

Remarks on “A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach”

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

20 Novembr 2000

A Remarks and questions

Calc (Fig. 16 and 18) If we have understood correctly – and we couldn’t find any other consistent interpretation – in your view a **Calc** is a pair in the graph of the semantic function associated with an expression, the one that associates a value with a binding (which includes the states of the objects needed for the evaluation, indirectly via the value of **self** and “navigation”). Thus in Fig. 16 the multiplicity of the **env** role should be 1 and in Fig. 18 the association between **Exp** and **Calc** should bear the * multiplicity also at the **Calc** end – in other word you associate with an expression its semantic function graph.

Instances Instance, counterpart of model, is one of the 4 key terms you use. We find the term **lstance** confusing, for a number of reasons:

- first, it does not correspond to the usual OO terminology; consider for example a class model: its instantiations are the objects, while for you they are the object configurations/states (at any instant time);
- second, after a careful scrutiny, it seems to us that in your proposal **Instance** corresponds to what in traditional semantics can be called Semantic domain (not just a set; it can be structured). Here, since you want to use a terminology more palatable for the non-expert, you could call them *Meaning* (the domain of meanings). Consider the case of expressions; with this interpretation everything comes out nicely: the semantic meaning at the abstract level (meaning.concepts) is a mathematical function, i.e., a set of pairs (argument,value); its syntactic counterpart (meaning.syntax) can comprise many different syntactic descriptions of the semantic function, as a
 - lambda calculus definition (see original denotational semantics) $\lambda bind.vexp$, where $vexp$ is the description of the value of expression associated with a binding $bind$
 - usual functional notation $f(bind) = vexp$,

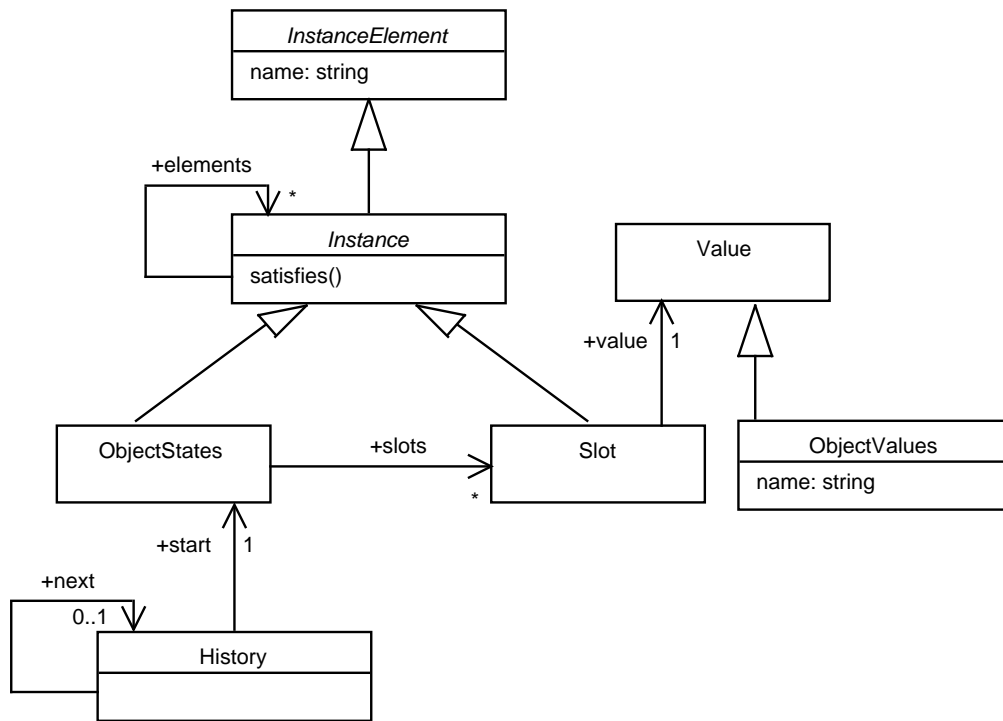


Figure 1: core.instance.concepts Package

- some graphical description,
- etc..

Of course also Model, here used for syntactic elements, is a bit confusing, for people more on the logic/semantic side; but we understand that you/we are bound to its use in the UML hierarchy. So we are not worried, either.

Attribute name for Instances Which is the reason for imposing names for all instances? for example in this way Calc get names; the same for data values; for what? While it is sensible to have names for Object (states). Of course one can impose uniqueness of names going down in the hierarchy, but would'nt be simpler to introduce them when needed at the appropriate level, for example for Object (states)?

Well-formedness constraints We have a number of remarks on missing rules, but we postpone a detailed account of these to your reaction about the proposed changes which follow.

