Evaluation of the Unified Modeling Language

Submission to OOADTF RFP-1

Conrad Bock

Kevin Murphy

Amy Justice

(2/17/97)

1. Introduction

This evaluation covers UML semantics as described in the UML 1.0 semantics and notation documents, hereafter called the *semantics document* and the *notation document*. Since our focus is on problems in these, it should be said at the outset that we are supporters of the UML. The UML is based on some of the best methodologies and is aimed in the right direction for industry and the OMG. We hope the critiques in this document are taken in the light of improving an already important specification.

1.1 Overall evaluation and summary

Our overall evaluation is that the UML as it is currently documented has serious flaws that make it unacceptable as an OMG standard. Some flaws violate the most basic rules of object-orientation; others are the result of attempting advanced work just before submission without consulting prior art; some seem as if little effort was applied to the most basic concepts; others result from poor consistency checking within and across documents, some make it appear that the model was never tested on practical problems; while others are the result of violations of common sense in defining terms, to the point where quite a few are incomprehensible. A standard to be adopted by an entire industry must be free of these problems, both to be functional and preserve the reputation of the body issuing it.

This document contains a detailed technical analysis of the UML, drawing on our extensive experience in object-oriented modeling and implementation. Except in summary comments, we give reasons for our opinions, citing for each case relevant passages in the UML documents. The chapters parallel those in the semantics document, with the problems divided into four kinds for each chapter: modeling, comprehensibility, consistency, and clarity. They are partially summarized below.

Modeling concerns choices about the meta-model and its semantics. This covers issues of standard OO practice, parsimony, completeness, scope, and suitability for real world applications. For example:

The meta-type CLASS, which has implementations, is specialized from TYPE, which does not allow them. Similar contradictions arise in METHOD being specialized from OPERATION. This is a violation of basic subclassing semantics. See sections 5.1.1 and 8.1.1, pages 12 and 42.

In the last two revision cycles , the UML attempted to address the advanced notion of objects playing roles in an organizing structure or pattern. This hurried schedule might have worked had the authors consulted existing work on the subject. As it is, the resulting model cannot support its claim to handle objects playing different roles in various “composite objects” without the roles affecting each other. See section 6.1.2, page 21.

States and events are concepts fundamental to object-orientation but are so poorly defined in the UML that they are either incomprehensible or confused with completely different concepts. We propose definitions from existing work which should have been consulted prior to submission. State variables are better defined, but are confused with view issues, incomplete, and are not used in the definitions of state and event. See sections 10.1.1, 10.1.2, and 10.1.3, pages 45, 48, and 49.

Control-flow, a concept very popular in business modeling, is squeezed into the UML through a view on state machines, rather than understood as a modeling issue. Consequently a fundamental aspect of traditional control-flow diagrams is not handled and features of modern control-flow diagrams are omitted entirely. See section 10.1.6, page 53.

Associations are identified with classes in the text, but separated in the meta-model. This results in unnecessary additions to the meta-model because the notion of class is not properly reused. It also makes the common technique of specializing associations too cumbersome, and its importance to implementing roles obscure. See section 6.1.1, page 19.

Types as instances of other types is an important technique supported by the notation, but only weakly in the meta-model by sweeping it under the rug of dependency. TYPE should be specialized from INSTANCE to support types of types. See section 5.1.2, page 13.

Multiple classification of objects, a technique with useful applications and available in some object-oriented languages, is explicitly disallowed at one point in the documents and supported by an undefined extensibility construct in another. Specializing the existing meta-association for instantiation would be a simpler solution. See section 5.1.3, page 13.

Association navigation does not handle some common applications. We propose a navigation model that provides more flexibility. See section 6.1.4, page 31.

Stereotypes and tagged values are for extending the meta-model, but the point of object-oriented meta-models is that they are already extensible.

Users can add subtypes, associations, and so on, with a meta-object facility. In addition, stereotypes introduce unusual semantics despite the claim that they are the same as subtyping. See sections 3.1.1 and 3.1.2, pages 8 and 9.

A kind of aggregation called “shared aggregation” is defined that is identical to normal associations, we assume because users like it. The introduction of such a “placebo”, as Rumbaugh calls it, encourages modelers to define their own meaning, thereby defeating the purpose of the standard. See later part of section 6.1.3, page 27.

The meta-model explicitly disallows multi-polymorphism, that is, message dispatch on more than one type, an advanced technique available in some object-oriented languages. The solution, separating operations from types, has the additional benefit of supporting polymorphism without requiring a common supertype. See section 5.1.5, page 15.

Attributes are described as kind of association in the text, but not in the meta-model. See section 7.1.1, page 36.

The notion of qualifier for associations omits association and classification as qualifying aspects of objects. Simple examples are not handled. See section 6.1.5, page 32.

The ISTYPESCOPE attribute is defined at the programming language level, obscuring its purpose as a way of describing types as instances. A more general shorthand for such services is proposed. See section 7.1.2, page 36.

Operation inheritance is currently done by matching signatures, so it is impossible to unambiguously inherit two differently named operations on the same type with the same signature. See section 7.1.3, page 37.

In a couple of cases, non-determinism is introduced into state machines where it is not appropriate. See section 10.1.4, page 51.

Comprehensibility concerns topics where we could not gather the meaning at all. This is usually due to circular definitions, abstract terms not commonly used, or ideas that were not well thought out. We did not include in this category any terms for which the meaning could be pieced together, even if we had to search through all its uses to do so. Needless to say, concepts with indiscernible meaning cannot be standardized. Examples are:

ROLES meta-association between INSTANCE and TYPE (section 5.2.1, page 15).
Collaboration (section 9.1.1, page 43)
Responsibility (section 5.2.3, page 19).
ISCHANGEABLE meta-attribute on ASSOCIATIONROLE (section 6.1.3, page 27).
Qualifier (section 6.1.5, page 32).
Member direction (section 7.2.2, page 40)

Signal, which is not sufficiently distinguished from operation (section 7.2.1, page 38)
Instance (section 5.2.2, page 18).

Consistency concerns situations where the documents clearly articulate conflicting semantics. These can be conflicts between two of its own concepts or between its own and generally accepted concepts. Often the semantics and notation documents disagree. For example, the semantics document says actions in states are not interruptible, but the notation document says they are (see section 10.1.5, page 52). They also conflict regarding the definition of discriminators, which to the semantics document is closer to powertypes, but to the notation document is a way of classifying objects under types (see section 6.2.1, page 33).

Clarity concerns topics the meaning of which we could gather after reading between the lines, or making complex inferences based on general modeling knowledge. This is most often due to circular definitions and use of uncommon terms, but sometimes is just from inadvertent omissions. Examples are the definitions of powertypes, association multiplicity, isOrdered, pseudostates, internal transitions, and transitions. Any concept for which we draw an incorrect meaning should be regarded as a comprehensibility issue. Clarity is essential for a standard that is to be uniformly implemented by many vendors.

Issues that fall into more than one of the above types are classified under the topmost in the list.

1.2 Semantics

Many difficulties in the UML documents arise from failing to apply common sense regarding the definition of terms. Every user of a dictionary is aware, to begin with, that a word should not be defined in terms of itself. Secondly, since definitions form chains from word to word, it is expected that a chain of definitions which does not loop back to the original word should have generally understood terms at its “leaves”. Finally, any chain looping back to the original word should be long enough that the meaning can be understood. A chain of two, for example, two words defined in terms of each other, is too short if that is all the information available.

Because the above rules are violated so often in the UML documents, we were frequently forced to search through all the uses of a word to determine its meaning. It is admitted that the specification is not a tutorial, but on the other hand readers should not need to build their own concordance to understand the document. In many cases, the current document fails even as a reference manual.

The glossary is largely omitted from detailed analysis in this evaluation because so many of its definitions are circular. For example:

> Powetype is a composite aggregation of a generalization to one type (the powertype). A type is the powertype of a generalization.

Extension points is a composite aggregation of a type to a collection of names. The names are the extension points of the type.

Role is a shared aggregation of a role to zero or one types. A type is the role of a role.

The glossary did not help us understand the UML.

1.3 Table of Contents

2. Common Elements

3. Common Mechanisms

3.1 Modeling

3.1.1 Stereotypes

Summary: Stereotypes are for extending the meta-model, but the point of object-oriented meta-models is that they are already extensible. Users can add subtypes with a meta-object facility. In addition, stereotypes introduce unusual semantics despite the claim that they are the same as subtyping.

The semantics document gives a very clear definition of stereotypes:

> Stereotype is one of the three extensibility mechanisms of the UML, permitting a modeler to extend the classes of the UML meta-model in controlled ways. Specifically, an Element instance E classified by Stereotype instance S is semantically equivalent to a new meta-model class with the same name as S and whose supertype is the Element instance E. Every predefined stereotype in the UML could have been written explicitly as a new meta-model class whose supertype is the meta-model class to which the stereotype applies. [Common Mechanisms]

So wherever we find a stereotype, we could replace it with a subtype of the meta-type to which it's attached, or vice-versa, with no difference in effect. However, a later section assigns this behavior to stereotypes:

> As described in section 6, Type instances may participate in Generalization relationships. Type instances that are subtypes of another Type instance inherit all of the properties of their supertypes, including but not limited to the Stereotype ... [Types]

Since stereotypes inherit to specializations of the types to which they are attached, they are not semantically equivalent to subtyping the meta-model, which do not inherit. Here's another difference:

> Classification is a shared aggregation of an Element instance to no more than one Stereotype instance. The responsibility of Classification is to attach a Stereotype instance to an Element instance. Every Element instance may have at most one Stereotype instance, and every Stereotype instance may be attached to zero or more Element instances. [Common Mechanisms]

The number of subtypes that can be made from meta-types is presumably not restricted. Only one stereotype may be attached to any element, however. Finally, the mechanism for adding semantics to stereotypes is

> The value attribute of Stereotype is Uninterpreted and typically is used to establish new semantics and visual cues for the Element instance to which the Stereotype instance is attached. [Common Mechanisms]

which of course is very different from using a subtype with operations, attributes, and associations. These various differences from subtyping are of course inconsistent with the definition of stereotypes.

By the way, the Common Mechanisms glossary definition for stereotype points to section 2 even though stereotypes are defined in section 3.

Regarding modeling, the motivation for stereotypes given in the semantics document are these:

> The responsibilities of Stereotype are to provide a classification and to optionally establish additional semantics and visual cues for the Element instance to which it is attached. [Common Mechanisms]
>
> Taken to its natural conclusion, the UML could have been defined by exactly two classes - Thing and Stereotype - with all other meta-model concepts derived as stereotyped Thing instances. This would have been technically correct but practically unapproachable. Therefore, the philosophy taken in the UML is this: all fundamental meta-model concepts that embody sufficiently interesting semantics and that have complex relationships with other concepts are expressed as distinct meta-model classes. Furthermore, any meta-model concept that can be expressed as a simple subtype of these more fundamental meta-model concepts is treated as a stereotype. [Common Mechanisms]

The first is provided already by meta-subtyping, and the second is a notational issue, saying essentially that stereotypes can appear in a list below the meta-model diagram rather than on it, to reduce clutter.

The additional semantics that stereotypes provide do not seem worth introducing a new construct in the language, especially considering that functionality is removed by relegating semantics to an uninterpreted attribute. A stereotype could be just as easily and more completely modeled using a meta-type subtyped from the existing meta-model, additional semantics and visual clues added as normal through a meta-object facility.

3.1.2 Tagged Values

Summary: Tagged values are for extending the meta-model, but the point of object-oriented meta-models is that they are already extensible. Users can add attributes and associations with a meta-object facility.

The semantics document gives a very clear definition of tagged values:

> TaggedValue is the second of three extensibility mechanisms of the UML, permitting a modeler to extend the attributes of the classes of the UML meta-model in controlled ways. Specifically, an Element instance E with the characteristic TaggedValue instance T is semantically equivalent to the meta-model class E but with a new attribute whose name and type are the name and value of T. [Common Mechanisms]

So wherever we find a tagged values, we could replace it with an attribute of the meta-type to which it's attached, or vice-versa, with no difference in effect. Nothing that we can find in the semantics document contradicts this, but only if we agree that the way attributes are added to meta-types should be different than the way they are added to user-model types. Tagged values can have tagged values themselves, but attributes on user-model types cannot have attributes, at least as defined by the meta-model (see section 7.1.1, Attribute, page 36). Tagged values are not explicitly described as inheriting to the specializations of the types to which they are attached, though not restricted from it either, while attributes on user-model types always inherit. Finally, tagged values are given the tagset functionality for which there is no corresponding functionality in user-model attributes. All this means that the technique for defining new meta-attributes in the UML, namely tagged values, is different from the technique for defining user-model attributes. This seems like an unnecessarily complicated policy. Why not just use the same technique for both levels, preferably the one defined for user-models?

