in some cases the constraints associated with the various pakages are presented only informally by English text, but clearly it is possible to precisely rephrased them using OCL. In some other cases, mainly when the OCL constraints are difficult to read, we have tried to experiment another notation; FOCL (for First Order Constraints Language), which differ from OCL mainly for the syntax of the various operations, FOCL tries to use the usual first order syntax as much as possible.

2 Revised Basic Packages

Differences with [1] *In [1] the values (of expressions) are* instances. *For example, a value of an expression having type* C, *where* C *is class, is an instance of* C, *precisely intended as a configuration (state) of an object of class* C *at a point in time. A special subclass of instances (*Datavalues*) corresponds to the values of the predefined data types of MML. We prefer to introduce explicitly a* concept *for values, and to have that the values with type* C *(a class) are the identities of objects of class* C.

The motivation is twofold.

- *First, using object identities instead of object states as values makes extremely simple to provide the semantics of objects in isolation; indeed we have only indirect references to other objects, avoiding that the containment relations between objects and slot have loops. In the case of active objects expressed by labelled transition systems, this corresponds to using object identities in the transition labels (very much in the style of CCS et similia) and this is a good basis for an easy compositional semantics.*
- *Second, having values as first class citizens not only looks more natural but avoids some puzzling things, like having many "zero values" with different identity (that happen when data values are objects). Notice that UML explicitly states that data values are not objects, so they are without identity.*

We denote the revised versions of the packages of [1], characterized by the explicit presence of values, by the suffix **V** and report their definitions in the following three subsections.

Differences with [1] *We think that to have at hand a package just handling the expressions, and not the constraints, may help to metamodel the various aspects of a notation. Such expression package may be used to build the constraint package, but also many other packages that have nothing to do with constraints, as methods and actions. And so here, differently from [1], we will have as basic package* Exp *instead of* Constraints.

2.1 StaticCoreV package (StaticCore with explicit values)

2.1.1 Abstract Syntax

We report this package in Fig. 2.

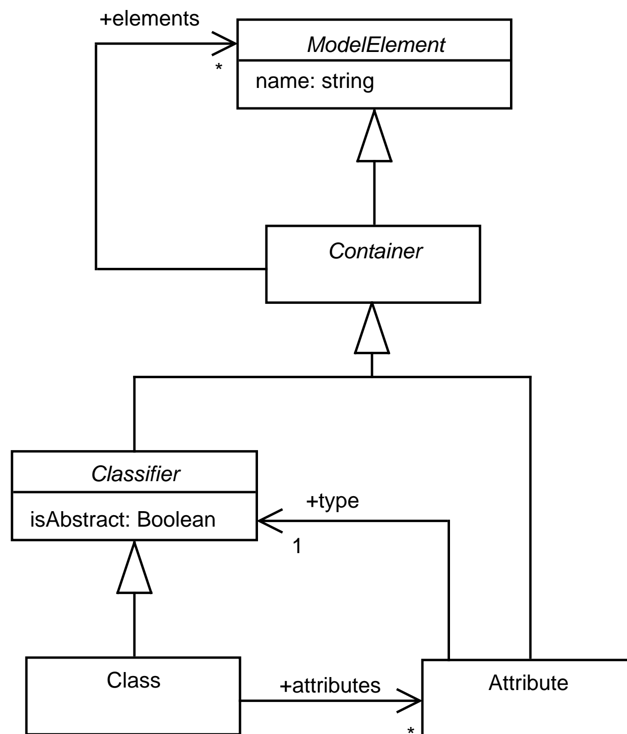


Figure 2: core**V**.abstract syntax.concepts Package

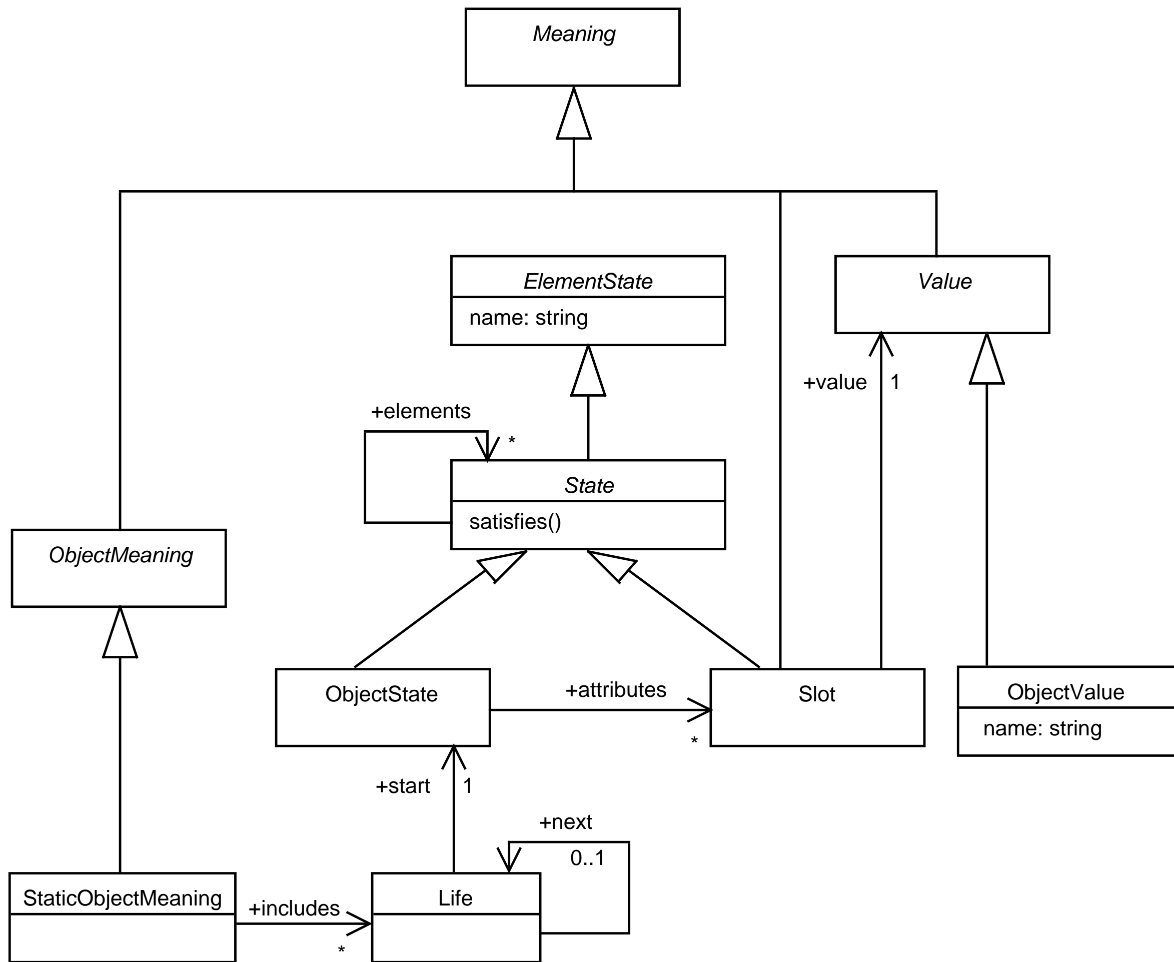


Figure 3: core**V**.semantic domains.concepts Package

In this picture, as in the following, we write using the *italic* font the names of the abstract classes, i.e., those which cannot be directly instantiated.

Differences with [1] *Here and in the following for simplicity we drop the part concerning generalization, because we think that it is orthogonal w.r.t. the dynamics issue; so it can be introduced later without problems.*

Here, to follow UML Attribute *is a specialization of* Container *but not of* Classifier*.*

2.1.2 Semantic Domains

We report this package in Fig. 3.

Methods

attributes This method given an object state returns the set of names of all slots associated with such state, i.e., the names of the attributes of the object of which it is a state.

```
allAttributes: ObjectState -> Set(string)
allAttributes(os) = os.attributes -> select{ sl |  sl.name}
```

Well-formedness rules

- Two object values are different iff the values of their attribute name are different; this constraint guarantee that the attribute name truly corresponds to an object identity.

  ```
  context ObjectValue inv:
       ObjectValue -> AllInstances ->
            forall{ov | ov <> self  implies  ov.name <> self.name}
  ```

  (Below there is the same constraint using the more "standard notation" FOCL.)

  ```
  for all ov1, ov2: ObjectValue .
       ov1 <> ov2  implies  ov1.name <> ov2.name
  ```

- All states of a "life" must be states of the same object, thus they must have the same name and the same attributes.

  ```
  context Life inv:
       (self.next->size =1 implies
            (self.start.name = self.next.start.name) and
            (self.start.allAttributes = self.next.start.allAttributes))
  ```

- All elements of an instance of StaticObjectMeaning must be lives of objects of the same class, thus they must have the same attributes. *(This constraint is not written in OCL, it uses a more "standard notation" FOCL.)*

  ```
  for all so: StaticObjectMeaning .
       for all lf1, lf2: Life .
            lf1 in so.includes and lf2 in so.includes implies
            lf1.start.allAttributes = lf2.start.allAttributes
  ```

Differences with [1] *We use the word "state" instead of "instance", because we think that it is more catching. Indeed, for example an instance of class Object in [1], now called ObjectState, is not an MML object, but just a description of a particular moment in the life of such object, thus it is an object state or an object configuration. Similarly, the instances of the class Slot correspond to the possible states of the attributes; moreover in this case the concept of instance of an attribute is not very clear. This choice seems also coherent with the subsequent use of instances in [1] when they are used to define configurations; now a configuration contains the actual states of the model elements actually existing.*

Furthermore, we do not think that a set of elements of the class ObjectState (Object in [1]) may be the semantic value/meaning of an MML class, because it is not clear how the object lives are represented (clearly, we assume that objects are persistent and updatable, and so

