Model Semantics and Mathematical Logic

Ed Seidewitz / Model Driven Solutions
August 2008
This research/publication was made possible by a grant from Lockheed-Martin Corporation.

Clause 10 of the *Semantics of a Foundational Subset for Executable UML Models* specification (referred to in the following as simply the "fUML Specification") defines the *base semantics* for Foundational UML (fUML). This base semantics is actually provided for the further subset of Base UML (bUML) that is used in writing the execution model that provides the operational semantics for all of fUML (see Clause 8 of the fUML Specification). The specification approach is based on using *first order logic* to specify the relationship between UML syntactic elements and instances in the semantic domain.

In extending the fUML semantics to cover SysML, it will be important to ensure that the formal base semantics continues to provide sufficient grounding to cover the semantic extensions. It is therefore relevant to further explore and explicate the semantic grounding of the current fUML Specification. This was the topic of discussion at the Executable UML/SysML Semantics Project Meeting held on 01 August 2008.

This document includes a description of the base semantic approach used in the fUML Specification and its relation to traditional concepts of mathematical logic, based on the discussion held at the August project meeting. This can be considered to be an elaboration of description of background concepts in Clause 6 of the fUML Specification.

Most of the semantic specification in Clause 10 of the fUML Specification has to do with the semantics of behavioral modeling, particularly activities. However, the fundamental concepts involved in the use of mathematical logic for semantic specification can be largely understood by just considering the simpler semantics of structural modeling. Therefore, for simplicity of presentation, the discussion here will focus only on structural semantics.

1 Modeling

Consider the simple class model shown in Figure 1. This is intended to be interpreted as allowing certain statements to be made about the domain being modeled: "A person has a name." "A person owns houses." And so on.

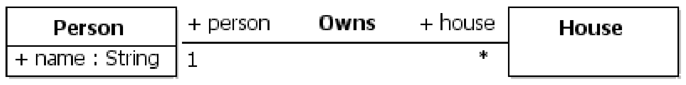

Figure 1 Simple Class Model

These informal English statements can be formalized using mathematical logic. First, we need to define a set of *predicates* for the basic terms specified in the model. These include:

- Person(*p*) means "*p* is a Person".
- name(*p*,*n*) means "*p* has the name *n*".
- House(*h*) means "*h* is a house".

- Owns(p,h) means "p owns h".[1]

We can then formally interpret statements made by the model in Figure 1 using the typical notation of mathematical logic:

- Property *name* is an attribute of class *Person*: "A name is for a person."
  $\forall p(\mathrm{name}(p,n) \rightarrow \mathrm{Person}(p))$
- Property *name* has type *String*: "The name of a person is a string."
  $\forall p \forall n(\mathrm{name}(p,n) \rightarrow \mathrm{String}(n))$[2]
- The multiplicity of *Person::name* is 1..1: "Every person has exactly one name."
  $\forall p(\mathrm{Person}(p) \rightarrow \exists \mathrm{n}(\mathrm{name}(p,n) \wedge \forall m(n \neq m \rightarrow \neg \mathrm{name}(p,m))))$
- The association *Owns* has end types *Person* and *House*: "Persons own houses".
  $\forall p \forall h(\mathrm{Owns}(p,h) \rightarrow \mathrm{Person}(p) \wedge \mathrm{House}(h))$
- The multiplicity of *Owns::house* is 1..1: "Every house has exactly one owner."
  $\forall h(\mathrm{House}(h) \rightarrow \exists \mathrm{p}(\mathrm{Owns}(p,h) \wedge \forall q(p \neq q \rightarrow \neg \mathrm{Owns}(q,h))))$

The fUML Specification does not actually use the traditional mathematical logic symbology used above. Instead, it expresses logical statements using the Common Logic Interchange Format (CLIF).[3] In this machine-readable format, atomic predicate formulas such as "Person(p)" are written in the form "`(Person p)`". The above statements are then written in CLIF as:

(1) "A name is for a person."
```
(forall (p b)
    (if (name p n) (Person p)))
```

(2) "The name of a person is a string."
```
(forall (p n)
    (if (and (Person p) (name p n)) (String n)))
```

(3) "Every person has exactly one name."
```
(forall (p)
    (if (Person p)
        (exists (n)
            (and (name p n)
                 (forall (m) (if (not (= n m)) (not (name p m)))))))
```

(4) "Persons own houses."
```
(forall (p h)
    (if (Owns p h) (and (Person p) (House h))))
```

[1] Note that we use a two place predicate to represent an association (e.g., "Owns(p,h)"), rather than using two separate predicates for the association end properties (e.g., "person(p,h)" and "house(h,p)"). Even though Clause 10 of the fUML specification uses the latter formalization in terms of association ends, treating them as multi-argument predicates is more consistent with normal practice in logic and simpler for our purposes here. And it does not affect the relevance of the points being made here to the actual approach in the fUML specification.