Regarding modeling, the motivation for tagged values given in the semantics document are these:

> The responsibility of TaggedValue is to provide a characteristic of the Element instance to which it is attached. [Common Mechanisms]
>
> Taken to its natural conclusion, the UML could have been defined with no attributes but will all characteristics of meta-model classes derived as TaggedValue instances. This too would have been technically correct but practically unapproachable. Therefore, the philosophy taken in the UML is this: all fundamental meta-model class characteristics that embody sufficiently interesting semantics are expressed as distinct attributes. [Common Mechanisms]

The first is provided already by meta-attributes, and the second is a notational issue, saying essentially that tagged values can appear in a list below the meta-model diagram rather than on it, to reduce clutter.

The additional semantics that tagged values provide do not seem worth introducing a new construct in the language, especially since it is different from similar functionality at the user-model level. A tagged value could just as easily be modeled using a meta-attribute added as normal through a meta-object facility.

3.1.3 Constraints

Summary: constraints are not given any semantics. They should be specialized from notes.

The definition of constraints in the semantics document is left intentionally unclear:

> Constraint is the third of the three extensibility mechanisms of the UML, permitting a modeler to extend the semantics of the UML in controlled ways. Specifically, an Element instance E with the Constraint instance C is semantically equivalent to the meta-model class E but with new semantics whose value is the value of C. [Common Mechanisms]

Since the "value" expressing the constraint is uninterpreted, this cannot be a definition of semantics. The semantics of constraints seems to be identical to those of notes, as supported by this comment:

> Visually, a note may be used to project any property of a model. In such cases, a Diagram instance may project a Note instance that is not itself part of a System instance, but rather exists just as a holder for the textual and graphical projection of some other Element instance property. For example, a note might appear in Diagram instance to display a TaggedValue instance or a Constraint instance. [Common Mechanisms]

Notes and constraints both just encapsulate a single uninterpreted value. They both inherit to specializations of the type to which they are attached. The difference is that the user will apply constraints more narrowly than notes. Consequently, CONSTRAINT should be a subtype of NOTE.

4. Common Types

5. Types, Classes, and Instances

5.1 Modeling

5.1.1 CLASS Specialized From TYPE

Summary: The meta-type CLASS, which has implementations, is specialized from TYPE, which does not allow them. This is a violation of basic subclassing semantics.

The UML defines generalization based on substitutability and consistency:

> The responsibility of Generalization is to specify an ordered unidirectional inheritance relationship, wherein an instance of the subtype is substitutable for an instance of the supertype. [Relationships]
>
> Generalization is the taxonomic relationship between a more general element and a more specific element that is fully consistent with the first element and that adds additional information. [Notation]

This requirement on user-models is not applied to the meta-model itself in the case of specializing CLASS from TYPE. As pointed out by Ernst, the specialization violates substitutability, because it allows a class to be used as an interface, thereby creating an interface which enforces an implementation. The UML defends this digression from standard modeling practice using the vague concept of “specification/realization dichotomy”:

> Class is a subtype of Type, and therefore instances of Class have the same properties as instances of Type, the fundamental difference being that Type instances specify interfaces, whereas Class instances specify the realization of these interfaces. This is the essence of the specification/realization dichotomy in the UML.   An implication of this dichotomy is that a Type instance may only have Operation instances as members, whereas Class instances may only have Method instances as members ... [Classes]

That is, types are constrained not to have methods, so all its subtypes may not have methods, except for classes. Following this logic, we could define a type BACHELOR as a man with no wives, then specialize HUSBAND from it by adding an association to wives. Subtyping under this definition allows information to be added to a specialization that is inconsistent with its generalizations.

In this particular case, the UML adopts the notion that subtypes are just for “adding things” to the supertype, omitting the consistency principle, and consequently confusing “is-a” with “has-a”.   This fundamental mistake of equating specialization with containment may be why the UML’s definition of aggregation is so weak (see section 6.1.2, Contextual Aggregation and Composition, page 21). Another negative effect is that the semantics of attributes is allowed to change from TYPE to CLASS. At TYPE, the notation document says the attributes are "abstract", that is, they do not imply an implementation (see section 10.1.1, States, page 45). However, at CLASS, we assume from UML's description that attributes are implementational.

CLASS and TYPE should be siblings in the meta-model, or at least cousins, with a meta-association between them for specifying the interfaces supplied by a class, instead of the vaguely defined "refinement" notion. See also section 8.1.1, Method Specialized From Operation, page 42.

5.1.2 Types as Instances

Summary: Types as instances of other types is an important technique supported by the notation, but only weakly in the meta-model by sweeping it under the rug of dependency. TYPE should be specialized from INSTANCE to support types of types.

Types can be instances of other types, even at the user-model level, as explained in the notation document:

> A generalization path may have a text label in the following format:
>
> discriminator : powertype
>
> where *discriminator* is the name of a partition of the subtypes of the supertype. The subtype is declared to be in the given partition;
>
> where *powertype* is the name of a type whose instances are subtypes of another type, namely the subtypes whose paths bear the powertype name. If a type symbol with the same name appears in the model, it designates the same type; it should be shown with the stereotype «powertype». For example, TreeSpecies is a powertype on the Tree type; consequently instances of TreeSpecies (such as Oak or Birch) are also subtypes of Tree. Either the discriminator, or the colon and powertype, or both may be omitted. Note that the word *type* also includes both types and classes. [Notation]
>
> Also see notation figure 23.

Consequently TYPE should be specialization of INSTANCE, so that types can be instances of other types. See additional discussion at section 7.1.2, IsTypeScope, page 36.

5.1.3 Multiple Classification and Semantic Variation Points

Summary: Multiple classification of objects, a technique with useful applications and available in some object-oriented languages, is explicitly disallowed at one point in the *documents and supported by an undefined extensibility construct in another. Specializing the existing meta-association for instantiation would be a simpler solution.*

The meta-association INSTANCE OF between TYPE and INSTANCE limits instances to being classified by one type. Roles also provide interfaces, but are restricted to subsets of those provided by the INSTANCE OF type. This does not cover some object-oriented languages, and also important applications. For example, a class library for GUI controls may have one class hierarchy for window-system independent information, and another for window-system dependent aspects. The library is used by making instances that have one parent from a window-system independent class and another from a window-system dependent class. Without multiple classification of instances, the classes from each library would need to subclassed in all combinations to provide GUI controls for each type of window system, even if they weren't all used in all applications.

The notation document handles this with "semantics variation points", an extensibility concept not described in the semantics document:

> There are different possible ways to interpret the semantics of generalization (as with other constructs). Although there is a standard UML interpretation consistent with the operation of the major object-oriented languages, there are purposes and languages that require a different interpretation. Different semantics can be permitted by identifying *semantic variation points* and giving them names, so that different users and tools could understand the variation being used (it is not assumed that all tools will support this concepts). These are some semantic variations applicable to generalization:
>
> > Multiple inheritance. Whether a class may have more than one superclass.
> >
> > Multiple classification. Whether an object may belong directly to more than one class.
> >
> > Dynamic classification. Whether an object may change class during execution.
>
> The ordinary UML semantics assumes multiple inheritance, no multiple classification, andno dynamic classification, but most parts of the semantics and notation are not affected if these assumptions are change.

Semantic variation points aren't defined in a comprehensible way. Specializing the GENERALIZATION and INSTANCE OF relation by restricting multiplicity or adding other constraints would achieve the same results without introducing a new extensibility concept that is only used once. See section 6.1.1, page 19, regarding specialized associations.

5.1.4 Multiplicity on Types

See section 6.1.2, Contextual Aggregation and Composition, page 21.

5.1.5 Operation as Global Entity

Summary: The meta-model explicitly disallows multi-polymorphism, that is, message dispatch on more than one type, an advanced technique available in some object-oriented languages. One solution, separating operations from types, has the additional benefit of supporting polymorphism without requiring a common supertype.

The meta-model has meta-association MEMBERS between TYPE and MEMBER, and OPERATION is specialized from MEMBER. It is an unshared aggregation with Type as the whole. This eliminates the possibility of multi-polymorphism, message dispatch on more than one type, which is supported by CLOS. Operations restricted to one type also encourage semantically identical operations to be declared on types that are not related by specialization. For example, an operation that prints may be used on disparate types that cannot be related by specialization for any other reason than printing. Rather than force modelers to make a spurious supertype PRINTABLETHING type to inherit the PRINT operation from, the PRINT operation should be a global entity. This global operation entity should have meta-associations to the types that it dispatches through and the methods it dispatches to, in such a way that the method can be found given the types and operation.

The UML implicitly suggests the above approach by the use of an association class for the MEMBERS meta-association between TYPE and MEMBER. Such an association class is unnecessary in the UML since its intention is to never share an instance of MEMBER. It would be simpler to put the attributes of this class directly on MEMBER. On the other hand, the association class makes it easy to share instances of MEMBER, and of OPERATION in particular, by just changing the MEMBERS aggregation to non-compositional.

5.2 Comprehensibility

5.2.1 Roles of an Instance

Summary: The definition of roles of an instance is not comprehensible due to circular definitions, use of undefined terms not commonly understood, and confusion with the definition of type.

The definition of the ROLES meta-association between INSTANCE and TYPE is

> Roles is a shared aggregation of an Instance instance to a collection of Type instances. The responsibility of roles is to specify the role that the given Instance instance is playing at a moment in time/space, where role in this context means the face or faces that the Instance instance is presenting to its clients. Whereas an Instance instance is always the instance of exactly one Type instance, the roles of an Instance instance may change. [Types, Classes, and Instances]

The circularity in the second sentence is broken by reference to the concepts of “client” and “face”. “Client” is only defined by the glossary, namely as “a type that uses an interface" [Classes glossary], with no other elaboration. "Face" "refers to the most significant or prominent surface of an element, especially one used for interaction or communication" [Introduction], a term UML explicitly exempts from its ban on "arcane terms". These definitions provide no relation to concepts in the rest of UML or common experience, other than the general notion of interface already covered by the TYPE meta-type. The derived semantics, however, compares "role" as defined for instances to "role" as defined for associations:

> The roles relationship described in this section interacts with the semantics of refinement as well as with the semantics of the role relationship described in section 6. Basically, the Type instances for which another Type (or subtype of Type) instance is the refinement of constitute the static interfaces of the refining Type instance. The role that a given refining Type instance plays in an Association (as described in section 6) must be equal to or a subset of these static interfaces; this is a statement of the static semantics of the refining Type instance and the role that it plays in an Association instance. The roles relationship described in this section must also be equal to or a subset of these static interfaces; this is a statement of the dynamic semantics of the Instance instance. In other words, the interface of a Type instance is static, but may be subsetted in a given context; further, the interface of an Instance instance is dynamic, because at different moments in time, that Instance instance plays a different role in the world. [Types, Classes, and Instances]

A constraint is added both to instance and association roles that the interfaces they define are a subset of the "static" interface defined by the INSTANCE OF meta-association between INSTANCE and TYPE. No interaction between the semantics of the two kinds of role is actually specified, only a point of similarity. Consequently the meta-association ROLES between INSTANCE and TYPE is still not defined in terms of anything else. By the way, the combination of "for which" and "of" in the second sentence makes it ungrammatical.

The additional restrictions given in the following paragraph are not comprehensible:

> Between each Type instance and its Instance instances, the values, actions, state instance, and roles of the Instance instance must match the attributes, actions, states, and roles of the associated Type instance. ... Matching a role instance to a Type means that the role instances must be one of the potential roles of the Type instance. [Types, Classes, and Instances]

The last sentence seems to say that role instances, presumably those types participating in instances of the ROLES meta-association, are instantiated from roles of types, but types are not instances in the UML, so don't have roles. Perhaps this is just a restatement the subsetting constraint of the previous paragraph, but it's unclear.

The following paragraph is also very difficult:

> The direction attribute of Signals and Members permits the specification of roles provided by a Type instance as well as roles wanted by a Type instance. A role provided by a Type instance constitutes an obligation of the Type instance to respond to the Signal instances specified as received, and a declaration of the Member instances specified as provided. Similarly, a Type instance may specify

> a role wanted by the instance. In this context, a wanted role is subset of the full set of the accumulation of all of the association roles opposite the role the given Type instance participates in. Member instances. This permits an evaluation of closure among the Type instance of a model. Specifically, if a given Type instance sends a given Signal instance, some other Type instance can be checked as receiving that same Signal instance. Similarly, a given Type instance may provide certain Member instances, which in turn are required by some other Type instance. The UML does not specify any strong semantics for the binding of providing and wanting roles, other than to permit their specification and to encourage their matching. The rationale for these semantics are that strong matching semantics are possible only in complete, self-consistent, and unchanging models, where as the vast majority of models are by their very nature incomplete, self-inconsistent, and constantly changing. [Types]

Roles, again used presumably to mean those types participating in instances of the ROLES meta-association, are divided into "wanting" and "provided". This division is not made in the meta-model. The definition for "provided role" doesn't give anymore semantics than the DIRECTION attribute of SIGNAL and MEMBER already do, so seems unnecessary. The definition of a "wanted role" is not clear because the term "opposite" as applied to association roles isn't defined. We gather, however, that wanting roles are a subset of association role types specified for the associations that a type participates in, following the ROLE meta-association between ASSOCIATIONROLE and TYPE. If this is the case, we don't understand why the wanting roles are a subset of the association role types, since all associations to a type could potentially need to use it. Perhaps the subsetting indicates some additional semantics over the association role types, but we can't figure out what they are. In any case, the support for consistency-checking between wanting and providing is more a tool issue than a modeling issue and should be omitted from the meta-model (see section 7.2.2, Member Direction, page 40).