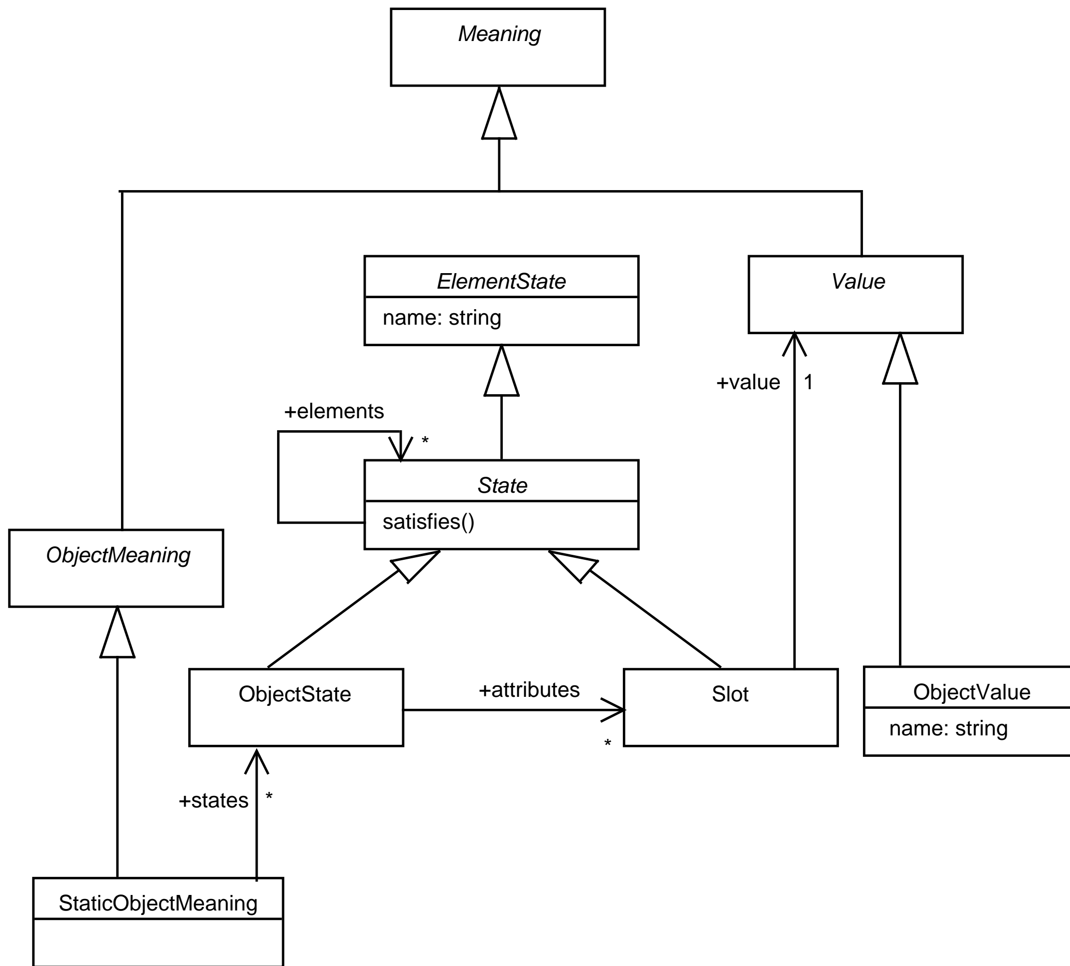


Figure 4: core**V**.semantic domains.concepts Package Simpler Version

they may have different lives). We have, thus, introduced a new metaclass ObjectMeaning *and a refinement of it,* StaticObjectMeaning*, whose elements are the descriptions of all possible lives of some objects; a single life is simply intended as an ordered sequence of object states.*

We specialize the class ObjectMeaning *as* StaticObjectMeaning *instead of defining it directly as a set of lives, because when introducing the active objects it will be refined in a different way.*

In this case the meaning of an object is the set of its lives, where a life is a sequence (finite or infinite) of object states. In the case that the language does not offer any way to restrict the possible lives of the object of a class (e.g., a constraint saying that the value of an integer attribute cannot be decreased), and so when from one state it can go into any other state, ObjectMeaning may be defined in a simpler way, as the set of all possible states of the object. In such case Fig. 3 may be replaced by Fig. 4.

2.1.3 Semantics

We report this package in Fig. 5.

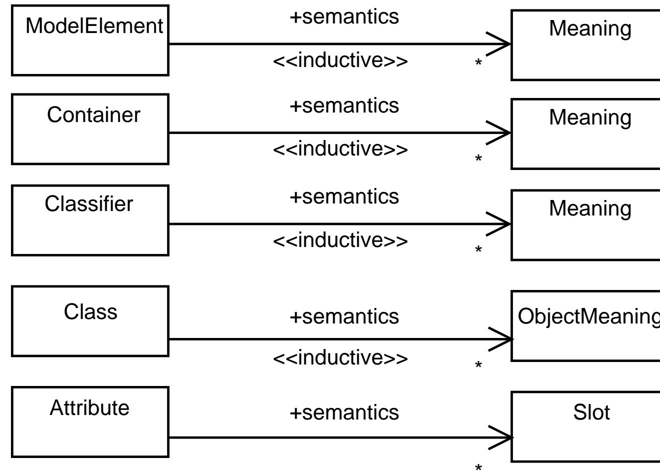


Figure 5: core**V**.semantics Package

Differences with [1] *To simplify the diagram here we prefer to name only the association between abstract syntactic domain and meaning, as* **semantics**, *instead of using* **semantics** *for an association end and* **of** *for the other.*

In Fig. 5 we use the association stereotype <<inductive>>, defined in Appendix A. The third <<inductive>> starting from the top corresponds to assert that:

- the association semantics between Attribute and Slot specializes the one with the same name from Classifier and Meaning, i.e., if a classifier C admits the type Attribute, then it must be related by semantics to elements admitting the type Slot, and vice versa. Similarly for the association semantics between Class and ObjectMeaning.

- the association semantics commutes with the association attributes between Class and Attribute and the one with the same name between ObjectStates and Slot i.e.,

  `C.attributes.semantics = C.semantics.attributes .`

Methods

attributesC This method returns the set of names of the attributes of a class.

```
attributesC: Class -> Set(string)
attributesC(c) = c.attributes -> select{ at | at.name }
```

Well-formedness rules

- The object meanings associated with a class must be correct, i.e., they must be made by states having the appropriate attributes.

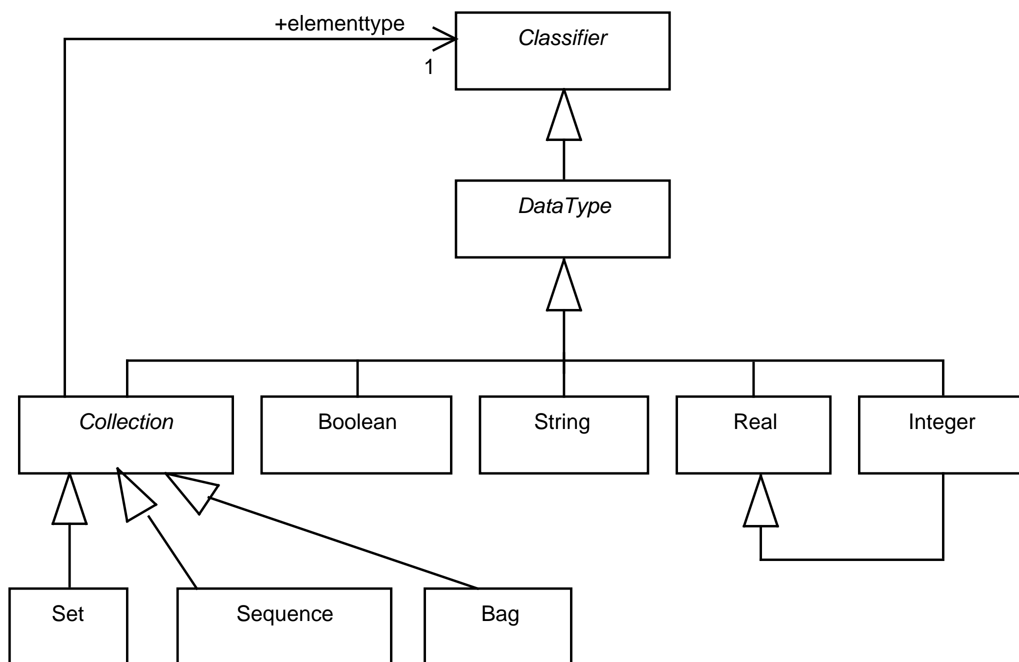


Figure 6: datatypes.abstract syntax.concepts Package

```
context Class inv:
    (self.semantics.includes.start.allAttributes = self.attributesC)
```

- The slots associated with an attribute must be correct, i.e., they must have the same name and the same type.

2.2 DatatypesV package (Datatypes with explicit values)

2.2.1 Abstract Syntax

We report this package in Fig. 6. It is as the one of fig. 12 of [1], except that the dependency relationships have been replaced by specialization.

2.2.2 Semantic Domains

We report this package in Fig. 7.

Well-formedness rules

- There are no two counts associated with a bag value associated with the same value by `element`, and no count having the `value` attribute equal to 0.

- The elements of the counts associated with a bag value are all of the same class.