[2] We are assuming here that the primitive concept of a "String" is pre-defined and that "String(s)" means "s is a String".

[3] See ISO 24707, http://standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007(E).zip.

(5) “Every house has exactly one owner.”

```
(forall (h)
    (if (House h)
        (exists (p)
            (and (Owns p h))
                 (forall (q) (if (not (= p q)) (not (Owns q h)))))))
```

Now consider the model in Figure 2 of instances of classes from Figure 1. This model also makes statements about the domain being modeled, but it now makes statements about *specific things* in that domain. The set of such things is what logicians call the *universe of discourse*—in this case, people and houses.

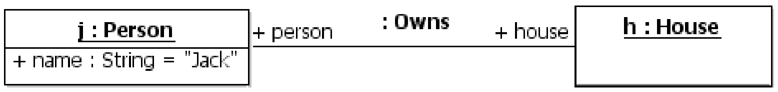

Figure 2 A Simple Instance Model

Just as we could formally interpret that statements made by the class model of Figure 1, we can make formal interpretations of the statements made by the instance model. Such statements have the form of *assertions* about specific elements of the universe of discourse—in this case, about the person labeled *j* in Figure 2 and the house labeled *h.* An assertion is a statement that is claimed to be true for the universe of discourse.

The assertions made by the instance model in Figure 2 include the following:

(6) “*j* is a person.”
`(Person j)`

(7) “The name of *j* is Jack.”
`(name j “Jack”)`[4]

(8) “*h* is a house.”
`(House h)`

(9) “*j* owns *h.*”
`(Owns j h)`

2 Deduction, Syntax and Semantics

The primary advantage of formalizing the statement made by a model as logical propositions is that the rules of logic may be used to *deduce* new statements from statements that are already known. Such deductions are made according to *deduction rules* that are carefully formulated so that any deduced statements are true as long as all the statements they are deduced from are true.

Some common deduction rules are:

(10) Given `(and P Q)`, deduce `P` and `Q`.

(11) Given `(if P Q)` and `P`, deduce `Q`.

(12) Given `(if (and P1 P2) Q)` and `P1`, deduce `(if P2 Q)`.

[4] Note that we are assuming here that there is a built-in notation for strings like `“Jack”`.

(13) Given `(forall (x) (P x))`, deduce `(P y)` for any `y`.

As an example deduction, consider that, given statement (5) from the class model and statement (9) from the instance model, we can use rules (13) and (11) to deduce:

(14) "No one other than *j* owns *h*."
`(forall (q) (if (not (= j q)) (not (Owns q h))`

Now, suppose further that we know of another person *k* who is not *j*:

(15) `(Person k)`

(16) `(not (= j k))`

Then, using rules (13) and (11) again, we can deduce `(not (Owns b h))`.

Deduction can be used not only to derive new true statements, but also to check on the consistency of a set of assertions. For example, from statements (4) and (9) we can deduce `(Person j)` and `(House h)`, using rules (10), (11) and (13). This is consistent with assertions (6) and (8) already made by the instance model.

In the parlance of logic, what we are doing here is taking the statements made by a model, as formalized in mathematical logic, to be a set of *axioms*—that is, a set of statements all asserted to be true together. The (generally infinite) set of all possible deductions that can be made from these axioms using some set of deduction rules is known as the *theory* derived from the axioms.[5] A *proof* of a statement in the theory is a series of deductions leading from the axioms to the statement. A theory is *consistent* if there is no statement `S` in the theory for which the logical inverse of that statement `(not S)` is also in the theory (that is, it is not possible to prove both the statement and its inverse in the theory).