That the terms "type" and "role" are used interchangeably in the above indicates a lack of semantics for the term "role" to clearly distinguish it. The uses of the term in the notation document suggests a connection to collaboration, but this isn't drawn out:

> A collaboration (as a complete entity representing a design pattern) is shown as a dotted ellipse containing the name of the pattern. A dotted arrow is drawn from the collaboration symbol to each of the objects or classes (depending on whether it appears within an object diagram or a class diagram) that participate in the collaboration. Each arrow is labeled by the *role* of the participant. The roles correspond to the names of elements within the context for the collaboration; such names in the collaboration are treated as parameters that are bound to specify elements on each occurrence of the pattern within a model. [Notation]

> A pattern is a template collaboration. [Collaborations synonym list]

The topic of collaborations is addressed in section 9.1.1, page 43.

Admittedly the UML documents are not tutorials, but some connection should be drawn between the ROLES meta-association between INSTANCE and TYPE and the rest of the language for the reader to understand its purpose. By comparison, contextual aggregation as defined by Bock and Odell gives a semantic basis for roles by showing how subtypes of aggregated objects can be used to add context-specific constraints (see section 6.1.2, Contextual Aggregation and Composition, page 21).

5.2.2 Instance

Summary: The definition of instance is not comprehensible due to use of undefined terms not commonly understood, and a unusual definition of persistence.

The semantics documents gives this definition for the INSTANCE meta-type:

> Instance is a subtype of ModelElement. The responsibility of Instance is to specify the concrete manifestation of a Type instance. Whereas a Class instance provides the realization of a Type instance, an Instance instance corresponding to a Class instance manifests a Type instance in time/space, meaning that the Instance instance represents an entity that exists in time and space. [Types, Classes, Instances]

We can gather from description that the instances of types are used for abstract things like describing interactions, whereas the instances of classes are used for modeling actual objects in the real world. However, the distinction between “concrete manifestation” and “realization” is not a good way to illuminate the difference.

The following paragraph is more difficult:

> A Type instance that has one or more provided operations for which isAbstract is True represents one that may not directly have any corresponding instances in the real world, although in a Model instance, there may be Instance instances that correspond to an abstract Type instance, representing a prototypical instance of one of its subtypes. [Types]

Since prototypes are not defined in the UML, the above description is not comprehensible.

The definition of persistence made us doubt our understanding of UML instances, and classify them as a comprehensibility issue:

> Persistence is the specification of the permanence of the state of an instance. Persistence is an enumeration specified as {transitory, persistent}. A transitory instance is one whose state is destroyed when the instance is destroyed; a persistent instance is one whose state is not destroyed when the instance is destroyed. The default value of persistence is transitory. Specifying this tagged value on a Type instance constrains the persistence semantics of its instances: all of the instances of a transitory type are transitory, and all of the instances of a persistent type are either transitory or persistent. Specifying this tagged value on an Instance instance states the actual persistence semantics of that instance. Specifying this tagged value on an Attribute instance specializes the persistence property of its owning Type instance. [Types, Classes, and Instances tagged value list]

The only interpretation we can think of for preserving the state of a destroyed instance is if instance refers to data stored in main memory. This is too low-level for an implementation independent modeling language, because model instances should be modelable whether or not they happen to be stored in main memory.

5.2.3 Responsibility

Summary: The definition of instance is not comprehensible due to use of undefined terms not commonly understood.

The semantics document defines the concept of responsibility this way:

> The responsibility of Responsibility is to indicate a contract by or an obligation of the Type instance to which it is attached. The value attribute of Responsibility is a string indicating the contract or obligation of the Type instance to which it is attached. [Types, Classes, and Instances]
>
> A responsibility is a contract by or an obligation of the type to which it is attached. [Types, Classes, and Instances glossary]
>
> As described in section 5.4, responsibility is a predefined tagged value that applies to Type and that specifies a contract or obligation of the Type instance to which it is attached. During the lifetime of this Type instance, such responsibilities are typically refined and ultimately realized by the members of the Type instance. It is possible to specify an explicit trace from a responsibility to the Member instances that realize it, by introducing a trace Dependency instance whose target is the Responsibility instance and whose sources are the Member instances of the Type instance to which the responsibility is attached. [Types]

The terms “contract” or “obligation” are not defined or related to other terms. If they are intended to incorporate other modeling languages, then at least references should be made to these for the reader to follow. Of course it would be best if the UML could define these terms itself or give some indication of how they are to be used.

6. Relationships

6.1 Modeling

6.1.1 Association and Association Class

Summary: Associations are identified with classes in the text, but separated in the meta-model. This results in unnecessary additions to the meta-model because the notion of class is not properly reused. It also makes the common technique of specializing associations too cumbersome, and its importance to implementing roles obscure.

The notation document states that associations are classes:

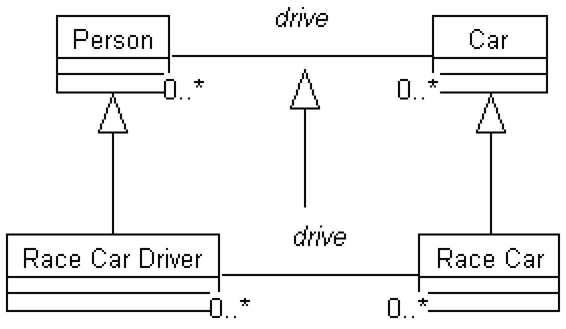
> Logically the association class and the association are the same semantic entity, but they are graphically distinct.

> An association class is an association that also has class properties (or a class that has association properties). Even though it is drawn as an association and a class, it is really just a single model element.
>
> A link is a tuple (list) of object references. ... It is an instance of an association.

And the semantics document uses terminology that suggests the same thing:

> Link instance is an association between a Link instance and an Association instance. The responsibility of Link instance is to specify that the Link instance is an instance of the Association instance. [Interactions]

The meta-model, on the other hand, has a meta-association ASSOCIATION CLASS between ASSOCIATION and CLASS, rather than a generalization between ASSOCIATION and CLASS. This forces creation of the LINK INSTANCE meta-association between LINK and ASSOCIATION, which is redundant with the INSTANCE OF meta-association between INSTANCE and TYPE. It also makes modeling association specialization more cumbersome. Take for example:



The association between CAR and Person is specialized when inherited so that only race car drivers can drive race cars. The specialized association inherits all the characteristics of normal driving, like obeying speed limits on city streets, while restricting the participating types. Association specialization follows the same semantics of normal subtyping, including that the instances of the specialized association form a subset of the instances of the generalized one.

The above can be modeled in the UML by using two associations, one for the upper and one for the lower association in the example above, relating their association classes with generalization, and arranging that the association roles on the lower association have specializations of the participating types and roles of the upper. Association specialization is a common feature of modeling, so should be provided as a straightforward service instead of requiring the user to piece together existing components. Making associations a kind of class would inherit some of this ability from generalizable elements, which could be specialized to require that the participating types, role types, and multiplicities of the special association are subtypes of the corresponding aspects of the general association. It would also save repeated traversals between association and association class, a common occurrence between objects that have identical semantics, for example, to invoke an operation on the association.

6.1.2 Contextual Aggregation and Composition

Summary: In the last two revision cycles , the UML attempted to address the advanced notion of objects playing roles in an organizing structure or pattern. This hurried schedule might have worked had the authors consulted existing work on the subject. As it is, the resulting model cannot support its claim to handle objects playing different roles in various "composite objects" without the roles affecting each other.

Please first read section 6.1.1, Association and Association Class, page 19.

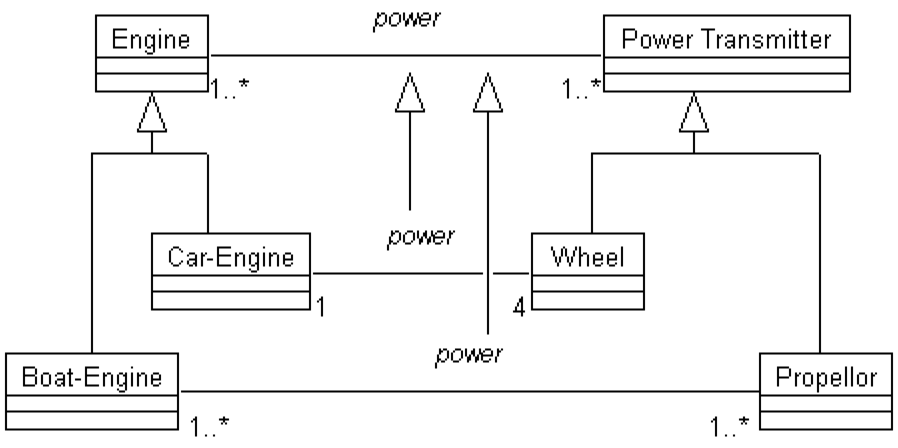
The notation document partially describes a modeling technique introduced by Bock and Odell (references at the end of this section), but which is not covered or completely supported by the semantics document. This topic is best called "contextual aggregation" when described to a UML audience, but the notation document uses the term "composition", defined differently from the semantics document:

> Within a composite additional associations can be defined that are not meaningful within the system in general. These represent patterns of connection that are meaningful only within the context of the composite. Such associations can be thought of as generating *quasiclasses* (or *qua-types* as Bock and Odell call them) that are specializations of the general classes; the specializations are defined only inside the composite.

The term *qua-type*, not "quasiclasses", is taken from the early AI language, KL-One. It is derived from phrases like "Man *qua* Husband", which refers to those aspects of a man relevant to his role in a marriage. The UML-knowledgeable will be reminded of the meta-association ROLE recently added between TYPE and ASSOCIATIONROLE in the meta-model. It is addressed in this section also. We discuss the requirements for contextual aggregation, then evaluate whether UML provides it.

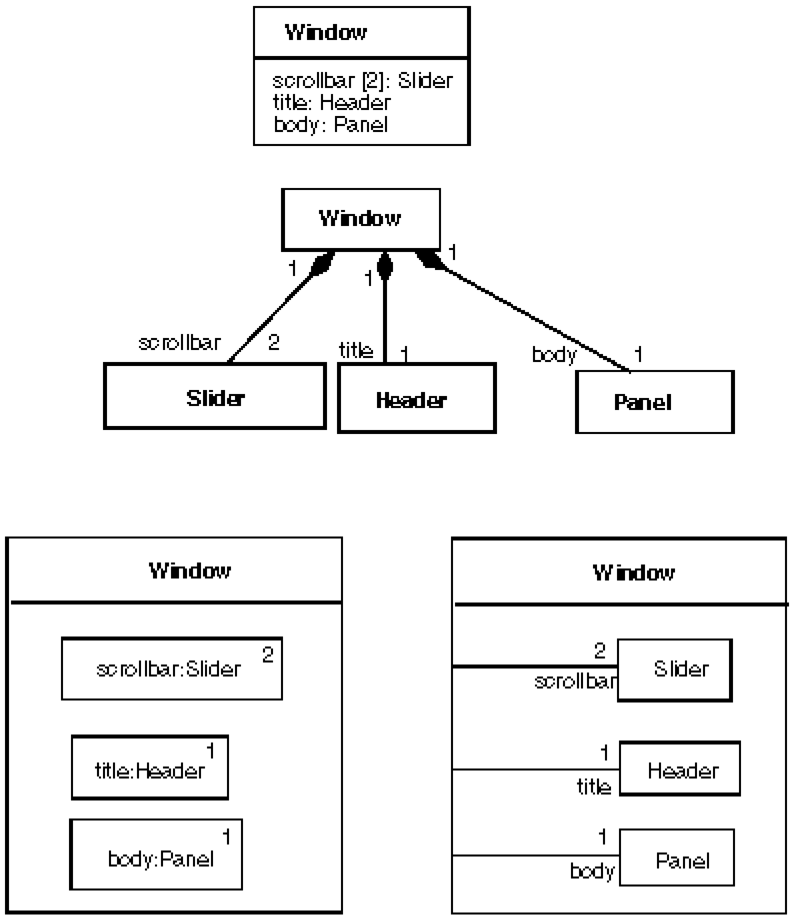
Qua-types of aggregation

Applying qua-types to aggregation, Bock and Odell (1994) note that aggregation is like any other association in that it's participants take on certain characteristics, or just as importantly, restrict existing characteristics, due to being associated. For example, an engine used in a car will power wheels, whereas the same engine used in a boat will power a propeller. This context-dependent information is expressed informally as "Engine *qua* Car-Engine" or "Engine *qua* Boat-Engine", and formally expressed by specializing the type ENGINE into subtypes CAR-ENGINE and BOAT-ENGINE. These particular kind of subtypes, called *qua-types*, are used to specialize the POWER association between ENGINE and POWER TRANSMITTER so that CAR-ENGINE powers WHEEL, and BOAT-ENGINE powers PROPELLER (see section 6.1.1 on specializing associations):



The above model has the advantage of retaining two levels of abstraction, but still keeping them coordinated. It records that engines power transmission devices, for applications that work on a general level, and that they power wheels and propellers in cars and boats respectively, for applications on those particular levels. The model also requires that general and particular levels are connected by specialization. A simpler diagram for defining qua-types is described in the context diagram subsection below.