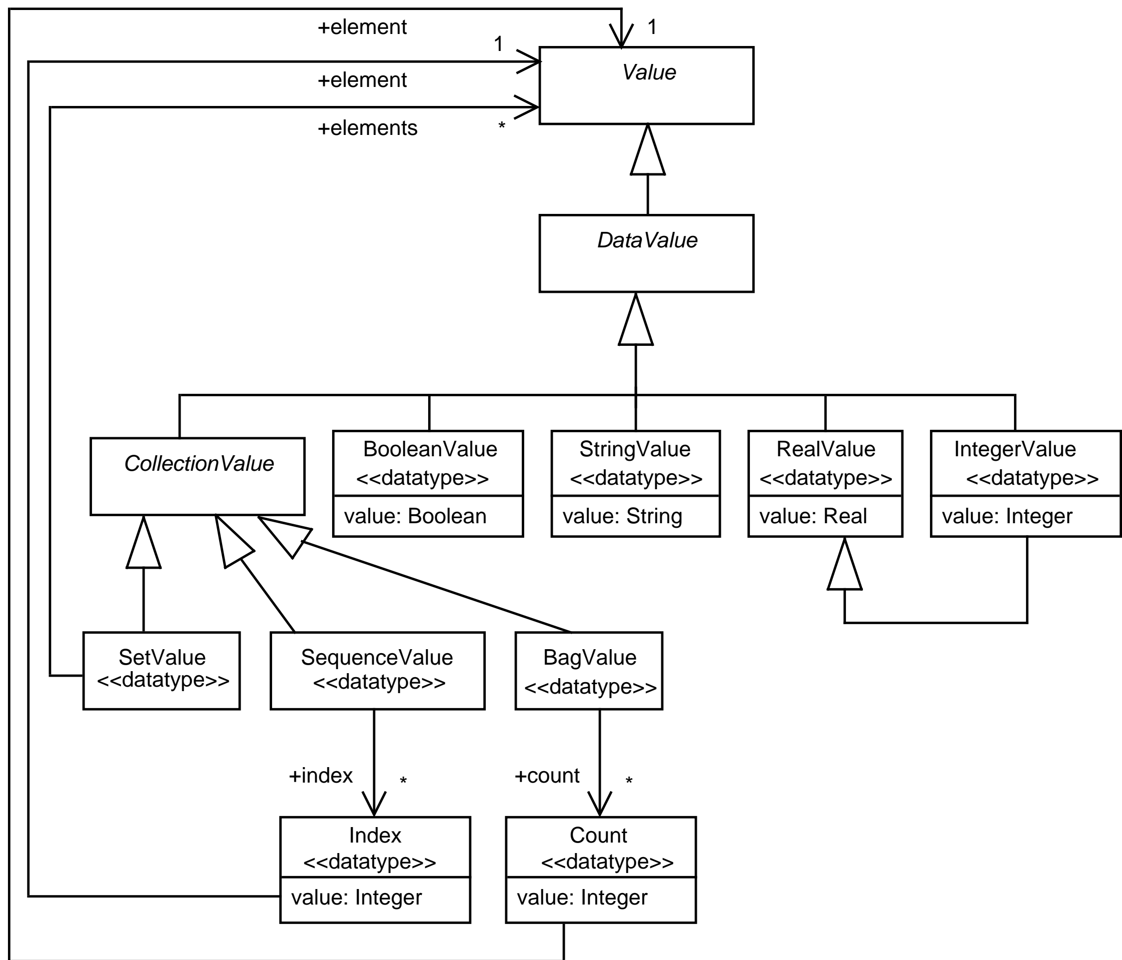


Figure 7: datatypes.semantic domains.concepts Package

- There are no two index associated with a sequence value having the value attribute equal; furthermore the values of the attribute value of all the indexes is an interval of the form [0 .. n].
- The elements of the indexes associated with a sequence value are all of the same class.
- The elements of a set value are all of the same class.

A problem with this package, in our opinion, is the absence of the definition of the operations of the introduced datatypes (recall datatype = sets of values + operations over them).

Differences with [1] *In this case we have that DataValue is a refinement of Value, instead of Instance, now renamed State.*

This seems in accord with the UML philosophy, because it asserts that datavalues are not objects, since they have not an identity and their state cannot be updated.

We have added also some trivial well-formedness rules, that we think are needed to guarantee that the elements of BagValue, SequenceValue and SetValue are really the expected values.

2.2.3 Semantics

We report this package in Fig. 8.

Well-formedness rules

- The semantics of a set (sequence) [bag] type S contains all and only the set (sequence) [bag] values whose elements belong to the class S.elementtype.

The stereotype <<inductive>> on the semantics association between *DataType* and DataValue guarantees that the semantics of Boolean are elements of the class BooleanValue, and similarly for all the other types. Analogously, the same stereotype on the on the semantics association between *Collection* and CollectionValue guarantees that the semantics of Set are elements of the class SetValue, and similarly for Sequence and Bag.

2.3 ExpV package (Exp with explicit values)

Here for simplicity, we do not report the parts of this package about the iterate construct, because we think that it is orthogonal w.r.t. the dynamics issue, and so it can be introduced later without problems.

2.3.1 Abstract Syntax

We report this package in Fig. 9.

We have assumed that self may appear within an expression because it is seen as a variable.

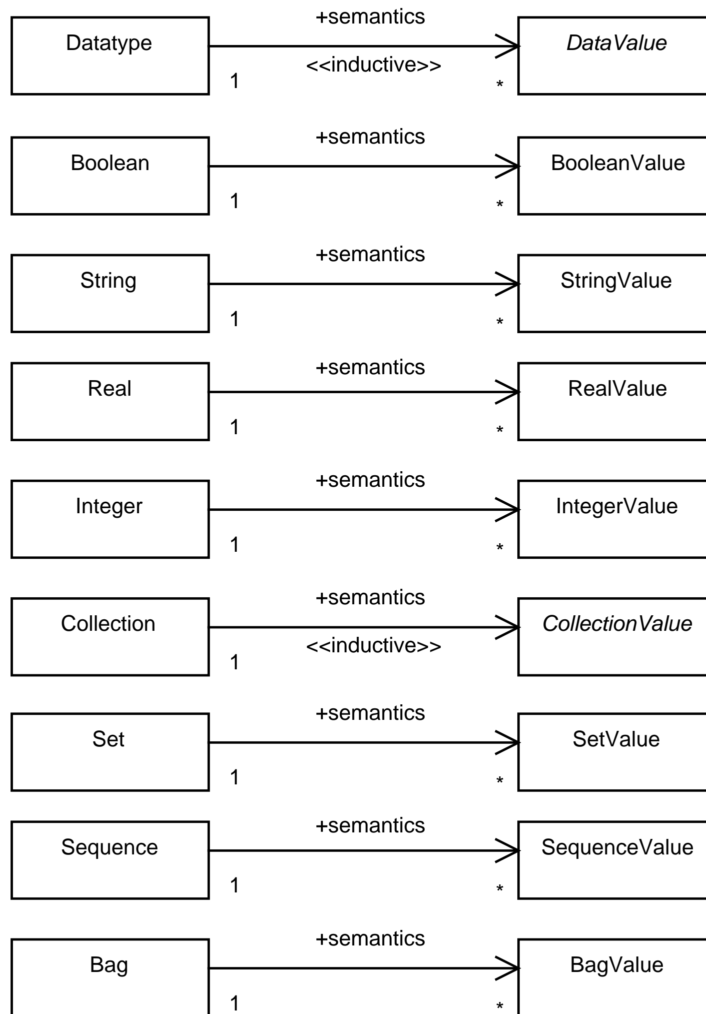


Figure 8: datatypes.semantics Package

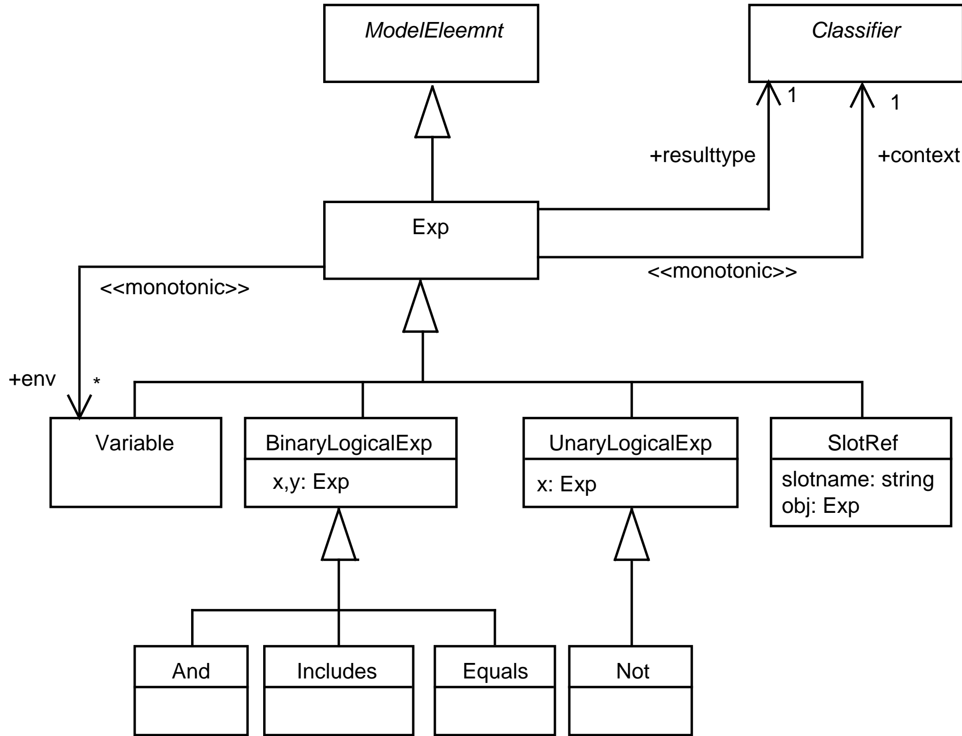


Figure 9: Exp.abstract syntax.concepts Package

Differences with [1] *We think that any expression should always have associated a context class, i.e., the one where it is evaluated, to help check its static correctness (think, e.g., attributes and operations that are only locally visible). The context class will be also the one giving the type to the self implicit variable.*

Here, similarly to UML, expressions are a specialization of ModelElement *and not of* Classifier.

Well-formedness rules The stereotype <<monotonic>> on the association context, defined in Appendix A, requires that the context of all subexpressions of an expression E (those given by the attributes x, y and obj) is the same of E, because the association context has multiplicity 1 on the Classifier end. Similarly, <<monotonic>> on the env association requires that the environment of a subexpression of E is included in that of E.

- The environment associated with an expression contains all the variables appearing in such expression and a special standard variable self whose type is the context class.
- The environment does not contain repeated variables.
- Any expression is correctly typed.
- The *resulttype* of an expression is correct (i.e., either a predefined data type, or an existing class).