In *first order* logic, statements are written in terms of predicates over some universe of discourse and all quantification ("`forall`" and "`exists`") is over this universe. In Section 1 we interpreted UML class and instance models in terms of such first order logic, in which each model implied certain predicates and certain axiomatic statements in terms of those predicates.

However, there are some general statements we would like to make about models such as those discussed in Section 1 that cannot be handled by our formal language so far. For example, in general in UML, if a property has a certain type, then all values of that property must be of the given type. Since Figure 1 shows that the property *Person::name* has the type *String,* we would like to be able to *deduce,* from the general rules of UML, that any value of *name* must be a string.

In order to formalize such general rules, we need to make statements about elements of the models themselves, such as classes like *Person* and properties like *name.* Since we interpreted these elements as logical predicates, we thus need to make formal statements about *predicates.* A logic language that allows formal statements to be made about first order predicates is known as *second order.*

For example, we can define the following second order predicates, whose arguments are first order predicates over the original universe of discourse:

[5] Clause 6 of the fUML Specification defines a "theory" as the set of deduction rules used, rather than the statements deduced using those rules. This is closer to the way the term *theory* is used outside of mathematical logic, but the conventional definition within mathematical logic is the one given here.

(17) "*C* has the owned attribute *A*."
```
(ownedAttribute C A)
```

(18) "*P* has the type *T*."
```
(type P T)
```

We can now state the general axioms:

(19) "If *A* is an owned attribute of *C* and *y* is a value of *A* for instance *x,* then *x* is an instance of *C*."
```
(forall (C A x y)
    (if (and (ownedAttribute C A) (A x y)) (C x)))
```

(20) "If *A* is an owned attribute of *C*, *A* has type *T, x* is an instance of *C* and *y* is a value of *A* for *x,* then *x* is an instance of *T*."
```
(forall (C A T x y)
    (if (and (ownedAttribute C A) (type A T))
             (C x) (A x y))
        (T y)))
```

If we then assert

(21) `(ownedAttribute Person name)`

(22) `(type name String)`

we can *deduce*

(23) `(forall (x y) (if (name x y) (Person x)))`

(24) `(forall (x y) (if (and (Person x) (name x y)) (String y)))`

which we previously simply *asserted* for the model as (1) and (2).

Now, second order languages are generally more difficult to deal with than first order languages, and automated provers, for example, generally only handle first order languages. To deal with this, logicians often recast a second order language into an equivalent first order form.[6] To do this, we extend our universe of discourse with new elements to represent what were formerly the first order predicates of the language. What were formerly second order predicates then become first order predicates over these new elements.

For the purposes of UML modeling, we need to add a set of *class* elements to represent classes, a set of *property* elements to represent properties, etc. We then define new predicates to distinguish these elements from the instances that were already in the universe:

(25) "*c* is a class."
```
(Class c)
```

(26) "*t* is a primitive type."
```
(PrimitiveType t)
```

(27) "*p* is a property."
```
(Property p)
```

[6] Technically, it can be shown that the direct or *absolute* semantics for a second order language cannot be fully duplicated by any first order language. However, there is an *alternate* semantics for second order languages that is essentially first order and which is equivalent for our purposes here. (See, e.g., Herbert B. Enderton, *A Mathematical Introduction to Logic, Second Edition,* Academic Press, 2001, Chapter 4.)

(28) “*a* is an association”
```
(Association a)
```

Thus, for the sample model of Figure 1, we have

(29) `(Class Person)`

(30) `(Class House)`

(31) `(PrimitiveType String)`

(32) `(Property name)`

(33) `(Association Owns)`

Finally, we need to define some additional predicates to be used in place of applying the old first-order predicates:

(34) “*c* classifies *v*.”
```
(classifies c v)
```

(35) “*p* has value *v* for *x*.”
```
(property-value x p v)
```

(36) “*a* links *x* and *y*.”
```
(link x a y)
```

for which the following basic axioms hold

(37) `(forall (p) (if (PrimitiveType p) (Classifier p)))`

(38) `(forall (c) (if (Class c) (Classifier c))`

(39) `(forall (c v) (if (classifies c v) (Classifier c)))`

(40) `(forall (x p v) (if (property-value x p v) (Property p)))`

(41) `(forall (x a y) (if (link x a y) (Association a))`

Thus, given (29) through (33), instead of `(Person p)`, `(name p n)` and `(Owns p h)`, we can now write `(classifies Person p)`, `(property-value p name n)` and `(link p Owns h)`. And this, in turn, lets us rewrite general rules such as (19) and (20) in a first order form:

(42)
```
(forall (c p x y)
    (if (and (ownedAttribute c p) (property-value x p y))
        (classifies c x)))
```

(43)
```
(forall (c p t x y)
    (if (and (ownedAttribute c p) (type p t)
             (classifies c x) (property-value x p y))
        (classifies t y)))
```

Then, given (21) and (22), we can make the equivalent deductions to (23) and (24):

(44) `(forall (x y) (if (property-value x name y) (classifies Person x)))`

(45)
```
(forall (x y)
    (if (and (classifies Person p) (property-value name p n))
        (classifies String n)))
```

Finally, rather than interpreting the instance model in Figure 2 in terms of the assertions (6) through (9), we have, instead, the equivalent assertions:

(46) `(classifies Person j)`

(47) `(property-value j name "Jack")`

(48) `(classifies House h)`

(49) `(link j Owns h)`

Then, given (44) and (45) (and other model-specific facts deduced from the general rules of UML), we can make deductions about the instance model, as discussed at the beginning of this section.

We can summarize the results of the discussion above by organizing our logic language into three categories:

- *Syntax* – Predicates such as `Class`, `Property` and `ownedAttribute` represent the *syntax* of UML models. A UML model (such as the class model in Figure 1) can be interpreted as a set of assertions using these predicates (such as (29) through (33), (21) and (22)).
- *Semantics* – Predicates such as `classifies`, `property-value` and `link` provide the basis for the *semantics* of UML models. They define how elements of a UML model can be used to make statements about instances in the domain being modeled. General rules such as (42) and (43) then specify how the structure of the model further constrains the valid statements that can be made about the domain.
- *Theorems* – From the general semantic rules, together with the syntactic assertions for a specific UML model, one can deduce model-specific *theorems* such as (44) and (45). These theorems, along with domain-specific assertions such as (46) through (49), can then be used to make specific deductions about the domain being modeled.

The abstract syntax of UML is, of course, determined by its specification (which we consider in a bit more detail below). It is the semantics of the bUML subset of UML, using essentially the approach discussed here, that is provided by Clause 10 of the fUML Specification.

3 Metamodeling

Consider the fragment of the UML abstract syntax model shown in Figure 3, which is sufficient to cover the syntax used in Figure 1. Since this is just a UML model, we can interpret it as a set of logical assertions, as discussed in Section 1. The result is a set of predicates `Class`, `Association`, `Property`, etc., that are exactly the syntactic predicates required in Section 2.