The notation document does not elaborate on qua-types in its description for contextual aggregation. For example, see Figure 20:



The types SLIDER, HEADER, and PANEL are shown in the middle as being aggregated into WINDOW without specializing them for that context. If there are characteristics of SLIDER peculiar to being in this particular window, like its range of values, there is nowhere to record it other than on sliders in general, affecting all other uses of slider. Contextual aggregation, or composition as the notation calls it, cannot be treated as a simple association without severely impairing modularity and reusability of the aggregated parts.

UML roles of aggregation

The meta-association ROLE, recently added to the UML between TYPE and ASSOCIATIONROLE, is not defined by the semantics document. The discussion of related aspects, like the ASSOCIATIONROLE meta-type and the ROLE meta-association between INSTANCE and TYPE, is enough to warrant applying it to contextual aggregation. Naturally, this constitutes a clarity issue. These quotations best sum up the UML description:

> The responsibility of AssociationRole is to specify the face that a type plays in an association. [Relationships]

where

> ... “face” refers to the most significant or prominent surface of an element, especially one used for interaction or communication. [Introduction]

Relating the above to ROLE between INSTANCE and TYPE:

> As described in section 5.4, there may be refinement Dependency instances whose source is a Type or Class instance and whose target is a Type instance. The target of a refinement specifies an interface, and this interface specifies a role of the source. Collectively, the target Type instances of all the refinement Dependency instances whose source is a given Type instance are called the roles of the Type instance. This concept of roles interacts with the semantics of association roles as described in section 6. For a given Type instance that participates in AssociationRole instances in multiple Association instances, each such AssociationRole instance specifies a role for that Type instance in the form of another (or the same) Type instance that specifies an interface. The complete roles of a Type instance must be a superset of the association roles in which that Type instance participates. [Types]
>
> In other words, the interface of a Type instance is static, but may be subsetted in a given context; further, the interface of an Instance instance is dynamic, because at different moments in time, that Instance instance plays a different role in the world. [Types, Classes, and Instances]

We take the above to mean that roles are the various interfaces which an object will provide when participating in various relations. This is not strong enough to support contextual aggregation, because adopting different interfaces cannot express the specialization of methods or associations on existing interfaces inherited to qua-types.

For example, the methods on engines by which they are operated may be specialized when an engine is used in a car rather than a boat, that is on CAR-ENGINE and BOAT-ENGINE. Likewise, the associations which connect engines to the transmitters are restricted in cars to connect only to wheels, as presented in the previous subsection. It is necessary to specialize ENGINE to CAR-ENGINE, inheriting the same associations and interfaces to those associations, but adding a constraint that CAR-ENGINE can only be connected to WHEEL. Finally, there may be additional constraints on attributes, as shown by the example of the previous subsection in which the range of a slider is restricted when it is used in a particular dialog box.

In all the above cases, filling a "role" means that objects come under additional constraints when implementing existing interfaces, not just implementing new interfaces as proposed in the UML. To make a day-to-day analogy, we as people use many of the same "interfaces" at home and at work. In both cases we talk, look at other people, think, and use body language. But the methods implementing these interfaces at home and work come under very different restrictions, which the reader can easily imagine. The UML proposes that adopting different interfaces are all that is needed to define the different ways an object behaves in any situation. Qua-typing also provides for adopting new interfaces, but in addition recognizes the need to constrain inherited methods, associations, and attributes, thereby providing a much more powerful foundation for the concept of "role".

Lifetime of parts in a contextual aggregation

The notation document gives this definition for contextual aggregation:

> Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end may not exceed one (it is unshared). The aggregation is unchangeable (once established the links may not be changed). Parts with multiplicity > 1 may be created after the aggregate itself but once created they live and die with it. Such parts can also be explicitly removed before the death of the aggregate. [Notation]

Naturally when modeling an aggregate system, we want to be able to remove parts, for example, when replacing an engine in a car. The last sentence acknowledges this, but the rest of the paragraph directly states that this is not possible. The definition also rules out making a composite from existing parts. It is an impractical modeling technique for contextual aggregation to require parts to exist exactly when the whole does.

By the way, the term "changeable" is used in the above quote differently than the semantics document, which is taken up in section 6.1.3, ISCHANGEABLE and Aggregation, page 27.

Contextual multiplicity

There are two MULTIPLICITY meta-attributes in the meta-model. One of them indirectly refers to contextual aggregation as defined in the notation document:

> The multiplicity attribute of Type is Multiplicity and is used to specify the number of allowable Instance instances of the Type instance within a specific composite. [Types, Classes, and Instances]

It is a consistency issue that there is no other explanation of contextual aggregation in the semantics document to support the above definition. Most importantly, it is not specified that the multiplicity is only useful on qua-types, rather than all types. The same problem arises in the notation document:

> The entire system may be thought of as an implicit composite, so that any multiplicity specifications within toplevel classes restrict the cardinality of the classes in a particular execution; ... [Notation]

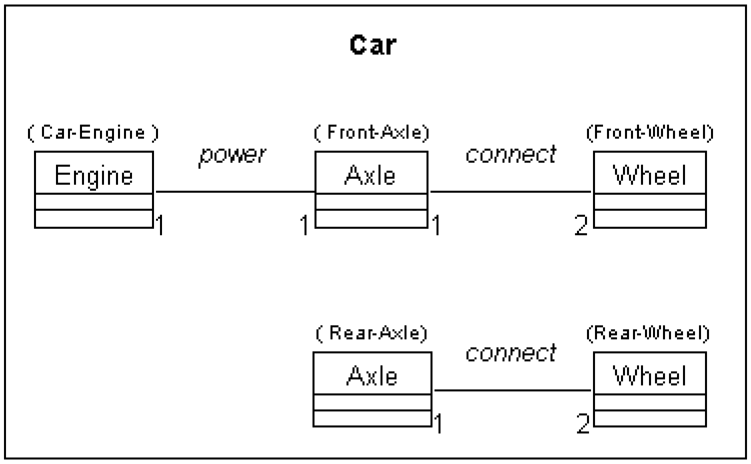
Generally top-level classes, if we understand the meaning of "top-level", should not have contextual multiplicity. For example, if the class WHEEL had a multiplicity of 4, it could only be used in cars, not bicycles.

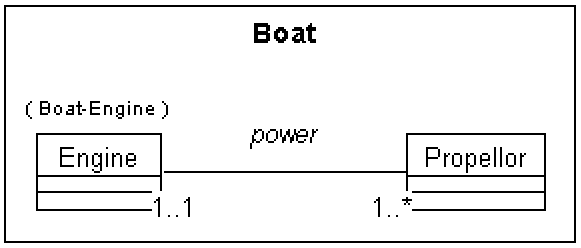
Contextual aggregation diagrams

The term "context" is used in a wide variety of ways in the notation document. In particular, it is used in the definition of contextual aggregation (see quote at beginning of this section), and sometimes in connection with the term "role". The definition given for context is:

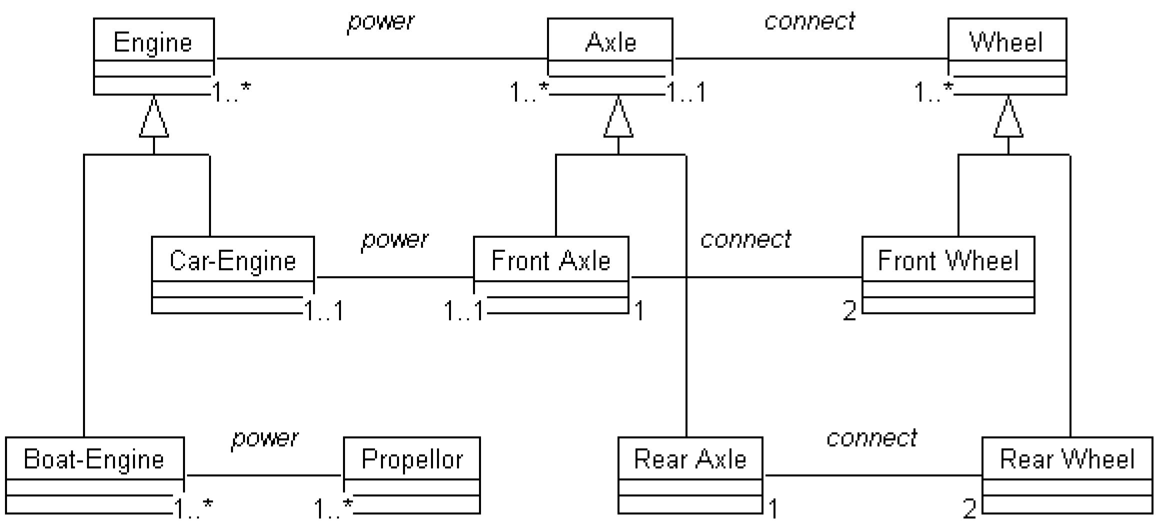
> A *context* is a model fragment that shows one or more classes together with their contents, associations, and neighbor classes, plus additional relationships and classes as needed to define operations on the class. Any classes not shown are not affected by operations on the class (or by a particular operation). [Notation]
>
> Since each context shows a local view of the entire system, classes may appear slightly differently in different contexts. Each context may show the attributes and relationships important to its purposes and suppress the others. Ultimately each context must be a projection from a consistent model of the entire system, but within a single local view the scope of each element in the context is not specified. [Notation]

So a context is just a set of view elements projected from a model, as the semantics document defines it, which makes it the same as any sort of diagram. Apart from the fact that they are redundant, contexts are not a powerful enough diagramming technique for contextual aggregation. For example, here is are two contextual aggregation diagrams adapted from Bock and Odell (1994):





The above diagrams are short hand for explicit definition of all the qua-types, notated in parentheses above each class. The full class diagrams looks like this:



When the user adds a type to a contextual aggregation diagram, qua-types should be created automatically for the context. This include the associations, which may need to be specialized from more general types. People use contextual aggregation in their minds very fluently, so any visual interface to it should downplay the creation of qua-types. They are a lower-level artifact of making semantics precise enough for a machine to interpret properly.

Conclusion

Contextual aggregation, only some of which is described in this evaluation, is a proven concept used in IntelliCorp products since 1985. By comparison, the ROLE meta-association between ASSOCIATIONROLE and TYPE is a recent addition to UML, conceptually approached in 0.91 and appearing in 1.0. It would best if existing work in this area were used in defining the OMG standard. The references are:

1. A Foundation for Composition [Bock & Odell, JOOP, 7:6, Oct 94, pp10-14, or http://www.intellicorp.com/ooieonline/compfoundation.html]

   A set-theoretic basis (written in English, though) for contextual aggregation, as well as some user-level services built on it. Portions of this paper are reproduced in Rumbaugh's JOOP column of Nov/Dec 95.

2. A User-Level Model of Composition [Bock & Odell, ROAD, 2:7, May/June 96, pp5-7, or http://www.intellicorp.com/ooieonline/compuserlevel.html]

   A user-view of our models, including computational services and diagram notation.

3. http://www.intellicorp.com/ooieonline/compinheritance.html

   An html demo presenting an visual interface to some of our model.

4. Six Different Kinds of Composition [Odell, JOOP, 6:8, January 1994, pp 10-15, or http://www.intellicorp.com/ooieonline/compkind.html]

   An analysis of intuitions on part/whole, showing the axes on which they can be broken down. Based on research in cognitive science.

6.1.3 isChangeable and Aggregation

Summary: The definition of isChangeable is not comprehensible due to use of undefined terms not commonly understood. A kind of aggregation called "shared aggregation" is defined that is identical to normal associations, we assume because users like it. The introduction of such a "placebo", as Rumbaugh calls it, encourages modelers to define their own meaning, thereby defeating the purpose of the standard.