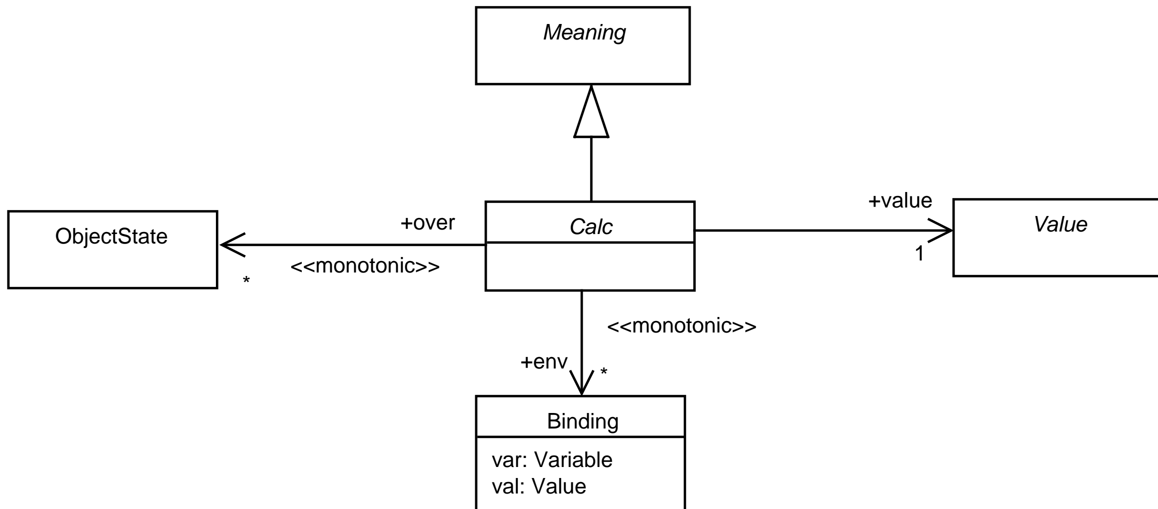


Figure 10: Exp.semantic domains.concepts Package

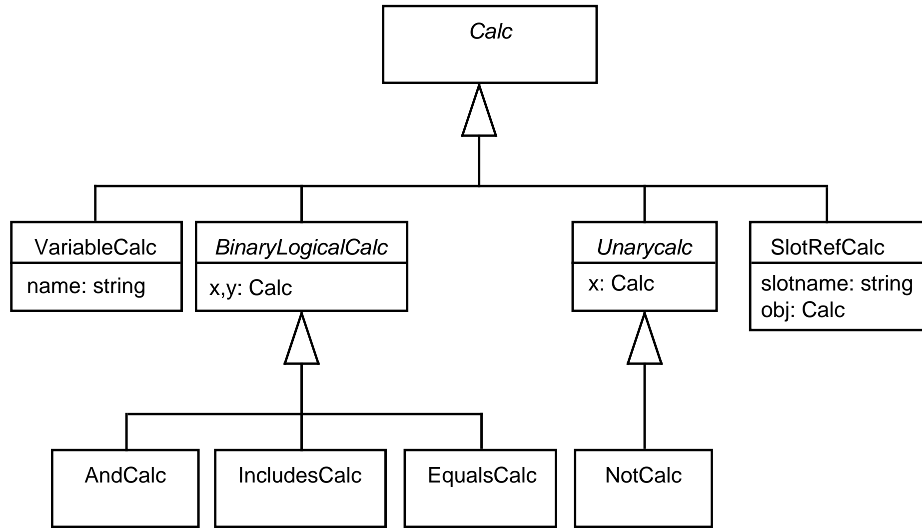


Figure 11: Calc

2.3.2 Semantic Domains

We report this package in Fig. 10 and 11.

The expression calculations (class Calc) are defined as a particular instance of the pattern **calculation** presented in Appendix A.

Differences with [1] *The calculations of [1], that are the semantic interpretations of expressions, roughly correspond to say that the meaning of an expression associates bindings of free variables with values, and this may seem strange, because usually the meaning of an expression in an imperative setting is something that associates bindings of free variables and states (in this OO setting object states giving the value of attributes) with values.*

Here we propose to introduce the over *association that links a calculation with the states of the objects over which an expression is calculated. This solution, which is now required having introduced object identities as object values, has the advantage of making explicit the fact that for evaluating an expression you have to take into account both a binding and some states (for getting the current attribute values); this is usual in the classical (denotational and operational) semantics. Thus, now a* Calc *would be a triple* <*binding,states,value*>.

Notice that now Calc *is a refinement of* Meaning *and not of* Instance*, and that we have completed the definition of* Binding.

We have also added some attributes to the slot and variable calculations, because in this case Calc *is not a refinement of* Instance*, and thus a calculation has not always a* name *attribute.*

We give also the new WF rules for this package; notice that some of them were considered as part of the semantics package in [1].

Well-formedness rules Notice that the <<monotonic>> stereotypes on the associations env and over require that the environment of any subcalculation of a calculation C is a subset of that of C, similarly for the set of the states over which C is performed.

- The value of an AndCalc is the logical conjunction of the values of its x and y components.

```
context AndCalc inv:
     self.value = self.x.value and self.y.value
```

- The value of an IncludesCalc is true iff the value of its x component includes the value of its y component.

```
context IncludesCalc inv:
     self.value = ((self.x.value) -> includes self.y.value)
```

- The value of an EqualsCalc is true iff the values of its x and y components coincide.

```
context EqualsCalc inv:
     self.value = (self.x.value = self.y.value)
```

- The value of a NotCalc is the logical negation of the value of its x component.

```
context NotCalc inv:
     self.value = not self.x.value
```

- The value of a SlotRefCalc is the value of the attribute identified by the value of its attribute slotname in the state of the object identified by the value of its obj attribute (such object state must be included in those related by the association over).

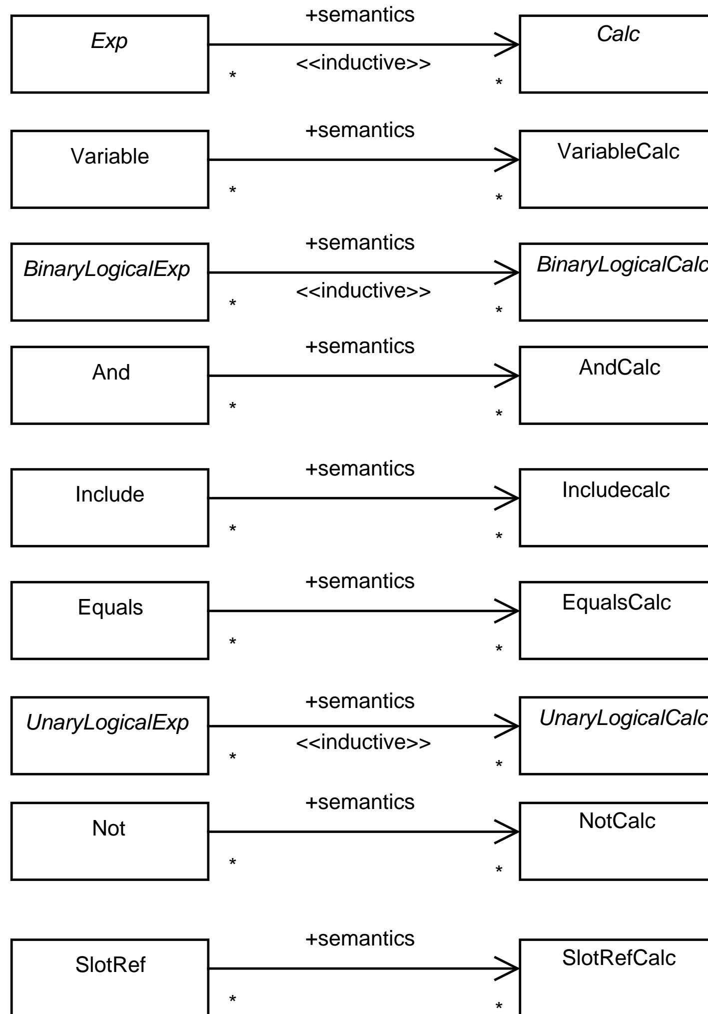


Figure 12: Exp.semantics Package

```
context SlotRefCalc inv:
     self.value =
     ((self.over -> select(os.name = self.obj)).slot ->
          select(sl.name = self.slotname)) -> collect(sl.value)}
```

- The value of a VariableCalc is the value associated with it by the `env` association.

```
context VariableCalc inv:
     self.value =
     ((self.env -> select(bd.variable = self.name)) -> collect(bd.value))}
```

2.3.3 Semantics

We report this package in Fig. 12.

Well-formedness rules Recall that the stereotype <<inductive>> of the semantics association requires that the semantics of a binary logical expression will be a binary logical calculation, and that the semantics of its x and y attributes are the semantics of the x and y attributes of its semantics; similarly for the other cases.

- The binding of a calculation that is the semantics of an expression E must evaluate all the variables in the environment of E and with values of the correct types.

- The semantics of an expression cannot contain two calculations which cannot be distinguished by using the associations over and env.

3 Dynamic Core Package (Active Classes in MML)

The package dynamic core introduces in MML active classes[1], i.e., classes whose instances are processes, which are called active objects. Here, differently than in UML, we assume that an active object corresponds to a unique thread.

This package introduces an abstract model element for describing the behaviour of the instances of an active class, *behaviour description*, but since we can consider different realizations of it we do not fix its form. Other packages introduced in this paper will present some possible realization of it, as the statemachines, but forthcoming extensions will be able to introduce completely different ways.

3.1 Abstract Syntax

We report this package in Fig. 13.

Active classes are a specialization of Class from the static core package, see Sect. 2.1.

An active object (i.e., a process) should have the possibility to communicate with its outside world (e.g., with the other active objects of the system), thus a presentation of an active class should include a description of such means to communicate/exchange information with the outside world. Such "communication means" are the instances of the abstract (meta)class Communication, and the association communications links them to active classes. Other packages will introduce appropriate refinement of this abstract concept; for example, when behaviour description is refined into statemachine, they will be the signals and the operations generating call-events.

3.2 Semantic Domains

We report this package in Fig. 14.

- The meaning of an active object is a labelled transition systems (an element of class LTS).

[1]We try to use for MML the same terminology of UML whenever possible.