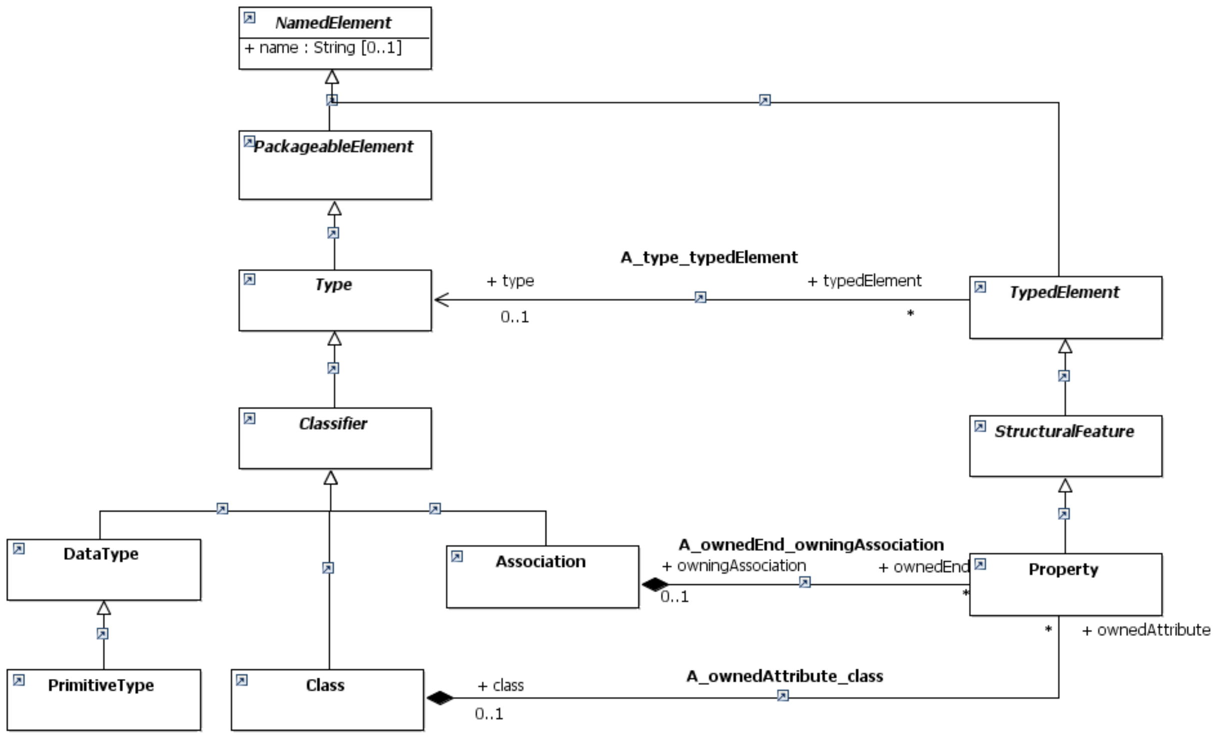

Figure 3 Classifier Abstract Syntax

In addition, the abstract syntax model places additional constraints that must be satisfied by any syntactically well-formed model. These include the following.

Generalizations:

(50) `(forall (r) (if (PackageableElement e) (NamedElement e)))`

(51) `(forall (t) (if (Type t) (PackageableElement t)))`

(52) `(forall (c) (if (Classifier c) (Type c)))`

(53) `(forall (c) (if (Class c) (Classifier c)))`

(54) `(forall (a) (if (Association a) (Classifier a)))`

(55) `(forall (d) (if (DataType d) (Classifier d)))`

(56) `(forall (p) (if (PrimitiveType p) (DataType p)))`

(57) `(forall (s) (if (StructuralFeature s) (TypedElement s)))`

(58) `(forall (p) (if (Property p) (StructuralFeature p)))`

Attributes:

(59) `(forall (n s) (if (name n s) (NamedElement n)))`

(60) `(forall (n s) (if (name n s) (String s)))`

(61)
```
(forall (n s1 s2)
    (if (and (name n s1) (not (= s1 s2))) (not (name n s2))))
```

Associations:

(62)
```
(forall (p c)
    (if (A_ownedAttribute_class p c) (and (Property p) (Class c))))
```

(63)
```
(forall (p c1 c2)
    (if (and (A_ownedAttribute_class p c1) (not (= c1 c2)))
        (not (A_ownedAttribute_class p c2))))
```

(64)
```
(forall (p a)
    (if (A_ownedEnd_owningAssociation p a)
        (and (Property p) (Association a))))
```

(65)
```
(forall (p a1 a2)
    (if (and (A_ownedEnd_owningAssociation p a1) (not (= a1 a2)))
        (not (A_ownedEnd_owningAssociation p a2))))
```

(66)
```
(forall (t e)
    (if (A_type_typedElement t e) (and (Type t) (TypedElement e))))
```

(67)
```
(forall (t1 t2 e)
    (if (and (A_type_typedElement t1 e) (not (= t1 t2)))
        (not (A_type_typedElement t2 e))))
```

For example, Figure 4 gives the abstract syntax representation of the simple model from Figure 1. This corresponds to the logical assertions:

(68) `(PrimitiveType s)`

(69) `(name s ”String”)`

(70) `(Property p)`

(71) `(name p ”name”)`

(72) `(A_type_typedElement s p)`

(73) `(Class c1)`

(74) `(name c1 ”Person”)`

(75) `(A_ownedAttribute_class p c1)`

(76) `(Property e1)`

(77) `(name e1 ”owner”)`

(78) `(A_type_typedElement c1 e1)`

(79) `(Class c2)`

(80) `(name c2 ”House”)`

(81) `(Property e2)`

(82) `(name e2 ”houses”)`

(83) `(A_type_TypedElement c2 e2)`

(84) `(Association a)`

(85) `(name a ”Owns”)`

(86) `(A_ownedEnd_owningAssociation e1 a)`

(87) `(A_ownedEnd_owningAssociation e2 a)`

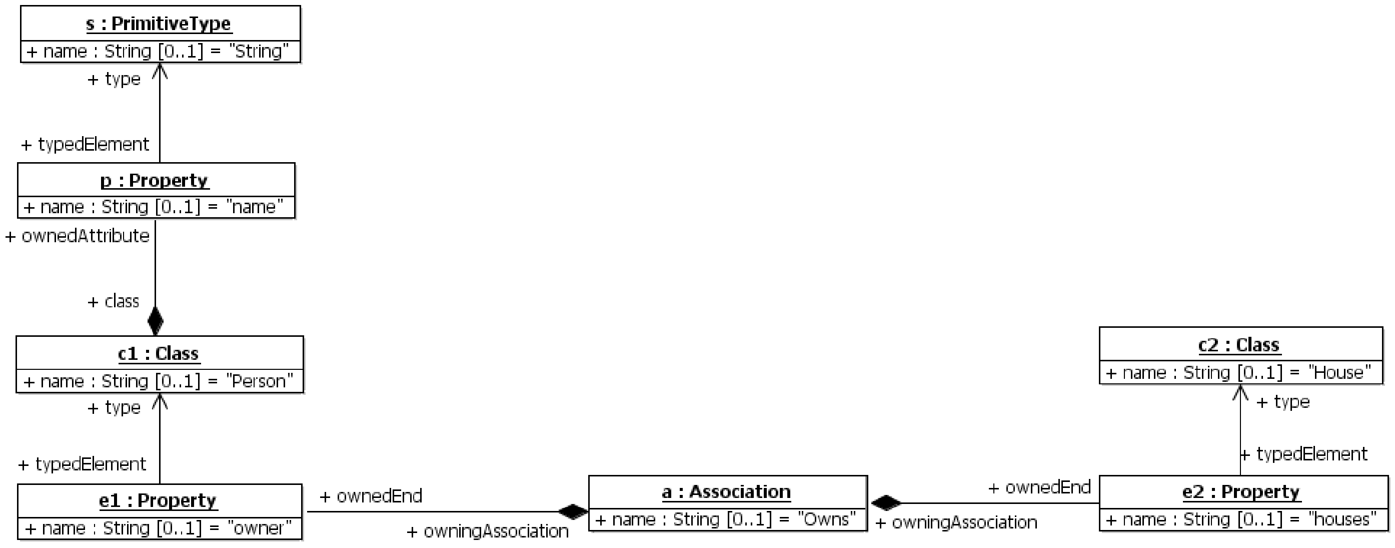

Figure 4 The Abstract Syntax Representation of the Model of Figure 1

The bUML semantics specified in Clause 10 of the fUML Specification presumes that a model has been represented in a logical form similar to this. Further, while the clause does not state these conditions explicitly, it is assumed that any model being interpreted satisfies all the logical well-formedness conditions derivable from the abstract syntax subset for bUML (such as (50) through (67)).

Now, there is, of course, also an abstract syntax for UML instance models. For example, the abstract syntax fragment given in Figure 5 is sufficient to cover the syntax used in the simple model of Figure 2. Interpreting this model leads to logical predicates such as `(A_classifier_instanceSpecification c i)`, `(A_value_owningSlot v s)` and `(A_slot_owningInstance s i)`. However, It is important to distinguish these from the seemingly similar predicates `(classifies c v)` and `(property-value x p v)`.

The statement `(A_classifier_instanceSpecification c i)` is an assertion of a *syntactic* relationship between two model elements: a classifier *c* and an instance specification *i.* On the other hand, `(classifies c v)` is an assertion of a *semantic* relationship between the classifier model element *c* and an actual instance *v* in the universe of discourse, giving meaning to what a “classifier” *is*. In terms of the OMG hierarchy of four meta-layers, `A_classifier_instanceSpecification` is a relation between two elements in layer M1 (*c* and *i*), while `classifies` relates an element in M1 (*c*) to an actual instance in M0 (*v*).