The definition of ISCHANGEABLE meta-attribute on ASSOCIATIONROLE is bound up with its use in defining aggregation, so they are covered together in one section.

isChangeable and composite aggregation

ISCHANGEABLE is defined as

> The isChangeable attribute of AssociationRole specifies the mutability of the relationship; the default value of isChangeable is True, meaning that the semantics of the Association instance are preserved even if the instance of the participating Type instance is replaced by a different instance of a Type instance. [Relationships]

There are at least two interpretations of "preserving semantics" that might be consistent with the UML documents:

1. If ISCHANGEABLE is False on an ASSOCIATIONROLE instance, then we are not allowed to change which object is participating in that association role without destroying the entire association instance. Odell uses the example of marriage, any instance of which is constant during its life. Attempts to change the husband or wife destroy the marriage instance. This is Odell's definition of immutability [private communication].

2. If ISCHANGEABLE is False on an ASSOCIATIONROLE instance, then we are not allowed to change which object is participating in that association role instance without destroying the corresponding object and the association instance.

The only other reference to ISCHANGEABLE or changeability in the documents are these (first paragraph included for context):

> The isAggregate attribute of AssociationRole specifies if the participating Type instance is the whole in a whole/part association. For all the AssociationRole instances associated with an Association instance, at most one AssociationRole instance may have isAggregate set to True, designating the participating Type instance to the be whole of the aggregation and all other participating Type instances to be the parts. When isAggregate is set True for at least one AssociationRole instance that is part of a given Association instance, the value of the multiplicity attribute has semantic implication for the life times of the whole and the part. Specifically, if the multiplicity of the whole is no greater than one, then the whole is said to own the parts, and destroying the whole destroys the parts. If the multiplicity of the whole is greater than one, then the whole is said to share the parts, and destroying the whole does not necessarily destroy the parts. [Relationships]
>
> An aggregation relationship specifies an Association instance with exactly one AssociationRole instance whose isAggregate attribute is True. Setting this attribute only specifies a whole/part relationship (with the participating Type instance associated with the AssociationRole instance whose isAggregate attribute is True designated as the whole); it says nothing about navigability, ownership, or lifetimes. A composite aggregation is a strong form of aggregation, with a multiplicity of no more than one established for the whole, and isChangeable set to True for the whole. The implication of a composite aggregation is that the whole owns its parts, and that the whole represents a shift in levels of abstraction over the parts. An aggregation relationship that is a multiplicity of greater than one established for the whole is called shared. By

> implication, a composite aggregation forms a tree of parts, whereas a shared aggregation forms a graph. [Relationships]
>
> Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end may not exceed one (it is unshared). The aggregation is unchangeable (once established the links may not be changed). Parts with multiplicity > 1 may be created after the aggregate itself but once created they live and die with it. Such parts can also be explicitly removed before the death of the aggregate. [Notation]

It isn't clear that the use of "changeability" in the notation document is the same as ISCHANGEABLE, but if it is, then perhaps the second interpretation applies. The association roles for the parts would have their ISCHANGEABLE attributes set to False, so that removing the part from the whole would also destroy the part. But destroying the whole would not destroy the parts, so the second interpretation doesn't cover all of the notation definition.

The use of ISCHANGEABLE in the semantics document doesn't provide any clues to its meaning, especially because it takes the default value of TRUE in defining composite aggregation, a value which has no unusual effect. The first paragraph quoted above would suggest that ISCHANGEABLE completes the semantics of composite aggregation but it doesn't say how. Since ownership seems to be a central aspect of composite aggregation, we might make some guesses:

1. Perhaps ISCHANGEABLE was meant to be False for both participants in composite aggregation, but since it applies ISCHANGEABLE only to the whole, and if we invoke the second interpretation, then trying to change the whole of a part without first deleting the association instance would destroy the whole. This is an impractical policy. The first interpretation doesn't seem applicable to aggregation, since it is not worth preserving the identity of a particular part-whole association instance, like it is a marriage.

2. Perhaps ISCHANGEABLE was meant to be set to False on all the association roles of the parts, with the second interpretation taken. Then a part is destroyed if it is removed from the whole. This would complete the ownership semantics started by requiring that destroying the whole destroys the parts in composite aggregation. However, it leaves a loophole in that destroying the association instance first would allow the part to be removed from the whole.

This counts as a comprehensibility issue, of course. We've included it in modeling to say that Odell's definition of mutability is obviously useful from his example, so should be considered in the UML submission. Also, an interpretation should be given that clarifies the relation of ISCHANGEABLE to compositional aggregation.

Non-composite aggregation

Non-compositional aggregation, that is, aggregation with multiplicity greater than one on the whole, is given no semantics in the UML documents apart from that of normal associations. We understand users like it anyway, which perhaps led Rumbaugh to call it a "placebo". Unlike medicine, however, placebos in modeling have severe negative effects that outweigh the benefits. Users will adopt various interpretations which of course will go uncoordinated until their complete system is tested. We recommend that non-composite aggregation be dropped from the UML entirely.

6.1.4 Association Navigation

Summary: Association navigation does not handle some common applications. We propose a navigation model that provides more flexibility.

The association navigation model is vaguely defined, perhaps inconsistent, and certainly incomplete. The semantics document says:

> The isNavigable attribute of AssociationRole specifies if the association is navigable to the participating Type instance, where navigable means that given an instance of the Type instance is directly reachable via the Association instance; the default value of isNavigable is True, meaning that the participating Type instance is navigable. Any number of the AssociationRole instances associated with an Association instance may have isNavigable set False. [Relationships]

One interpretation of "via" is that the source of the navigation being specified is the association instance and the target is one of its participants. The phrase "the participating Type instance is navigable" uses the term being defined in its definition, so can't be interpreted.

The notation document says:

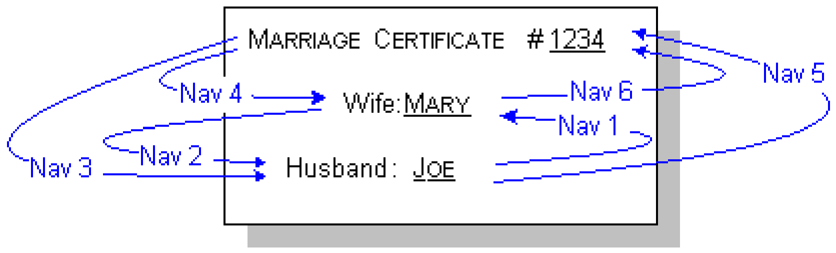
> An arrow may be attached to the end of the path to indicate that navigation is supported toward the class attached to the arrow. Arrows may be attached to zero, one, or two ends of the path. In principle arrows could be shown whenever navigation is supported in a given direction. [Notation]

This is ambiguous because it gives the target of the navigation without the source. The particular choice of notation would suggest that the navigation is from one participant to the other. This of course would be inconsistent with the possible interpretation of the semantics document given above.

In any case, neither of the interpretations provide a complete definition of association navigation, even when taken together. For example, the MARRIAGE relation has six navigation paths:

1. Given the husband, return the wife.
2. Given the wife, return the husband.
3. Given the instance of marriage (the certificate), return the husband.
4. Given the instance of marriage, return the wife.
5. Given the husband, return the marriage.
6. Given the wife, return the marriage.

In graphical form:



Any of the navigations, or any combination of them, might be useful in a particular application. For example, city hall may handle its marriage records by implementing only navigations 5 and 6, because the most frequent query it gets is to find the marriage certificate for a particular man or woman. The OMG standard should remain agnostic on which navigation will be optimal in any particular application.

6.1.5 Qualifier

Summary: The notion of qualifier for associations omits association and classification as qualifying aspects of objects. Simple examples are not handled.

The semantics document definition of the QUALIFIER meta-association between ASSOCIATIONROLE and ATTRIBUTE substitutes the word "role" for "qualifier":

> Role is a shared aggregation of an AssociationRole instance to a collection of Attribute instances. The responsibility of role is to specify how the instances of Type instance are partitioned at one end of an Association instance.
>
> The Attribute instance is said to qualify the Association instance, meaning that across an Association instance with an AssociationRole whose multiplicity attribute specifies greater than one instance of a participating Type instance, the qualifier (or qualifiers) may be used to designate a specific instance of the Type instance. [Relationships]

The confusing phrases "how the instances of Type instance are partitioned at one end of an Association instance" and "across an Association instance" are clarified by the notation document:

> The qualifier is attached to the source end of the association; that is, an object of the source class together with a value of the qualifier uniquely select a partition in the set of target class objects on the other end of the association. [Notation]
>
> The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value. Common values include "0..1" (a unique value may be selected, but every possible qualifier value does not necessarily select a value), "1" (every possible qualifier value selects a unique target object, therefore the domain of qualifier values must be finite), and "*" (the qualifier value is an index that partitions the target objects into subsets).

These are clearer definitions, even though associations don't have sources and targets. Presumably these terms are used because qualifiers apply to navigation of associations, rather than associations themselves. The submission should make it clear that qualifiers apply to association navigation.

Regarding modeling issues, target objects of an association navigation can be qualified by their participation in association as well as the values of their attributes. For example, suppose we have a MEMBER association between ACTIVITYGROUP and PERSON. The activity group may want to partition its members according to whether they have children. In this case, navigating towards PERSON along the MEMBER association would be qualified by a person's association to children. The same argument can be made for partitioning target objects by type. Using the activity example again, the model may partition members according to whether they are classified under SINGLE and MARRIED.

6.2 Consistency

6.2.1 Discriminator

Summary: The semantics and notation documents conflict regarding the definition of discriminators, which to the semantics document is closer to powertypes, but to the notation document is a way of classifying objects under types.

The semantics and notation documents have different definitions of discriminator. The semantics document defines it this way:

> The Name instance associated with a Generalization instance is called the discriminant of the relationship. ...For a given supertype, there may be Generalization instances with the same discriminant Name instance, meaning that these identically named relationships partition all of the subtypes of the given supertype into a set named by this discriminant. [Relationships]

This is a set of types, which is very close to a powertype (see section 6.3.1, Powertypes, page 34). However, powertypes define types that describe types, rather than just sets that have types as members.

The notation document defines discriminator this way:

> Generalization is shown as a solid-line path from the more specific element (such as a sub-class) to the more general element (such as a superclass), with a large hollow triangle at the end of the path where it meets the more general element. A generalization path may have a text label in the following format:
>
> > discriminator : powertype

where *discriminator* is the name of a partition of the subtypes of the supertype. The subtype is declared to be in the given partition;

...

Either the discriminator, or the colon and powertype, or both may be omitted. Note that the word *type* also includes both types and classes. [Notation]

The *discriminator* must be unique among the attributes and association roles of the given superclass. Multiple occurrences of the same discriminator name are permitted and indicate that the subclasses belong to the same partition. [Notation]

Also see figure 22. The notation document implies that discriminator is used to specify how objects are classified under types, using attributes and associations. Of course the semantics and notation documents should be consistent.

6.3 Clarity

6.3.1 Powertypes

Summary: the semantics document uses a circular definition for powertypes, and is different from the notation document.

The definition of powertypes in the semantics document is circular:

Powertype is a composite aggregation of a Generalization instance to a Type instance. The responsibility of Powertype is to specify the Type instance that is the powertype of the Generalization instance. Every Generalization instance may have zero or one Type instance as a powertype, and every Type instance may be the powertype of zero or one Generalization instances. [Relationships]

Powetype is a composite aggregation of a generalization to one type (the powertype). A type is the powertype of a generalization. [Relationships glossary]

The notation document defines it per Odell's usage:

A generalization path may have a text label in the following format:

discriminator : powertype

where *discriminator* is the name of a partition of the subtypes of the supertype. the subtype is declared to be in the given partition;

where *powertype* is the name of a type whose instances are subtypes of another type, namely the subtypes whose paths bear the powertype name. If a type

> symbol with the same name appears in the model, it designates the same type; it should be shown with the stereotype «powertype». For example, TreeSpecies is a powertype on the Tree type; consequently instances of TreeSpecies (such as Oak or Birch) are also subtypes of Tree. Either the discriminator, or the colon and powertype, or both may be omitted. Note that the word *type* also includes both types and classes. [Notation]

Also see figure 23 in the notation document. We recommend Odell's definition of powertype for the UML. Also see section 5.1.2, Types as Instances, page 13.

6.3.2 Association Role Multiplicity

Summary: the wording of the semantics document regarding association role multiplicity is unclear.

The semantics document defines the MULTIPLICITY attribute of ASSOCIATION ROLE this way:

> The multiplicity attribute of AssociationRole specifies the number of instances of a Type instance that participate in the Association instance; [Relationships]

The phrase "that participate in the Association instance" implies that the role can have multiple participants in a single association instance, which the meta-model contradicts. The multiplicity of the PLAYER meta-association between LINKROLE and INSTANCE makes this clear, as well as this definition from the notation document:

> Multiplicity for n-ary associations may be specified but is less obvious than binary multiplicity. The multiplicity on a role represents the potential number of instance tuples in the association when the other N-1 values are fixed. [Notation]

Perhaps the semantics document could say something like "The multiplicity attribute of AssociationRole constrains the total number of instances of a Type instance that may participate in that particular association role in all the instances of the Association".