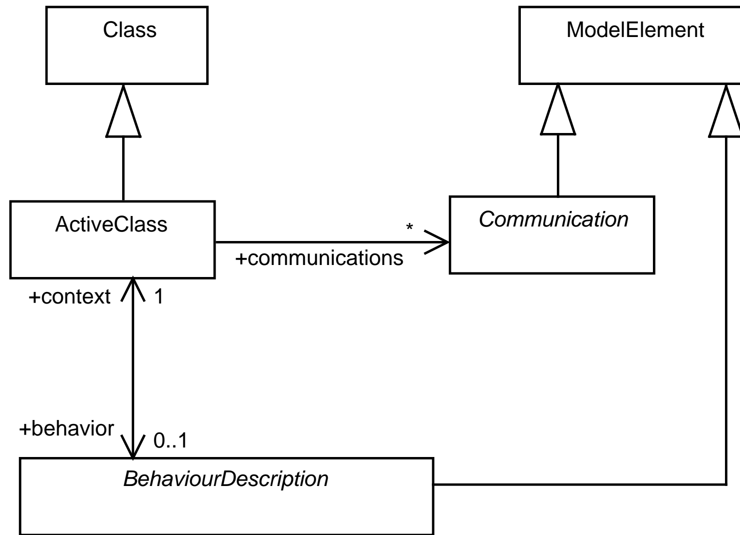


Figure 13: dynamic core.abstract syntax.concepts Package

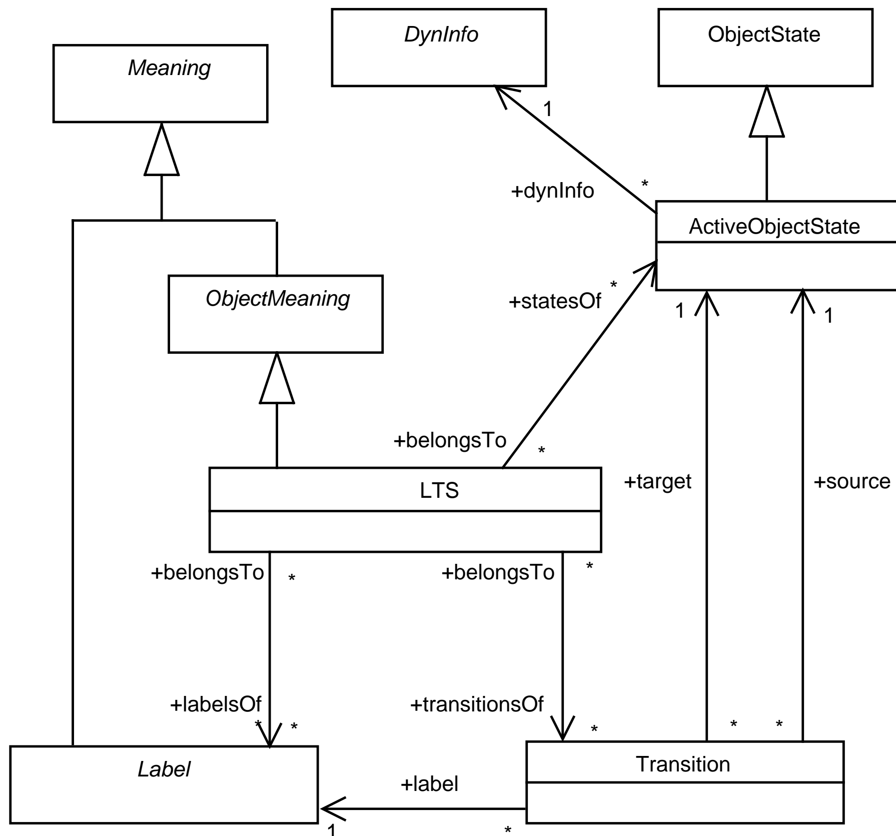


Figure 14: dynamic core.semantic domains.concepts Package

- An lts is characterized by its states (in this particular setting they are just states of active objects), labels (not further detailed here) and transitions (triples consisting of a source state, a label and a target state).
- A state of an active object cannot be simply characterized by the values of its attributes and by its identity (attribute name), as for the non-active ones, but it must include some additional information about its past behaviour that can influence its future behaviour (we call them in general *dynamic information*).

  The dynamic information are not further detailed in this package (indeed the class DynInfo is abstract); their form will depend on the chosen refinement of BehaviourDescription.

Notice, that here for simplicity we do not consider the initial and final states of the lts's; however they could be added later without problems.

Well-formedness rules

- All states of an lts must have the same name and must have the same attributes (i.e., they should be states of the same object).

```
context LTS inv:
     self.statesOf.name -> size = 1 and
     self.statesOf -> forall{ st1 | self.statesOf ->
                         forall{st2 | st2.allAttributes = st1.allAttributes}}
```

  alternative presentation using “FOCL”

```
context LTS inv:
      forall st1, st2 in self.statesOf
          st1.name = st2.name  and
          st1.allAttributes = st2.allAttributes
```

- The source and the target of each transition of an lts must belong to the states of lts itself.

```
context LTS inv:
     self.statesOf -> includes
          ((self.transitionsOf -> { tr | tr.source }) -> union
           (self.transitionsOf -> { tr | tr.target }) )
```

- The label of each transition of an lts must belong to the labels of lts itself.

```
context LTS inv:
     self.labelsOf -> includes (self.transitionsOf -> { tr | tr.label })
```

- A transition is completely determined by its source and target states and its label.

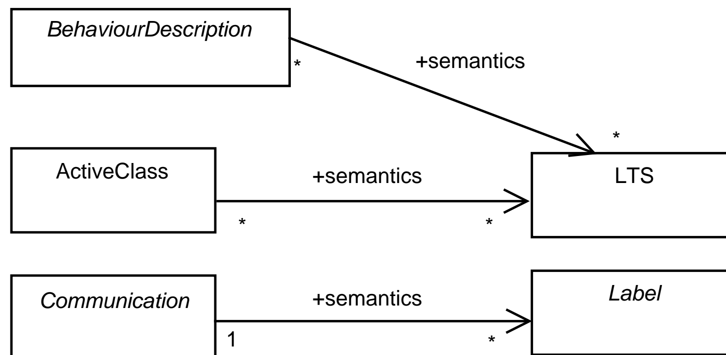


Figure 15: Dynamic Core.semantics Package

```
context Transition inv:
     Transition.AllInstance->forall{ tr |
          tr.source = self.source and
          tr.target = self.target and
          tr.label = self.label   implies tr = self }
```

3.3 Semantics

We report this package in Fig. 15.

Well-formedness rules The stereotype <<inductive>> on the semantics association between Class and ObjectMeaning (see Fig. 5) ensures that the semantics of an active class will be made by LTS's and that a nonactive class will never get as semantics an LTS.

- If an active class has associated a behaviour description via the association behaviour, then their semantics coincide.

```
context ActiveClass inv:
     (self.behaviour -> notEmpty) implies
          (self.semantics = self.behaviour.semantics)
```

- Let LTS be the semantics of an active class AC
  - each states of LTS is a state appropriate for class AC

    ```
    context ActiveClass inv:
         self.semantics.statesOf-> forall{ st | self.attributesC = st.allAttributes}
    ```

  - each labels of LTS is the semantic interpretation of a communication of AC

    ```
    context ActiveClass inv:
         (self.communications.sematics) -> includes(self.semantics.labelsOf)
    ```

3.4 Open Hot Points

There are some points that need further investigation to be settled.

Generalization for active classes (lts) The meaning of generalization in the case of an active class, that is the meaning of generalization for lts, must be formulated, and this is not a trivial point (Harel has stated in some talk that this is one of the theoretical points to settle in the UML landscape).

A minimal proposal is the following. We can refine an lts just by refining its states and labels and by adding more transitions; and an active class just by refining its attributes, its communications and by adding more attributes, more communications and more transitions.

Constraints on active classes/lts An OCL-like language is not sufficient to express all the relevant constraints on the behaviour of active objects, think for example of liveness properties. In UML some of the properties on active objects are expressed using particular diagrams (as sequence and collaboration) but not with OCL. However, we think that it would be very useful if this basic core is be able to express such properties as constraints, having in mind that it will be used for metamodelling.

4 **Local Action** package (sequential /local/encapsulated statements)

This package introduces what we call *local actions*, that are *sequential*, *local* and *encapsulated* statements.

- Sequential means that are usual statements of classical imperative languages (e.g., assignments or control flow statements) and that do not involve communications with other objects.
- Local means that their possible side effects are localized to the object over which they are executed.
- Encapsulated means that to execute them there is no need to access the states of objects different from the one on which they are executed.

Thus, the semantic interpretation of a local action is just a pair of states of the object on which it is executed (the state before and the one after its execution).

4.1 Abstract Syntax

We report this package in Fig. 16.

The association `env` is similar to the association with the same name in the `Exp` (see Sect. 2.3) and associates the action with a set of local variables – its environment. The association `context` instead gives the class of the objects over whom the action may be executed.

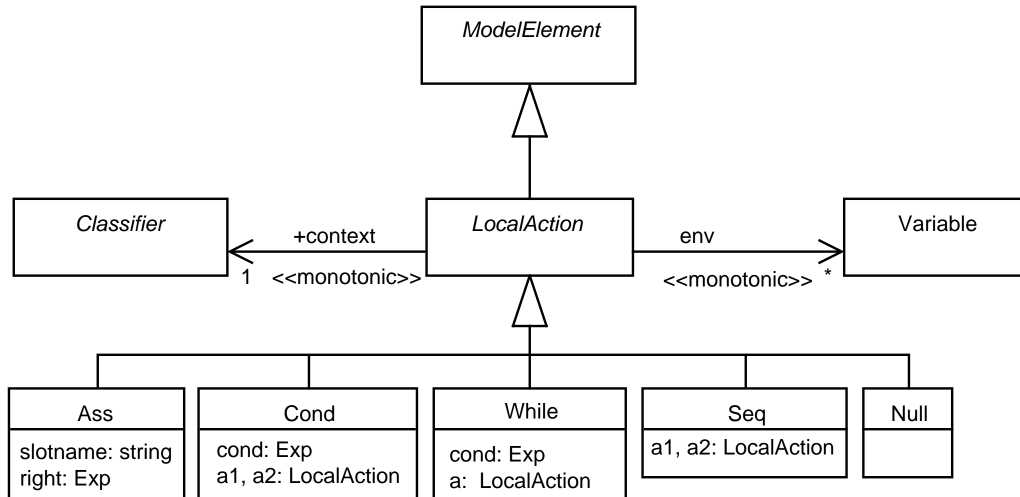


Figure 16: local action.abstract syntax.concepts Package

Here we have that LocalAction is a specialization of ModelElement, whereas in UML the class Action seems to be a root in the specialization hierarchy of the metamodel.