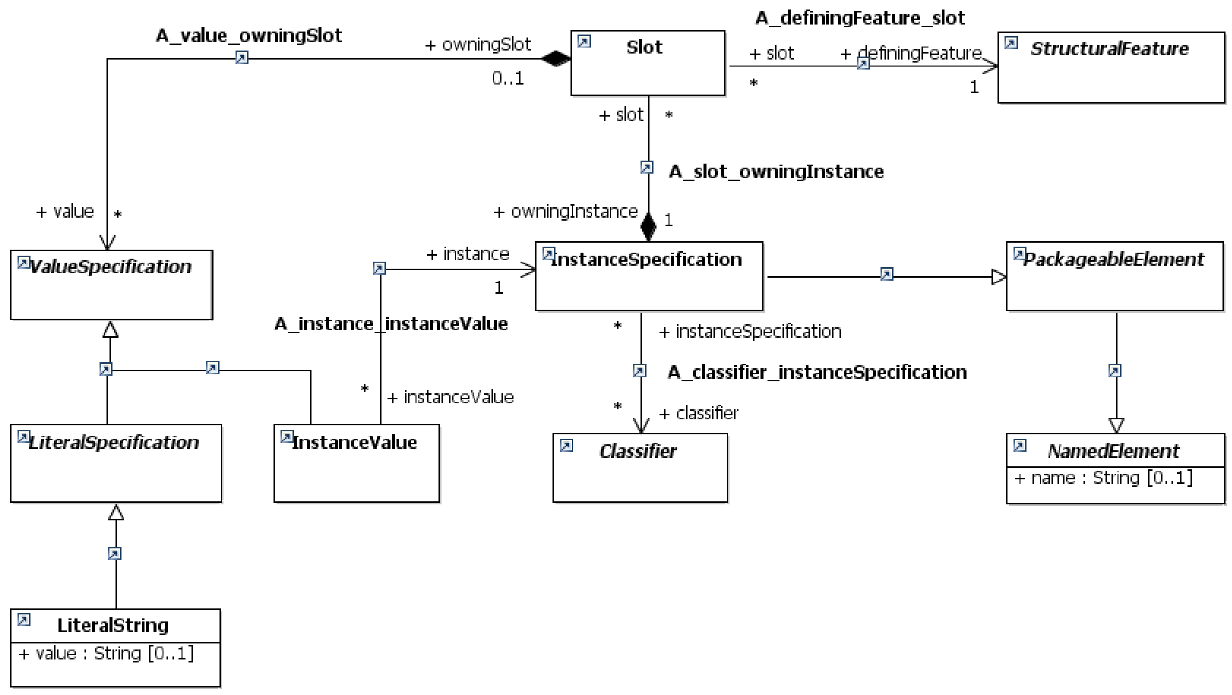

Figure 5 Value Specification Abstract Syntax

In order to express the semantics of instance models thus requires additional predicates to relate instance model elements to actual instances at M0. These clearly should be consistent with the predicates `classifies` and `property-value`. For example, suppose we introduce the predicates

(88) `(specifies-instance i x)` means "Instance specification *i* specifies instance *x*."

(89) `(specifies-value v x)` means "Value specification *v* specifies value *x*."

Then we would expect to have semantic relations such as:

(90) "The instance specified by an instance specification is classified by the classifier of the instance specification."

```
(forall (c i x)
    (if (and (A_classifier_instanceSpecification c i)
             (specifies-instance i x))
        (classifies c x)))
```

(91) "The value specified by a slot is the value of the property given by the defining feature of the slot, for the instance specified by the instance specification that owns the slot."

```
(forall (i s v p x y)
    (if (and (A_slot_owningInstance s i)
             (A_value_owningSlot v s)
             (A_definingFeature_slot p s)
             (specifies-instance i x)
             (specifies-value v y))
        (property-value x p y)))
```

(92) "The value specified by an instance value is the same as the instance specified by the instance specification of the instance value."

```
    (forall (v i x)
        (if (A_instance_InstanceValue i v)
            (iff (specifies_instance i x) (specifies_value v x))))
```

Of course, by obvious intent, there is a parallelism between the syntactic structure of instance specifications and the property-value structure of the instances being specified. Indeed, an alternate approach to defining the semantics for UML class models would be to specify the semantics of a class model in terms of the set of all possible instance *models* that conform to the class model, rather than in terms of sets of instances form some separate "problem domain."