6.3.3 isOrdered

Summary: The wording of the semantics document regarding ISORDERED is unclear.

The semantics document defines ISORDERED this way:

> The isOrdered attribute of AssociationRole applies if the multiplicity of the AssociationRole instance is greater than one, and means that the instances that participate in the Association instance are ordered.

The phrase "that participate in the Association instance " implies that there is some ordering on participating instances in a single association instance, or link, which we assume isn't the case. It would be clearer to say that multiple queries returning the objects participating in this role for all association instances should return the objects in the same order as long as no changes are made to the association instances between those queries.

7. Types

7.1 Modeling

7.1.1 Attribute

Summary: Attributes are described as kind of association in the text, but not in the meta-model.

Attribute is not well-defined in the description sections of the semantics document:

> The responsibility of Attribute is to specify a structural feature of a Type instance. [Types]

The term "structural" is undefined and not related to other terms. The glossary and notation adds:

> An attribute is semantically equivalent to a composite aggregation with navigation restricted to navigation from the type to the attribute. [Relationships glossary]
>
> Note that an attribute is semantically equivalent to a composition association. [Notation]

which make sense. ATTRIBUTE should be specialized from ASSOCIATION. Like associations, attributes should be generalizable (see section 6.1.1, Association and Association Class, page 19).

7.1.2 IsTypeScope

Summary: The ISTYPESCOPE attribute is defined at the programming language level, obscuring its purpose as a way of describing types as instances. A more general shorthand for such services is proposed.

The ISTYPESCOPE attribute on MEMBERS is defined this way:

> The isTypeScope attribute of Members is a Boolean specifying the scope of the associated Member instance. The default value of isTypeScope is False,

> meaning that the Member instance is instanced scoped. An instance scoped Member instance is one that is unique to the instance of the Type instance, and a type scoped Member instance is one that is shared by all instance of the Type instance.

The purpose of statically scoped C++ variables, from which we assume this is derived, is for members to be created for characteristics of types themselves, rather than for their instances. Some examples are keeping a count of the number of instances of a class, giving a unique identifier to a class, and declaring an operation that creates an instance of a class. In these examples, you can see two kinds of type level characteristics, namely those which:

1. Do not inherit to their subtypes, like a unique identifier for a class.

2. Do inherit to their subtypes, like an operation for creating instances of a type.

ISTYPESCOPE covers the first but not the second. A better model would be to replace ISTYPESCOPE with an enumerated attribute TYPESCOPE with possible values OWN, covering the first case, and SUBTYPE, covering the second case. Both cases can also be modeled by declaring members on a type's type (section 5.1.2, Types as Instances, page 13), a simple solution for the first case, but not so simple for the second case (exercise for the reader). Types as instances provide a semantic foundation for an enumerated TYPESCOPE, which is just a shorthand for that semantics.

7.1.3 Operation Inheritance

Summary: Operation inheritance is currently done by matching signatures, so it is impossible to unambiguously inherit two differently named operations on the same type with the same signature.

The semantics document describes operation inheritance like this:

> As described in section 6, Type instances may participate in Generalization relationships. Type instances that are subtypes of another Type instance inherit all of the properties of their supertypes, ... There is not explicit relationship between the Operation instance of a supertype Type instance and its corresponding Operation instances in the subtypes of the Type instance. Rather, these operations are matched by signature: Operation instances with the same signature are considered to match. [Types]

There is a clarity issue regarding the term "matching", but we assume it means a tool implementing UML can tell when an inherited operation is the same as its one on its supertype. If this is the case, matching should use the name as well signature, otherwise no two operations on the type could have the same signature.

7.2 Comprehensibility

7.2.1 Signal

Summary: the definition of signal is not comprehensible due to a contradictory reference to events and being insufficiently distinguished from operations.

The semantics document defines signal this way:

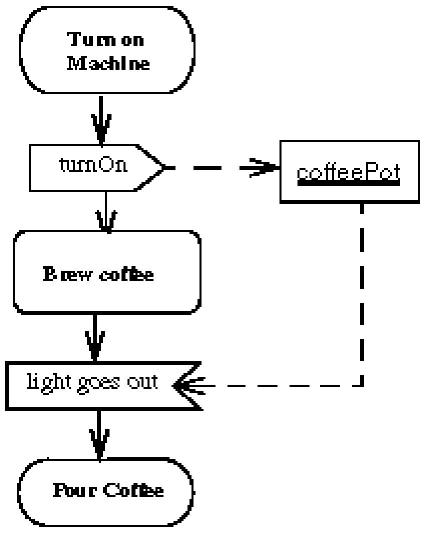
> The responsibility of Signal is to specify a named event. [Types, Classes, and Instances]
>
> The responsibility of Signal is to name a potential event representing a significant occurrence in time/space. [Types]

EVENT is a subtype of MODELELEMENT which have names, so the above definitions do not distinguish signals from events. Also Signal is not a subtype of EVENT in the meta-model. In any case, the usage of signals in the semantics document seems to be the same as for operations. Here are some examples:

> Note that Signal and Operation appear in this section connected to both Event and Action. Connected to Action, Signal and Operation represent invocations; connected to Event, Signal and Operation represent receipt. Thus, these relationships provide closure: an Event or Operation produced in one StateMachine instance may be consumed in another. [State Machine]
>
> Signal/Message, Operation/Message each form an essence/manifestation pair. [Types]
>
> It is possible to specify an explicit send Dependency from an Operation instance to a Signal instance, representing the signals that may be sent during an invocation of the operation. The source of this send Dependency is an Operation instance, and the target is a Signal instance, both of which must be members and signals, respectively, of the same Type instance. [Types]
>
> If a Signal instance S is specified as a signal received by a given Type instance, this means that the Type is obligated to respond to instances of S as well as instances of subtypes of S. [Types]
>
> An action is the invocation of a signal or an operation, representing a computational or algorithmic procedure. [State Machines glossary]
>
> Invocation is a shared aggregation of an action to an operation. The action is the invocation of the operation. Invocation is a shared aggregation of an action to a signal. The action is the invocation of the signal. [State Machines glossary]
>
> Occurrence is a shared aggregation of a call event to an operation. A call event is an occurrence of an operation. Occurrence is a shared aggregation of a signal

event to a signal. A signal event is an occurrence of a signal. [State Machines glossary]

The notation document reinforces the equation between signals and messages. For example, see figure 55 on page 113:



It has the following description:

Signal receipt. The receipt of a signal may be shown as a concave pentagon that looks like a rectangle with a triangular notch in its side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. This symbol replaces the event label on the transition. A dashed arrow may be drawn from an object symbol to the notch on the pentagon to show the sender ofthe signal; this is optional.

Signal sending. The sending of a signal may be shown as a convex pentagon that looks like a rectangle with a triangular point on one side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. This symbol replaces the send-signal label on the transition. A dashed arrow may be drawn from the point on the pentagon to an object symbol to show the receiver of the signal; this is optional.

In the receipt case, signal is equated to an event, which the notation document equates with messages (see section 10.1.3, Event, page 49). In the sending case, signal is equated with the send-event/message on transitions. By the way, the above figure shows a transition out of operation TURN ON MACHINE sending an event/message TURN ON to the object COFFEE POT, when presumably the coffee pot is already turned on by the TURN ON MACHINE activity.

Also by the way, section 5, Types, Classes, and Instances, Derived Semantics, says Signal is explained in sections 7, 10, and 11, but the section 7 glossary says it is explained in section 5.

Also by the way, the following uses of “triggers” are inconsistent with the meta-model, which uses it to relate events to transitions:

> (State machine glossary) A signal event is an event triggered by a signal.
>
> (State machine) SignalEvent is a subtype of Event. The responsibility of SignalEvent is to specify an Event instance triggered by the invocation (sending) of a Signal instance.

The only differences between signals and operations that we could find are

1. Signal is a subtype of class, whereas operation is not. So signals can have implementations, whereas operations cannot, since they are separated from methods.

2. And this interaction:

   > Dynamically, a Class instance may receive Signal instances or Operation instances, but not both. [Classes]

The first doesn’t shed light on how signals are used differently than operations. In the second case, we’re not sure what “dynamically” means here, so can’t interpret it. Since signals aren’t distinguished significantly from operations, we can’t determine their semantics. The submission should of course clarify the meaning of signals.

7.2.2 Member Direction

Summary: The operations and attributes of types are given “direction”, which is not comprehensibly defined, due to a mismatch between the apparent intention and its representation in the meta-model. In any case, it appears to be an issue for tools rather than modeling

The semantics document defines the DIRECTON attribute on MEMBER this way:

> The direction attribute of Members is an enumeration, specifying the direction of the associated Member instance. The default direction of Members is provide, meaning that the Member instance is one that is declared in the Type instance. A required Member instance is one that the Type instance requires in order to preserve its semantics. Specifying a required Member instance introduces the Member instance to the Type instance, but does not constitute a declaration of the Member instance. Unlike template parameters, the specification of required

Member instances does not introduce a template Member instance, but rather is a statement of the semantics of the Type instance's interface, in which the Member instances that it expects to use are specified.

The circular definition of the first sentence is broken by the definitions of the individual values of the meta-attribute. The definition of PROVIDED is clear, but redundant with normal member declaration. The phrase "preserve its semantics" clouds the definition of REQUIRED. That required members are "introduced" is not comprehensible because declarations are generally the only way that a member is attached to a type. The last sentence says that required members are those used by the interface of the type, and since the required members are not declared on that same type, presumably they are declared on another type, a relation which is not specified in the meta-model.

Taking all of the above as comprehensibility issues, no motivation for this aspect of the meta-model is not given. Perhaps it is related to the "provided" and wanted" roles, which are found to be unclear, and in any case, a tool issue rather than modeling (see section 5.2.1, Roles, page 5.2.1). Assuming all of this were clearly defined, it would be a matter of modeling that associations in which a type participates can be provided and required also.

8. Classes

8.1 Modeling

8.1.1 METHOD Specialized From OPERATION

Summary: The meta-type METHOD which is an implementation, is specialized from OPERATION, which is restricted not to be. This is a violation of basic subclassing semantics.

The arguments against CLASS as a specialization of TYPE apply to METHOD as a specialization of Operation (see section 5.1.1, page 12). Method cannot be a specialization of OPERATION, because methods cannot be used where types can be. This would violate the definition of generalization in the UML based on substitutability. The semantics document admits the inconsistency, but does not fix it:

> Class is a subtype of Type, and therefore instances of Class have the same properties as instances of Type, the fundamental difference being that Type instances specify interfaces, whereas Class instances specify the realization of these interfaces. An implication of this dichotomy is that a Type instance may only have Operation instances as members, whereas Class instances may only have Method instances as members (even though Method is a subtype of Operation and hence Method instances are semantically substitutable). [Classes]

METHOD and OPERATION should be siblings in the meta-model, or at least cousins, with a meta-association between them for specifying the method implementing an operation, instead of the vaguely defined “refinement” notion.

9. Collaborations

9.1 Comprehensibility

9.1.1 General

Summary: the concept of collaboration is obviously important to the UML, because it connects both use cases and behavior to types and operations. However, these connections are so vaguely drawn out that the current text cannot serve as a comprehensible specification.

The description of collaboration raises issues on the border of clarity and comprehensibility. From the references scattered through various chapters, and a significant amount of extrapolation, we might with great effort put together a coherent story of collaborations. This distance from that story to the current text is too much to consider this simply a clarity issue, especially considering the importance of collaborations to specifying use cases and behavior.

For example, take the references to "inside" and "outside" specification:

> Applying a Collaboration instance to a Type, Class, or a subtype of Class instance permits a statement of the semantics of that Element instance. This may be an outside view - specifying the meaning of the Element instance without proscribing its realization - as well as an inside view - specifying the behavior of its realization. In the former case, the structural dimension of the Collaboration instance would contain Element instances that describe the vocabulary of that Type. In the latter case, the structural dimension of the Collaboration instance would contain Element instances that draw from the Member instances and neighbors of that Type. Furthermore, the behavioral dimension of the Collaboration instance permits UseCase instances to be associated with a Type instance (and its subtypes, although in most cases this only applies to Class): as described above, a Collaboration instance may have Type instances and as described in section 5, UseCase is a subtype of Type, hence, a Collaboration instance may have UseCase instances. A UseCase instance associated with a Type instance must conform with any lower level UseCase instances that might be attached to parts of the Type instance. [Collaborations]

We understand the general notion that some specifications are about the interaction of a type with its environment, the requirements or use cases, and others are how those environmental specifications are satisfied, the implementation. However descriptions like "the vocabulary of that Type" and "the behavioral dimension of the Collaboration instance permits UseCase instances to be associated with a Type instance" do not explain how collaborations are used to achieve this. The other discussions of this topic in the Types section are only slightly more illuminating:

> As described in section 5, a UseCase instance specifies a set of scenarios, where a scenario is a sequence of actions. Thus, a UseCase instance (not to be

> confused with the Instance instance corresponding to the UseCase instance) is represented by a Collaboration instance that in turn has a collection of Behavior instances that specify all of the potential scenarios of the UseCase instance. As described in section 5, its corresponding Instance instance is a scenario, which is a single flow of actions matching this potential flow of actions. [Types]
>
> In the case of Operation instances, the semantics of the operation may be expressed in a Collaboration which includes both the participants of behavior as well as the dynamic semantics of their collaboration, either specified by StateMachine instances or reflected by Interaction instances. [Types]

These point in a general direction, but it's too long a road to leave for vendors to end up a different destinations for lack of a good map.