Well-formedness rules The <<monotonic>> stereotype on the association context requires that the context is the same for a local action and all its subactions, and also for the composing expressions. Whereas the same stereotype on the association env guarantees that the environment of an action extends those of its subcomponents.

- The environment does not contain repeated variables.
- The value of the cond attribute of a Cond local action must have boolean as result type.
- The value of the cond attribute of a While local action must have boolean as result type.
- The type of the value of the slotname attribute of a Assignment local action must be the result type of the value of its right attribute.
- The value of the slotname attribute of a Assignment local action must be an attribute of the class linked to the local action by the association context.

4.2 Semantic Domains

We report this package in Fig. 17.

The semantics of a local action is a collection of local executions (instances of class LocalExec), and each local execution is characterized by a pair of active object states, the one before and one after the execution of the action. The properties on the local executions listed below guarantee that this

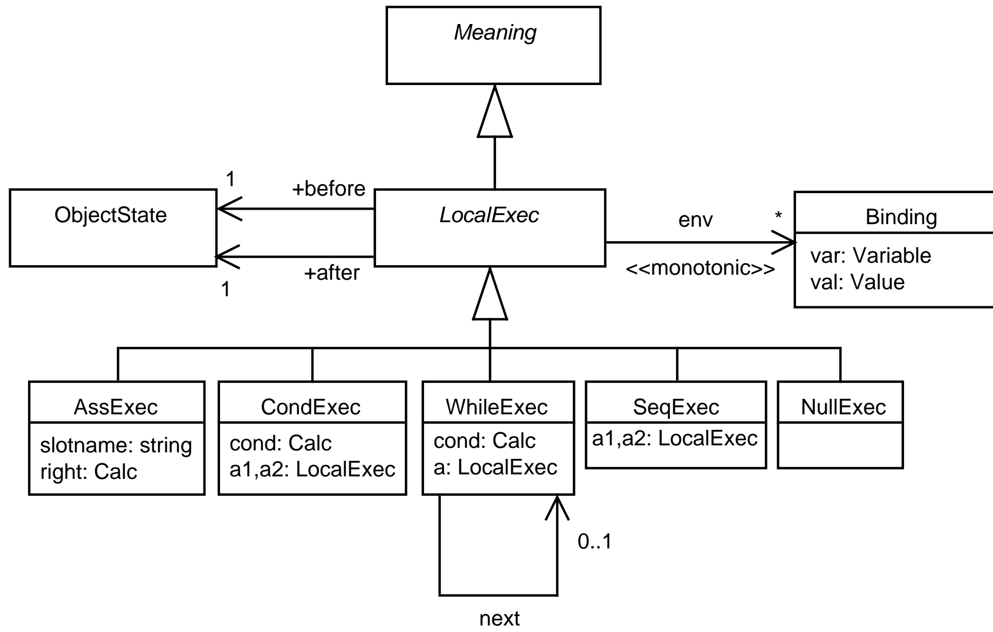


Figure 17: Local Action Domains

Well-formedness rules The <<monotonic>> stereotype on the association env guarantees that the environment of a local execution extends those of its subcomponents (local actions and expressions).

- The environment associated with a LocalExec does not contain two pairs for the same variable.

```
context LocalExec inv:
     self.env -> forall{ bd | self.env -> forall{ bd' |
          bd <> bd'  implies bd.var <> bd'.var} }
```

- The expression subcomponent of an assignment execution is calculated over the starting state.

```
context AssExec inv:
      self.before = self.right.over
```

- The assignment local action updates only the value of the assigned attribute.

```
context AssExec inv:
     self.after.slot -> includes
          (self.before.slot -> select { sl |
               sl.name = self.slotname and sl.value = self.right.value}
          union
          self.before.slot -> select{ sl | sl.name <> self.slotname})
```

- The expression subcomponent of a conditional local execution is calculated over the starting state.

```
context CondExec inv:
     self.before = self.cond.over
```

- The expression subcomponent of a while local execution is calculated over the starting state.

```
context WhileExec inv:
     self.before = self.cond.over
```

- Because expressions have no side effects, the execution of a conditional action is either the execution of its `a1` component or of its `a2` component, depending on the value returned by the calculation of its condition, over its starting state.

```
context CondExec inv:
     (self.cond.value = True Implies
          self.before = self.a1.before and self.after = self.a1.after)
     and
     (self.cond.value = False Implies
          self.before = self.a2.before and self.after = self.a2.after)
```

- In a while local execution the subcomponents `cond` and `a` are the same of the next local execution, if present.

```
context WhileExec inv:
     (self.next -> size = 1) implies
          self.cond = self.next.cond and self.a = self.next.a
```

- While

```
context WhileExec inv:
     (self.cond.value = False Implies
          self.before = self.after and self.next -> size = 0)
     and
     (self.cond.value = True  Implies
          self.next -> size = 1 and
          self.before = self.a.before and
          self.a.after = self.next.before and
          self.after = self.next.after
```

- Sequential concatenation

```
context SeqExec inv:
     self.before = self.a1.before and
     self.a1.after = self.a2.before and
     self.after = self.a2.after
```

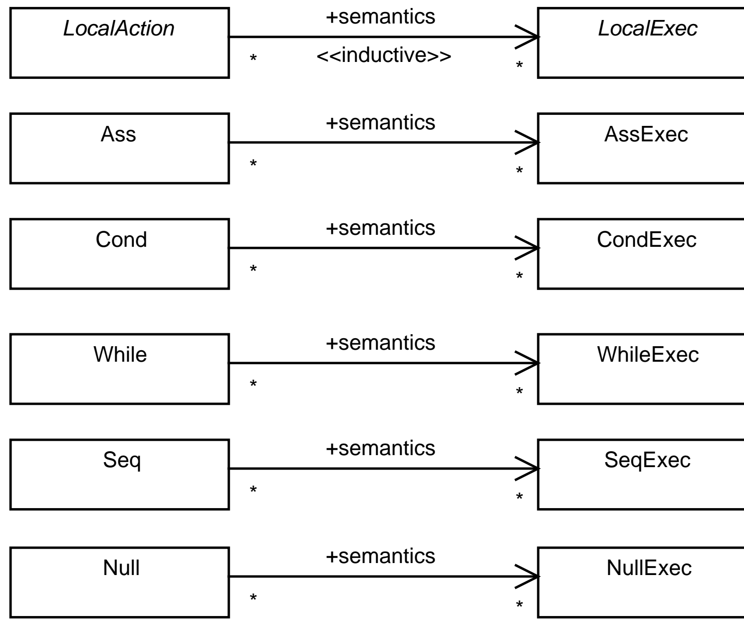


Figure 18: Local Action.semantics Package

- The null local action does not change the object state.

```
context NullExec inv:
     self.before  = self.after
```

4.3 Semantics

We report this package in Fig. 18.

Well-formedness rules The stereotype <<inductive>> on the semantics association requires that the semantics of an assignment will be an instance of AssExec, and that the semantics of its right attribute is the value of the right attribute of its semantics; similarly for the other cases.

- The binding of a local execution that is the semantics of a local action LA must evaluate all the variables in the environment of LA itself and with values of the correct types.

5 LTD (Labelled Transition Diagrams) package

Labelled Transition Diagrams are a basic visual notation to describe the labelled transition systems corresponding to the semantic interpretation of active classes. Technically, they will be a specialization of the metaclass BehaviourDescription of the Dynamic Core (see Sect. 3).

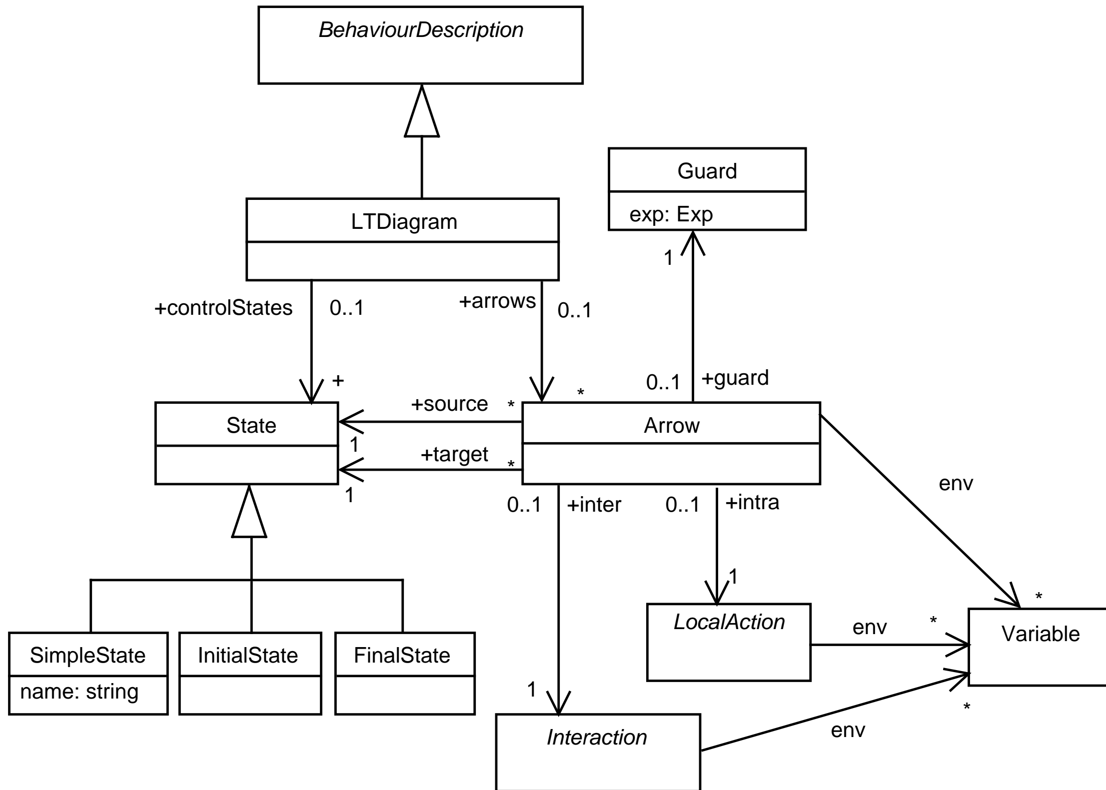


Figure 19: LTD.abstract syntax.concept

5.1 Abstract Syntax

We report this package in Fig. 19.