In this alternate approach, one would, in fact, use syntactic relations such as `A_classifier_instanceSpecification` in the rules for class model semantics, rather than introducing new predicates such as `classifies`. Deduction rules would then, in effect, allow for the direct checking of the consistency of an instance model against a class model and for the derivation of new, valid instance models from instance models already shown to be consistent with the class model. This would be a formal "proof theoretic" approach to specifying class model semantics, entirely at the M1 layer.

This alternate approach is only possible, though, because UML provides an instance modeling syntax rich enough to express all the relations needed to define class model semantics. The approach does not generalize, however, to UML behavior modeling, because UML does not include a notation for expressing the state of an executing behavior. In order to define the semantics of behaviors it is therefore necessary to take the approach of relating the syntactic denotation of behavior (at M1) to a semantic domain of behavior execution (effectively at M0). Since the majority of Clause 10 of the fUML Specification has to do with behavioral semantics, which requires the approach of using a semantic domain, it is only reasonable to use a similar approach for the definition of structural semantics, which is what has been presented here.

4 Meta-metamodeling

Of course, we are now again left with a given set of predicates derived from UML abstract syntax metamodels such as Figure 3 and Figure 5. Following the same approach as in Sections 2 and 3, we can come up with a meta-metamodel for expressing the UML metamodel. Since the abstract syntax models are all class models, class modeling is all that needs to be covered by this meta-metamodel. Indeed, the OMG Meta Object Facility (MOF) meta-metamodel is based on a subset of the UML infrastructure whose key classes are essentially the same as those shown in Figure 3.[7]

Since the semantics of abstract syntax metamodeling is thus exactly that of UML structural semantics, as given by, e.g., the predicates `classifies` and `property-value`, as discussed in Section 2. Now, the MOF Core Specification actually does provide an "abstract semantics" for Complete MOF (CMOF).[8] This is given in terms of a "semantic domain" that is specified by a class model of instances, similar (though not identical) to the UML instance model abstract syntax given in Figure 5. However, this model really just completes the circularity of the MOF specification, because it presupposes that there is already some semantic grounding for interpreting the instance model itself—which is not provided in any formal way by the UML

[7] See *Meta Object Facility (MOF) Core Specification, Version 2.0,* January 2006, OMG document formal/06-01-01, Subclause 14.1.
[8] *Ibid*, Clause 15.

Infrastructure specification.[9] The formal specification for CMOF can be completed either by using the structural semantics of the CMOF subset of the UML abstract syntax or by providing instance semantics for the CMOF abstract semantic domain model (as outlined at the end of Section 3 for the UML instance model abstract syntax).

The subset of UML class modeling capabilities used to model the MOF abstract syntax is, essentially, the capabilities included in the MOF meta-metamodel itself. Therefore, it is not advantageous to add any additional meta-layers above that of MOF (i.e., M3). Providing the semantic grounding directly for the structural semantics of MOF then specifies the semantics for the abstract syntax models for all languages defined using MOF.

While MOF does not currently include any behavior modeling capabilities,[10] a similar approach can be taken as for abstract syntax, to define a small subset of UML behavior modeling capabilities that can reflexively model itself. This is essentially what has been done in the definition of the bUML subset in the fUML Specification. It is worth future consideration as to whether a similar subset could be incorporated into MOF to provide a common basis for defining the “abstract semantics” of MOF-based languages, as is now done for abstract syntax.

Until this is done, however, all behavioral semantics must be defined in terms of the specific language metamodel. This is what is done for bUML in Clause 10 of the fUML specification. However, because bUML must currently be a subset of UML superstructure abstract syntax (which is where all the behavioral modeling features are), not the infrastructure, it is not particularly useful to introduce the MOF level at all in its definition. Therefore, the semantic definition for bUML is specified directly in terms of UML M2 abstract syntax predicates, without any consideration of how that abstract syntax is defined in terms of the MOF M3 layer.

[9] *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.1.2,* November 2007, OMG document formal/07-11-04.

[10] CMOF does include the ability to model *operations* on metaclasses, but it provides no way to model the *behavior* of these operations. Thus, while it is possible to specify operation behavior using preconditions and postconditions (as used, e.g., in the UML abstract syntax models to specify additional operations for use in well-formedness constraints), it is not possible to operationally define this behavior.