The particular application of collaborations to swimlanes is ad hoc modeling:

> Behavior instances may include clusters specified by Collaboration instances. These Collaboration instances are typically used to specify "swimlanes" in the corresponding ViewElement instance. [State Machines]
>
> In this context, Collaboration instances that serve as the clusters of a Behavior instance are typically used to specify "swimlanes" in the corresponding ViewElement instance. [Interactions]

Figure 53 of the notation document shows clusters for the activities of a customer, sales, and in a stockroom. These classifications have much more direct semantic value than a "cluster", which vaguely defined this way:

> Clusters is a composite aggregation of a Behavior instance to a collection of Collaboration instances. The responsibility of clusters is to specify interesting groups of Instance instance and Link instances in the context of a Behavior instance. [Interactions]

The concept "interesting group" is not defined Activities of a customer, sales, and work in a stockroom would be much better modeled by using normal classification techniques to create types of behavior according to whatever constraints the "swimlanes" represent.

By the way, the term "collaboration" is used inconsistently for both a general semantic mechanism and the notation diagram for interactions. This terminological problem, located as it is at the highest level of the semantics and notation documents, is very confusing to the first-time reader.

The notation document seems to have a different view of collaborations (see figure 32):

> A collaboration (as a complete entity representing a design pattern) is shown as a dotted ellipse containing the name of the pattern. A dotted arrow is drawn from the collaboration symbol to each of the objects or classes (depending on whether it appears within an object diagram or a class diagram) that participate in the collaboration. Each arrow is labeled by the *role* of the participant. The roles correspond to the names of elements within the context for the collaboration;

> such names in the collaboration are treated as parameters that are bound to specify elements on each occurrence of the pattern within a model. [Notation]

In addition to yet another confusing use of the term “role”, the definition of parameters in the semantics document doesn’t mention that they can be substituted for types and instances in the collaboration:

> The Name instances of the Parameter instances become names that are visible to the template Collaboration instance and can be used in the scope of the template in a manner that conforms to the type of the Parameter instance. [Collaborations]

Related to the notation’s use of roles is the semantics document’s definition for pattern as a template collaboration. All this suggests a connection to contextual aggregation, which is poorly defined in the UML (see section 6.1.2, page 21).

10. State Machines

10.1 Modeling

10.1.1 States

Summary: State is a concept fundamental to object-orientation but is so poorly defined in the UML that it is not comprehensible. A definition is proposed from existing work which should have been consulted prior to submission.

The term “state” is not well defined in the semantics document:

> A state is the condition of an instance at a given moment in time/space. [Types, Classes, and Instances glossary]
>
> State is the condition on an Instance instance at a given moment in time/space. The responsibility of State is to reify a state. [Types, Classes, and Instances]

Both definitions rely on the word “condition” which is not defined or related to other terms. The notation document is clearer in that it ties the notion of state to attributes:

> Attributes in a type define the *abstract state* of the type. These represent the state information supported by objects of the type, but an actual class implementing the type may represent the information in a different way, as long as the representation maps to the abstract attributes of the type. Type attributes can be used to define the effects of type operations. A type may contain specifications for operations, including their signatures and a description of their effects, but the operations do not contain implementations. The effect of an operation is defined in terms of the changes it makes to the abstract attributes of the type. [Notation]

Probably a better term for the above idea would be *complete-state*, because users will not make a state for each possible combination of all attribute values. For example, a type PERSON with an AGE attribute may only have the states YOUNG, MIDDLE-AGED, and OLD. Each state covers many values of the AGE attribute compounded with the many possible values of other attributes unrelated to AGE. So each state covers many complete-states. Because they are very specific, complete-states can serve as a foundation for the semantics of states and state machines.

Even if complete-states are not modeled, the state of an instance deserves separate representation from the states of its type. For example, suppose the type PERSON has the states WELL and SICK, and we want to keep track of how many times each person gets sick. If we model WELL and SICK as types, with attribute HOWMANYTIMES on SICK, then we can record the sick-time for a particular person on its associated instance of SICK. Basic aspects of this technique are described in the semantics document:

> State instance is a shared association of an Instance instance to its state. The responsibility of state instance is to specify the static State instance of an Instance instance. Every Instance instance is associated with zero or one State instances and every State instance is associated with zero or more Instance instances. The state of an Instance instance must match with the possible states of the Type instance of which the Instance instance is an instance. [Types, Classes, and Instances]
>
> Between each Type instance and its Instance instances, the values, actions, state instance, and roles of the Instance instance must match the attributes, actions, states, and roles of the associated Type instance. ... Matching a state instance to a State means the state instance must be an instance of one of the potential states of the Type instance. . [Types, Classes, and Instances]

However, the meta-model is not in line with the text in a couple ways:

1. The STATE meta-type is not specialized from TYPE, so it can't have instances.

2. There is no state instance meta-type. The STATE INSTANCE meta-association connects INSTANCE to STATE, rather than to a state instance meta-type.

These changes to the meta-model would support the modeling technique described here and in the semantics document (see our meta-model in section 10.1.3, Event, page 49, and for further information see Object-oriented Methods, A Foundation, by Martin and Odell). Perhaps these changes were omitted intentionally, but the only possible reference we can find is this uninterpretable paragraph:

> Note that that State instance associated with an Instance instance represents an occurrence of the State instance. This manifestation of a State instance is not made explicit, but rather is a consequence of the semantics described in section 10, wherein State is a part of StateMachine which is a kind of Behavior, and Behavior/BehaviorInstance provide an explicit essence/manifestation dichotomy. [Types, Classes, and Instances]

The term "occurrence" is not defined, and terms like essence/manifestation are too vague to be useful.

Association participation part of complete-state

The notation's definition of "abstract state" of an object should include participation in associations, as supported by UML's definition of attribute:

> Note that an attribute is semantically equivalent to a composition association. [Notation]
>
> An attribute is a structural feature of a type. An attribute is semantically equivalent to a composite aggregation with navigation restricted to navigation from the type to the attribute. [Relationships glossary]

Since non-attribute associations only differ in composition and navigation from attribute associations, all associations should be included in complete-states. For example, the participation of a particular person in the association MARRIAGE can be considered part of that person's complete-state. The same techniques described in the last subsection apply, namely a state MARRIED for PERSON is a type that can be instantiated for each person, if necessary, to record characteristics like how many times each person has been married.

By they way, following Odell, the classification of an object should also be considered part of its complete-state (see Object-oriented Methods, A Foundation, Martin and Odell).

States just for state machines

An alternate but compatible semantics for state is suggested in the notation document:

> A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. An object remains in a state for a finite (non-instantaneous) time. [Notation]

The phrase "waits for some event" raises the question of whether the purpose of a state is just to describe the behavior state machines, and not for describing attribute values. To be specific, we might imagine a state machine that has two or more states with identical complete-states. This is clearly possible because operations do not need to modify the complete-state of an object if they are just queries or computations:

> The isQuery attribute is a Boolean and specifies whether or not the Operation instance is a behavior that preserves the state of the instance. The default value of isQuery is False, which means that the semantics of the Operation instance allow the state of the instance of the Type instance to be modified. A value of True means that the semantics of the Operation must guarantee that the state of the instance of the Type instance not be modified. [Types]

For example, a state machine might do nothing but control which queries can be asked when. The association-based model of states still works because it is not required that a complete-state belong to exactly one state instance at a time, so an object can change state without changing complete-state. Of course, the implementation level requires some record on the object of what state it is in to drive the state machine, but this will not appear in the "abstract" attributes defined on a type.

10.1.2 State Variables

Summary: state variables as defined are a view issue, not a modeling issue, and are incomplete in any case.

Please read section 10.1.1 (States, page 45) first.

The STATE VARIABLE meta-association between STATE and ATTRIBUTE resolves some of the ambiguity of the state variables definition in the semantics document:

> The responsibility of state variable is to specify attributes of the state. [State Machines]

The preposition "of" could be interpreted to mean the state itself has characteristics. The ATTRIBUTE meta-type is associated to TYPE, from which STATE is not specialized, so we know state variables do not refer to characteristics of a state. The notation document is clearer and adds more information that should be included in the semantics document:

> State variable are attributes of the owning class but are distinguished because they are affected by or used by actions in the state diagram. [Notation]

However, the notation document is describing a view issue, rather than modeling, because listing the attributes affected by an action is presenting some of the information already existing in actions. A modeling definition would be, for example, that the attribute values available as actual arguments for actions are restricted to those from attributes listed as state variables. The documents should of course be explicit and consistent on these questions.

Association participation part of state variables

As described in section 10.1.1, the state of an object may include its participation in associations, of which attributes are a specialization (see section 7.1.1, Attribute, page 36). If the semantics of state variables is resolved, the STATE VARIABLE meta-association should associate STATE to ASSOCIATION rather than ATTRIBUTE. By they way, following Odell, the classification of an object should also be considered part of its state variables (see Object-oriented Methods, A Foundation, Martin and Odell).

10.1.3 Event

Summary: Event is a concept fundamental to object-orientation but is so poorly defined in the UML that it is either not comprehensible or confused with other terms. A definition is proposed from existing work which should have been consulted prior to submission.

Please read section 10.1.1 (States, page 45) first.

The term “event” is not well defined in the semantics document:

> The responsibility of Event is to specify a significant occurrence in time/space. [State Machines]

The terms “significant” and “occurrence” are not defined or related to other terms. The notation document describes it this way:

> An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. [Notation]

but then identifies them with messages:

> A *message flow* in the notation that shows the sending of a message from one object to another. The implementation of a message may take various forms, such as a procedure call, the sending of a signal between active threads, the explicit raising of events, and so on. [Notation]
>
> A final (pseudo)state is shown as a circle surrounding a small solid filled circle (a bull’s eye). It may be labeled by a *send-event-expression*; if so, it represents the occurrence of an event at the level of the enclosing state. (In effect, reaching the state causes a subsequent transition on the enclosing state.) If the state is unlabeled, it represents the completion of activity in the enclosing state which triggers any transition on the implicit activity completion event. [Notation]
>
> A transition is shown as a solid arrow from one state (the *source* state) to another state (the *target* state) labeled by a *transition string.* The string has the following format:
>
>> *event-signature* ‘[’ guard-condition] ‘/’ action-expression ‘^’ send-clause
>
> The *event-signature* describes an event with its arguments:
>
>> *event-name* ‘(‘ *parameter* ‘,’ . . . ‘)’
>
> The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. [Notation]
>
> ‘The *send-clause* is a special case of an action, with the format:
>
>> *destination-expression* ‘.’ *destination-event-name* ‘(‘ *argument* ‘.’ . . . ‘)’

> The transition may contain more than one send clause (with delimiter). The relative order of action clauses and send clauses is significant and determines their execution order.
>
> The *destination-expression* is an expression that evaluates to an object or a set of objects. The *destination-event-name* is the name of an event meaningful to the destination object(s).
>
> The *destination-expression* and the arguments may be written in terms of the parameters of the triggering event and the attributes and links of the owning object. [Notation]
>
> See Section 8.5 for the text syntax of sending messages that cause events for other objects. [Notation]

Events are not the same as messages, or even related by specialization, as explained by Martin in his OTUG discussion with Schauer:

> There seem to be no difference between "messages" of interaction diagrams and "events" of state diagrams, except that events don't allow synchronization. Harel's Statechart notation, on which the UML state diagram notation is based, uses the term "event". Object-oriented techniques use the term "message". To make things easier, shouldn't the UML unify these two terms expressing the same concept? [Reinhard Schauer]
>
> I don't think the two concepts are particularly close. An message is a function invocation, it takes arguments and returns values. It takes time. An event takes no arguments, does not return a value, and is instantaneous. Messages cause methods in the receiving object to be invoked. Events *may* invoke actions either on the new state, or on the transition between states. I think the two concepts are different enought to prevent unification. [Robert Martin]

In addition, messages are directed towards a specific object to invoke their methods, while events are not directed at any object. Events are changes in an object that happen without concern for their effect. Any other object can choose to react to an event, but this is unbeknownst to the event itself.