The states of an LTD, which should be better named control states, are similar to the states of the state machines. Indeed, the states of the represented lts consist of one of these states (i.e., a control state) plus an object state, giving the current values of the attributes of context active class.

An arrow describes a set of labelled transitions that are

- starting from states whose control part is the source, and whose attribute values satisfy the guard,
- reaching states whose control part is the target, and whose attribute values has been modified as describe by the intra part (that is a local action),
- labelled by the evaluation of the inter part, that is an interaction. Here Interaction is an abstract class, that will be refined whenever the package LTD will be used.

Notice, that it is possible to use free variables in an arrow (within the guard, the local action and the interaction parts); they with their types are abstractly represented by the env associations.

Well-formedness rules

- An LTD has exactly one initial (control) state.

```
context LTDiagram inv:
     (self.controlStates -> select{ s | InitialState ->isTypeof(s)}) -> size = 1
```

- There are no arrows leaving a final (control) state.

```
context Arrow inv:
     not (FinalState ->isTypeof(self.source))
```

- The names of the simple (control) states in an LTD are unique.

```
context LTDiagram inv:
     (self.controlStates -> select{ SimpleState | s -> isTypeof(s)}) -> size =
     ((self.controlStates ->
          select{ SimpleState | s ->isTypeof(s)}) -> collect(s.name)) -> size
```

- A (control) state that is the target or the source of an arrow of an LTD is a (control) state of the LTD.

```
context LTDiagram inv:
     self.states  -> includes
          (self.arrows -> collect{a.source} -> union(self.arrows -> collect{a.target}))
```

- The result type of the exp attribute of a guard is Boolean.

```
context Guard inv:
     self.exp.resulttype = Boolean
```

- The context of the exp attribute of the guard, of the local action and of the interaction of an arrow of an LTD is the same of the LTD.

```
context LTDiagram inv:
     self.arrows -> forall{a | a.guard.exp.context = self.context and
                               a.intra.context = self.context and
                               a.inter.context = self.context}
```

- The environment of the exp attribute of the guard, of the local action and of the interaction of an arrow is that of the arrow itself.

```
context Arrow inv:
     self.env = self.guard.exp.env and
     self.env = self.intra.env and
     self.env = self.inter.env
```

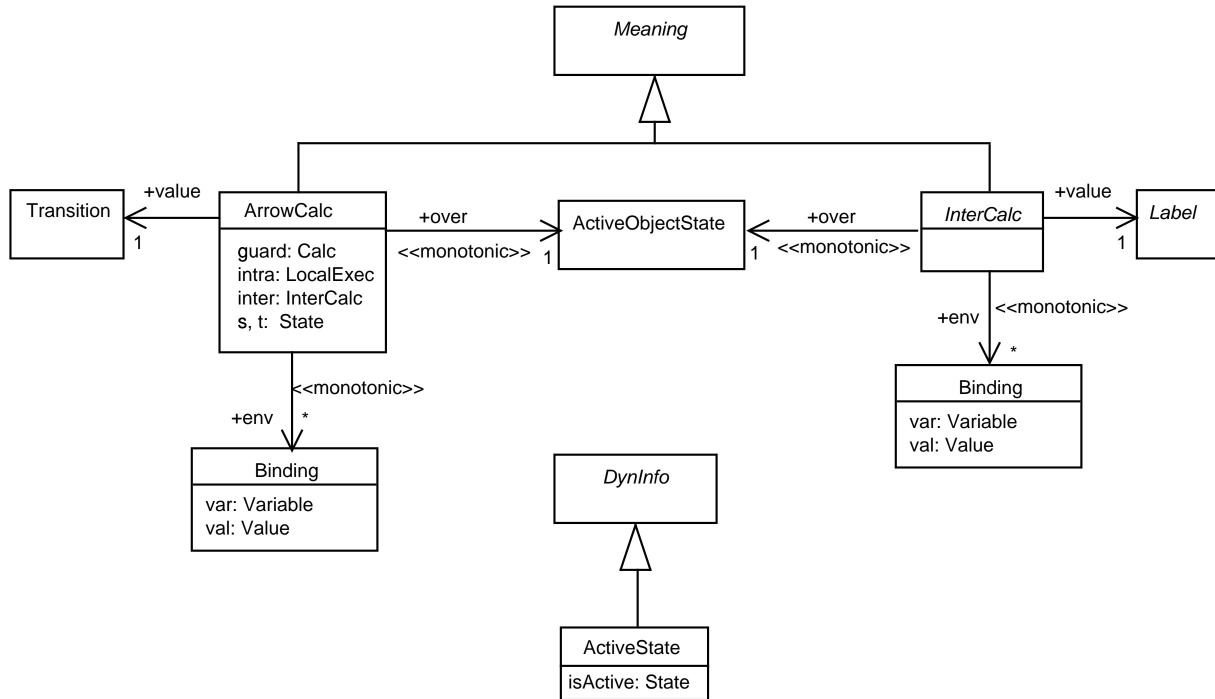


Figure 20: LTD.semantic domains.concepts Package

5.2 Semantic Domains

We report this package in Fig. 20.

To give the semantics of an LTD we need to refine the LTS, defined in Sect. 3.2, by refining the abstract class dynamic information. In this case, a dynamic information is just the (control) state of the LTD that is active at such moment (exactly one)[2].

The meaning of an arrow, an arrow calculation, has been obtained by instantiating the **calculation** pattern; analogously the interaction calculations (giving the meaning of the interactions) have been obtained by instantiating the same pattern.

Well-formedness rules The <<monotonic>> stereotype on the association over of ArrowCalc requires that the state over which a transition is evaluated is the same for its guard, local action and interaction. Whereas the same stereotype on the association env guarantees that the environment of an arrow calculation extends those of its subcomponents.

- The transition TR associated by value with an arrow calculation is such that:
  - the guard evaluated over the source state of TR is true,
  - the source and target states of TR are those defined by the intra attribute (a local execution),

[2]Using the statemachine terminology, an LTD has exactly one hierarchical sequential state.

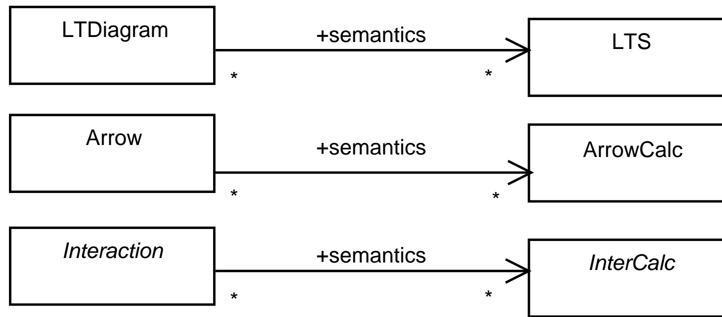


Figure 21: LTD.semantics Package

- the (control) state of the source of TR is the value of the s attribute of the arrow interpretation,
- the (control) state of the target of TR is the value of the t attribute of the arrow interpretation,
- the label of TR is that defined by the inter attribute (an interaction calculation),

```
context ArrowCalculation inv:
     self.guard.over =  self.value.source and
     self.guard.value = True and
     self.value.source = self.intra.source and
     self.value.target = self.intra.target and
     self.value.source.isActive = self.s and
     self.value.target.isActive = self.t and
     self.value.label = self.inter.value
```

5.3 Semantics

We report this package in Fig. 21.

- The semantic interpretations of an LTD are labelled transition systems (instances of class LTS), where dynamic information have been refined as ActiveState, and labels, instead, are still abstract.
- The semantic interpretations of an arrow are transition calculations (instances of class TransitionCalc).
- The semantic interpretations of an interaction are interaction calculations (instances of class InterCalc) whose values are labels. Whenever the interactions will be refined consequently, labels will be coherently refined; notice that this requires also a coherent refinement of the communication part of the context active classes.
- There are no semantic interpretations of State (the control states), they are just syntactic elements interpreted by themselves.

Methods

- Derived method for class LTS

  `name: -> string` defined by `name(lts) = lts.statesOf -> collect s | s.name`

  It is well defined, because all states of an lts have the same name (identity).

Well-formedness rules

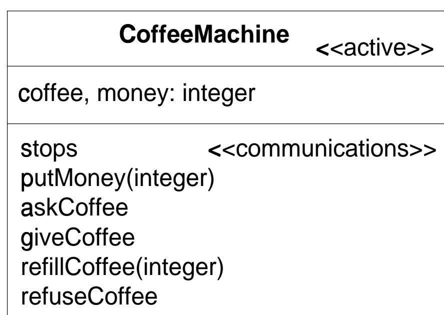
- For each labelled transition system LTS belonging to the semantics of an LTD, whose context is the active class C, the transitions of LTS are the semantic interpretations of all the arrows of LTD with the correct identity (given by the name attribute).

```
context LTDiagram inv:
     self.semantics -> forall{ lts |
          ( lts.transitions =
               (self.arrows.semantics -> select{tr | tr.target.name = name(lts)}))}
```

5.4 Examples of LTD

Here we present an example of an LTD associated with an active class, using a concrete syntax similar to that of UML.