B Proposed modifications to your packages

B.1 core.instance.concepts package

See Fig. 1.

Essentially we propose one main modification with two aspects

- to introduce explicitly a class of values for data and object values
- moreover to define object values not as object state/configurations (in your terminology, the elements of the class `Object`) but as object identities.

The motivation is twofold:

- first, using object identities instead of object states as values makes extremely simpler 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 objects not only looks more natural but avoids some puzzling things, like having many “zero values” with different names (see remark above); notice that UML explicitly states that data values are not objects, so they are without identity, which would imply well-formedness rules for `datatypes.instance.concepts` guaranteeing uniqueness of values, even in the case we avoid names at the instance level.

B.2 Meaning of a class

Moreover, as illustrated in Fig. 1, we think that the object states (objects in your terminology) are not sufficient to give the semantics of a class; indeed, the elements of a class are objects considered in their full lives. Let us consider for a moment nonactive classes. Then, the possible lives of an object may be represented by a set of histories (where a history is a sequence of states with the same value of the `name` attribute). Thus in Fig. 1 we need to introduce a class `History`, and the semantics package is modified as in Fig. 2. For active classes, as in our proposal, we think that labelled transition systems are a good choice for their semantics.

B.3

Then some other packages have to be slightly modified accordingly, including constraints (what we have done).

B.4 `constraints.instance.concepts`

See Fig. 3

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 states (for getting the current attribute values); this is usual in the classical (denotational and operational) semantics. According to the previous discussion, now a `Calc` would be a triple `<binding,states,value>`.

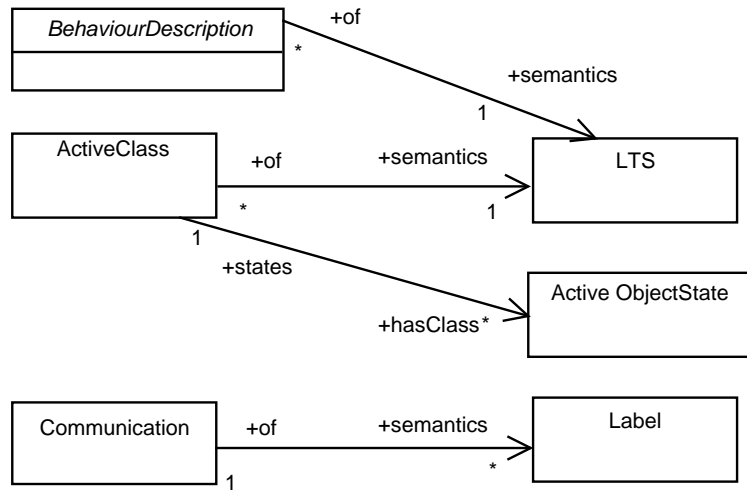


Figure 2: Dynamic Core Semantics

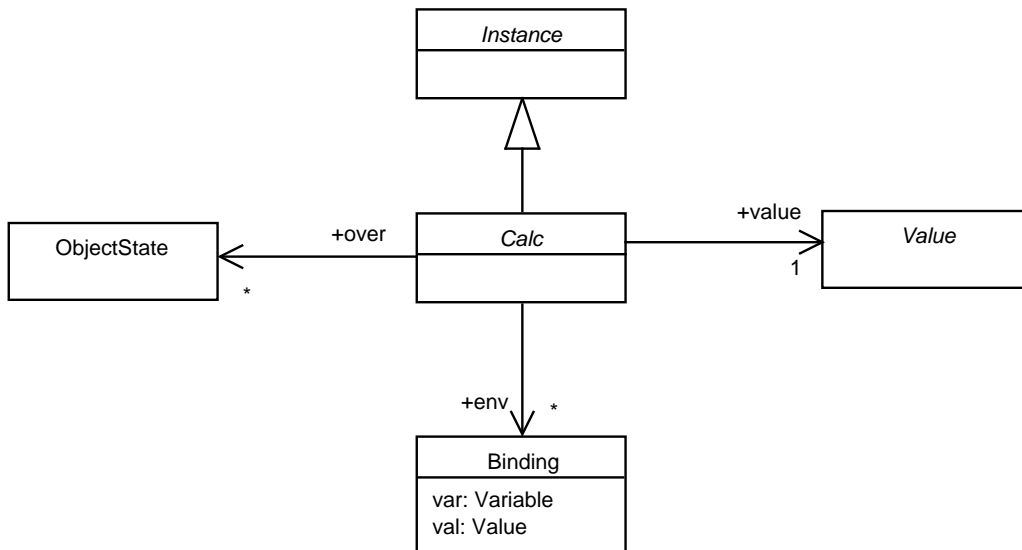


Figure 3: constraints.instance.concepts Package