Messages may be used as an implementation, but the OMG standard should not dictate such implementations, of course. The implementation proposed by the UML, which seems to be the same as “callback” operations on eventful objects, is not the only one or even the best one. The OLE programmer passes message pointers directly to eventful objects for later notification. Better implementations have a “clearinghouse” that accepts subscriptions to events and notifies subscribers, thereby keeping eventful objects completely unaware of other objects interested in their changes. A similar opinion regarding implementation-independence is expressed by the UML regarding signals:

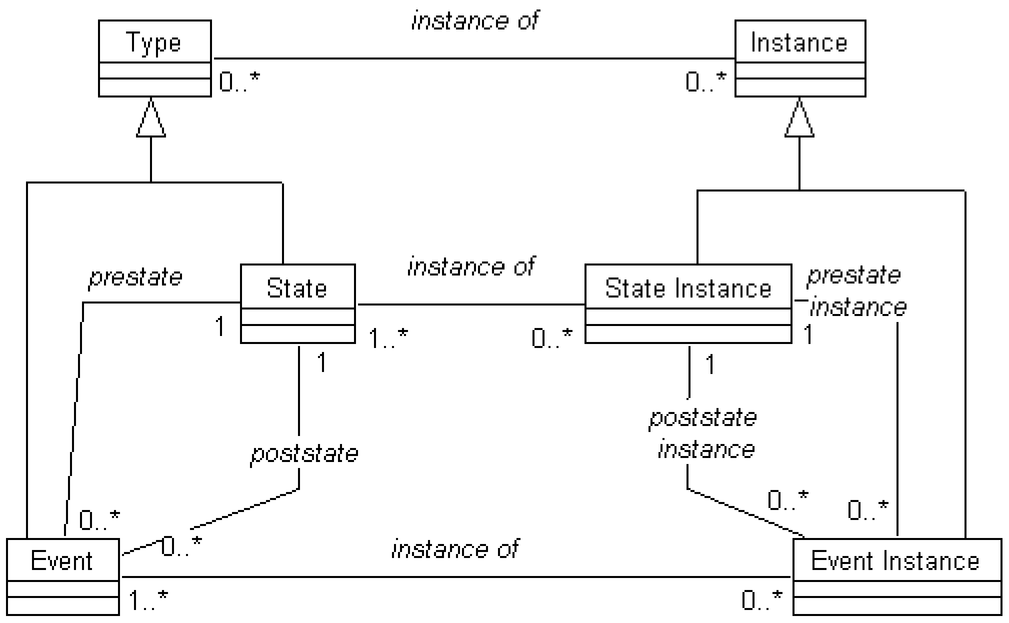
> The UML does not specify the underlying mechanism whereby a Signal instance is broadcast to a Type instance nor how a Type instance receives or sends a Signal instance. [Types]

The same policy should be adopted regarding events.

Events as changes in state

Since events are changes in objects that may or may not be noticed by other objects, they should be modeled simply by the states before and after a change occurs, rather than any interaction with other objects. EVENT should be specialized from TYPE, with two meta-associations to STATE, one for the pre-state, and one to the post-state. Each instance of event is then associated with two state instances that give the specific change in state of the instance. For example, an instance of the JUST-MARRIED event type on an instance of PERSON will be associated with two state instances, one an instance of UNMARRIED and the other an instance of MARRIED.

Here's a meta-model for states and events (see Object-oriented Methods, A Foundation, by Martin and Odell for more information):



The particular kinds of events listed in the UML, for signal and operation receipt, and time changes, can be specialized from the above definition of event. Signal and operation receipt are changes in state of the "operating" system that implements message-passing, and time events refer to changes in state of a virtual or physical clock.

10.1.4 Non-determinism in Transitions

Summary: In a couple of cases, non-determinism is introduced into state machines where it is not appropriate.

There are a couple cases where non-determinism is introduced into state machines where it is not appropriate:

> Second, consider the case where a State instance is the source of several Transition instances, more than one of which has no trigger. The guards of each of these triggerless Transition instances are evaluated exactly once, in a nondeterministic order not specified by the UML. ... If more than one of these guards evaluates True, the execution of the StateMachine instance is nondeterministic: execution will proceed as above, following only one of the triggerless Transition instances whose guard evaluated True, the choice of which is implementation dependent. [State Machines]
>
> Third, consider the case where a State instance is the source of several Transition instances, all of which have triggers. Execution of the StateMachine instance proceeds, waiting in the State instance for a pending Event instance. Upon receipt of an Event instance, execution continues with an evaluation of all Transition instances for which that State instance is a source and whose trigger is that Event instance. This yields three possible cases. First, ... Second, ... Third, consider the case where there is an Event instance, and more than one Transition instance triggered by that Event instance. The guards of these Transition instances are evaluated in an order not determined by the UML. ... If more than one of these guards evaluates True, the State instance is left and execution of the StateMachine instance proceeds as above, following only one of the Transition instances triggered by the Event instance, the choice of which is implementation dependent. [State Machines]

In each of the above cases, the state machine chooses one transition non-deterministically from multiple transitions. This model forces users to carefully check that their guards will never be satisfied at the same time, under penalty of unpredictable behavior at runtime. Why not let the user indicate whether they expect the guards to be disjoint, so that code generators can emit runtime checks? Such an indication could be the default. If the user indicates non-disjoint guards, then the successful transitions just spawn their own threads, which users might want. The submission should provide some motivation for its proposal.

10.1.5 Interruptability

Summary: The semantics and notation documents are not in agreement about whether actions are interruptible.

The semantics and notation documents are in disagreement on whether actions in states are interruptible. The semantics document says actions, or effects, on transitions are not interruptible, and that the actions in states are just the actions of the internal transitions:

> The evaluation of an effect is not interruptible; all Action instances run to completion. [State Machines]
>
> Since State instances are not interruptible, this means that any Event instances that are invoked during the execution of an Action instance are queued. [State Machines]

The Action instance associated with a do transition is in fact not interruptible; however, the implementation of an Operation instance associated with the do Action instance may periodically check to see if there are pending events (using the predefined _hasEvent operation), and if there are events, complete its processing so that the event can be handled. Note that these semantics do not specify implicit interruptability - the StateMachine instance runs-to-completion, but the modeler may specify points where the presence of an event is tested. [State Machines]

Internal transitions are just Transition instances, and so may have guards, triggers, and effects. [State Machines]

If there is an entry internal Transition instance associated with the State instance, its guard attribute is evaluated. If this guard evaluates to True, then the effect of the entry internal Transition instance is evaluated. After completion of the evaluation of this effect, the StateMachine instance is said to be in this State instance. Execution of the StateMachine instance waits until conditions arise that cause a state transition, as described below. At this point, if there are is a do internal transition, the Action instance associated with the do transtion is begun. [State Machines]

The notation document says only do actions are interruptible:

An internal "do" action is an ongoing process performed while the object is in the given state. The action need not be atomic; it is interruptible by outside events. It is initiated when the state is entered (after any incoming transition actions and entry actions). It may terminate by itself, in which case the termination represents an implicit "action complete" event. Otherwise it is externally terminated whenever the state is exited (before any exit action or outgoing transition actions). Nested state machines are equivalent to do-actions. [Notation]

[When explaining non-internal transition notation] The *action-expression* is a procedural expression that is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the owning object and the parameters of the triggering event. The action-clause must be an atomic operation, that is, it may not be interruptible; it must be executed entirely before any other actions are considered. The transition may contain more than one action clause (with delimiter). [Notation]

The best resolution would be to let the user decide when an action is interruptible or not. Since it is not practical to allow transition actions to be interrupted, because the machine would be caught "between states", this could be limited to the actions, or even just the do action, of internal transitions.

10.1.6 Activity Diagrams

Summary: Control-flow, a concept very popular in business modeling, is squeezed into the UML through a view on state machines, rather than understood as a modeling issue.

Consequently a fundamental aspect of traditional control-flow diagrams is not handled and features of modern control-flow diagrams are omitted entirely.

The notation document describes activity diagrams, a form of control-flow diagrams, as a view onto state machines (section 9). This strategy causes a number of serious problems. To begin with, the control flow diagram user normally expects to invoke reusable activities, calculating inputs values for these activities according to the particular context of their use. However, basing control-flow on UML state machines forces the user to calculate inputs to an activity as part of the action expression it internal "do" transition. So each activity is not a reusable entity, but is made specific to the context in which it is used. For example, two transitions into the same activity cannot calculate the inputs of that activity in different ways. By comparison, in OOIE's Event Diagrams, input values are calculated on the transition into an activity, so that the activity itself is context-independent (see Object-oriented Methods, A Foundation, by Martin and Odell).

Modern forms of control flow, like OOIE's Event Diagrams, show the goals reached by each activity, so the user can understand why an activity is invoked by the objects it is affecting. The user can also see what object changes are triggering the next activity. Activity diagrams, on the other hand, omit these by assuming there is nothing interesting about the termination of an activity except the termination itself:

> Transitions leaving an action state should not include an event signature; such transitions are implicitly triggered by the completion of the action in the state. The transitions may include guard conditions and actions. [Notation]

So for example, in figure 50 of the notation document, GET CUPS is not tied to the specific events and objects reflecting the goal of getting cups. One consequence is that it appears there is no way to pass the return values of one activity to the input values of the next, like passing the cup gotten from GET CUPS to POUR COFFEE. Though there must be a way to do this in the UML (we couldn't find it in the documents, by they way), the point is that the activity diagrams don't present this information. In Event Diagrams return values are stored in the goals of each activity, which have their own notation.

The problem above also applies to using goals and events for decision-making, which activity diagrams relegate to transition guards. In particular, the subtyping of goals and events provided by OOIE, along with disjointness and completeness, is completely obscured by using guards. The user cannot specify whether guards are complete or disjoint coverings of the goals attained by an operation (see section 10.1.4, Non-determinism in Transitions, page 51). This further severs the ties that modern control flow diagrams have to the objects they affect and are affected by, which is a foundation of current business modeling practice.

For decades it has been recognized that there are three kinds of process model: control-flow, data-flow, and state machines. Any two of these dimensions can be reduced the third, of course. For example, data-flow can achieve the same effect as control-flow by passing "control tokens" to explicitly order the execution of activities. Control-flow can achieve the same effect as state machines using wait loops for detecting events before proceeding. But there is hardly any point in such a reductionist exercise. It just puts unnecessary burden on the user to squeeze one model into another. In the case of UML state-machines, the user is forced to create the machinery for calculating inputs on transitions, and expressing goals and event partitions. Users will be much happier with a control flow standard that handles it as a first-class modeling issue.

10.2 Clarity

10.2.1 Pseudostates

Summary: The semantics document is inconsistent on whether the initial pseudo state is required.

The semantics document is inconsistent on whether the initial pseudo state is required:

> There must be exactly one initial Pseudostate instance that is immediately part of a StateMachine instance, as well as exactly one initial Pseudostate immediately part of every CompositeState instance. [State Machine]
>
> Final and history Pseudostate instances are optional; each StateMachine and CompositeState instance need not include any Pseudostate instances. [State Machine]

We assume the second clause in the second item is just written incorrectly.

10.2.2 Internal Transitions

Summary: Guards on internal transitions are not handled in the notation.

Notation does not include guards in internal transitions syntax:

> event-name argument-list / action-expression

but semantics says:

> Internal transitions are just Transition instances, and so may have guards, triggers, and effects. [State Machines]

Compare the notation for simple transitions:

> event-signature `[' guard-condition `]' `/' action-expression `^' send-clause

We assume guards on internal transitions were inadvertently left out of the syntax.

10.2.3 Complex Transitions

Summary: The notation document leaves out the triggering event in its description of complex transitions.

The notation document leaves out the triggering event in its description of complex transitions:

> If the owning object is concurrently in all of the source states of a transition, then the transition is enabled. If the guard condition for the transition is true, then the transition fires and performs its actions. Then the object ceases to be in all of the source states and becomes in all of the target states. Normally this involves crossing out of or into a concurrent state region. [Notation]

We assume complex transitions are triggered in the same way simple transitions are, per the meta-model.

11. Interactions

11.1 Modeling

11.1.1 Qualification

Summary: qualification of link roles records information redundantly.

The semantics documents defines the QUALIFICATION meta-association between LINKROLE and VALUE this way:

> Qualification is a composite aggregation of a LinkRole instance to a collection of Value instances. The responsibility of qualification is to specify the values for the LinkRole instance corresponding to any Attribute instances of the corresponding AssociationRole instance. The Value instances of a LinkRole instance must match the corresponding Attribute instances in order and in type. [Interactions]

It doesn't seem necessary to record the qualifier values redundantly on the link role instances when the combination of qualifier attributes on the association role and the participant of the link role is enough to calculate the value.

11.2 Consistency

11.2.1 Role Instance

The semantics document defines ROLE INSTANCE meta-association between LINKROLE and ASSOCIATIONROLE like this:

Role instance is an association between a LinkRole instance and an AssociationRole instance. The responsibility of role instance is to specify that the LinkRole instance is an instance of the AssociationRole instance. [Interactions]

Association roles are not types, however, so they can’t have instances.