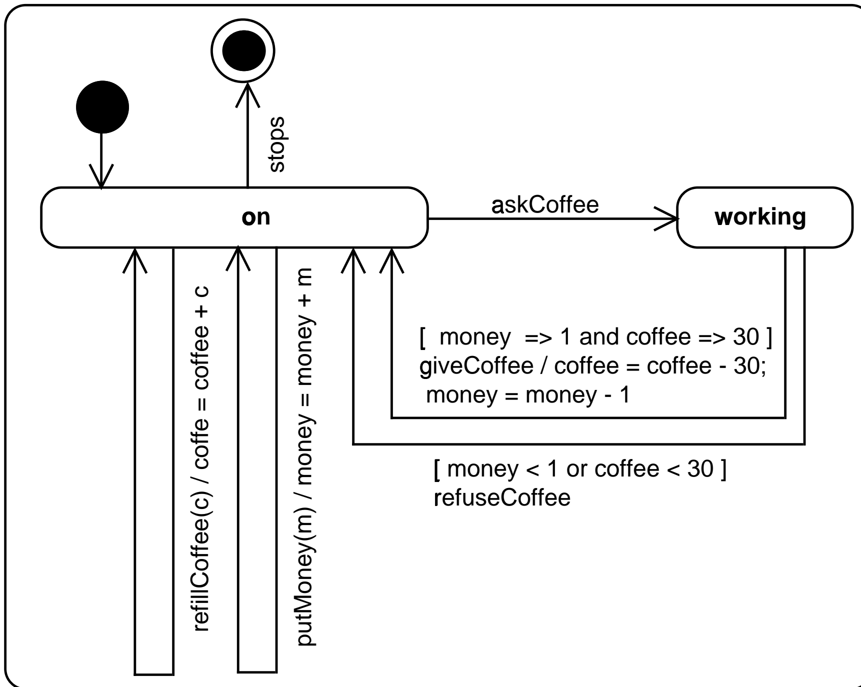
The active class CoffeeMachine has two attributes (coffee and money) and six different communication means, characterized by a name and 0 or more parameters (stops has no parameter whereas putMoney has a parameter of integer type).



The LTD diagram is reported below. The concrete syntax is similar to that of the state machines. In this case there are four states (an initial one, a final one and two simple ones). An arrow is depicted by putting over the arrow line

[guard] inter / intra

where an interaction has the form communicationId(exp1,...,expn).

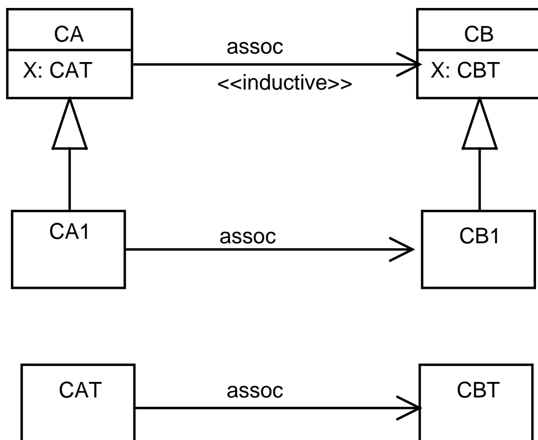

stops
on
askCoffee
working
refillCoffee(c) / coffe = coffee + c
putMoney(m) / money = money + m
[money => 1 and coffee => 30]
giveCoffee / coffee = coffee - 30;
money = money - 1
[money < 1 or coffee < 30]
refuseCoffee

A Used Stereotyps and Patterns

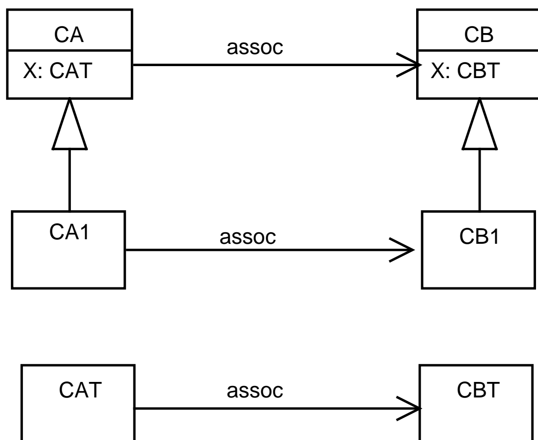
A.1 <<**inductive**>> (stereotype)

The stereotype <<inductive>> put on an association assoc from a class CA to a class CB (CB may be also equal to CA) means that:

- whenever there exists an association assoc from a specialization CA1 of CA into a specialization CB1 of CB, then for each instance A of CA that admits type CA1 the elements associated by assoc to A admits the type CB1 and vice versa;
- whenever CA has an attribute X of type CAT and CB has an attribute X of type CBT, and there exists an association assoc from CAT into CBT, then if an instance A of CA is linked by assoc with an instance B of CB, then the value of the attribute X of A is linked by assoc to the value of the attribute X of B.



is equivalent to



plus the following constraints

- If an element of CA admits the type CA1, then it is asociated by assoc to elements of CB admitting the type CB1.

```
context CA inv:
     CA1.isTypeOf(self)  implies
          (self.assoc -> forall{ b | CB1.isTypeOf(b)})
```

- If an element of CA is associated by assoc to an element of CB that admits the type CB1, then it admits the type CA1.

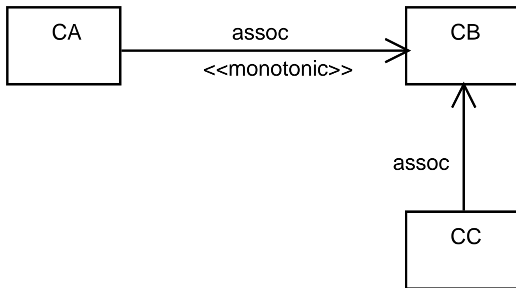
```
context CB inv:
     self.assoc -> select{ a | CA1.isType(a)} -> size > 0  implies
          CB1.isType(self)
```

- If an element A of class CA is associated by assoc to an element B of class CB, then the value of its attribute X is associated by assoc to the value of the attribute X of B.

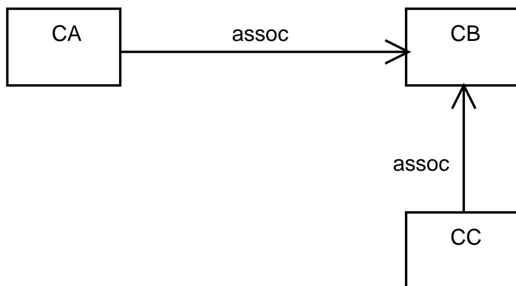
```
context CA inv:
     self.assoc -> forall{ b | self.X.assoc -> includes b.X }
```

A.2 <<monotonic>> (stereotype)

The stereotype <<monotonic>> put on an association assoc from a class CA, that is a specialization of Container, to a class CB (CB may be also equal to CA) means that whenver an instance A of CA has a subelement C (i.e., C belongs to A.elements) whose class CC (it may be also equal to CA) has an association assoc to class CB, then C.assoc is a subset of A.assoc.



where CA is a specialization of the class Container, is equivalent to

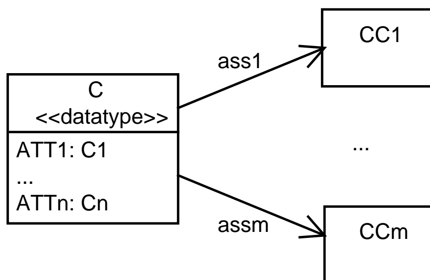


plus the following constraint

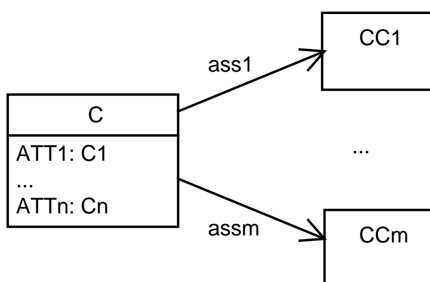
```
context CA inv:
     self.elements ->
          forall{ C | C.isType(CC) implies  self.assoc->includesAll(C.assoc) }
}
```

A.3 <<datatype>> (stereotype)

The stereotype <<datatype>> put on a class C means that whenver two instances A and B of C cannot be distinguished by using the attributes, the associations leaving C then they musy coincide.



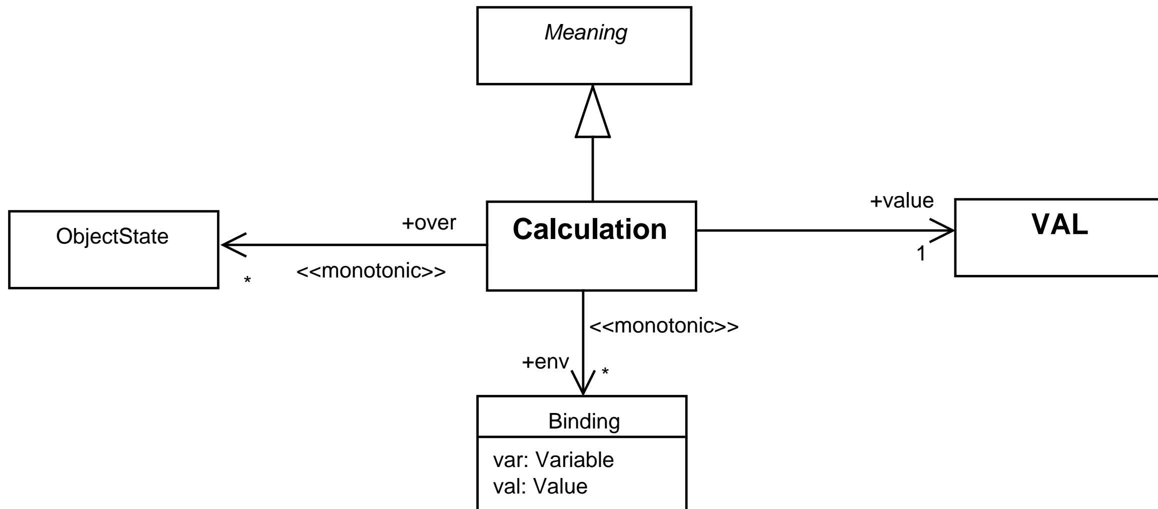
is equivalent to



plus the following constraint

```
context C inv:
     C.allInstances ->forall{ c |
          c.ATT1 = self.ATT1 and
            ... and
          c.ATTn = self.ATTn and
          c.ass1 = self.ass1 and
          ... and
          c.assm = self.assm implies  c = self }}
```

A.4 *calculation* (package)



In the above picture we have highlighted the parts that can be freely instantiated when using this pattern (**Calculation** and **VAL**); for example in Fig. 10 **Calculation** has been replaced by *Calc* and **VAL** by Value.

References

[1] T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach - Version 1.0. (September 2000). Available at `http://www.cs.york.ac.uk/puml/mmf.pdf.`, 2000.

[2] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[3] P. Stevens. On Use Cases and Their Relationships in the Unified Modelling Language. In H. Hussmann, editor, *Proc. FASE 2001 - Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.