

Foundations of the Unified Modeling Language

Tony Clark, Andy Evans
Formal Methods Group,
Department of Computing,
University of Bradford, UK

Abstract

Object-oriented analysis and design is an increasingly popular software development method. The Unified Modeling Language (UML) has recently been proposed as a standard language for expressing object-oriented designs. Unfortunately, in its present form the UML lacks precisely defined semantics. This means that it is difficult to determine whether a design is consistent, whether a design modification is correct and whether a program correctly implements a design.

Formal methods provide the rigor which is lacking in object-oriented design notations. This provision is often at the expense of clarity of exposition for the non-expert. Formal methods aim to use mathematical techniques in order to allow software development activities to be precisely defined, checked and ultimately automated. This paper aims to present an overview of work being undertaken to provide (a sub-set of) the UML with formal semantics. The semantics will facilitate the use of the UML in the software development process by allowing development steps to be defined and checked.

1 Introduction

The ‘Unified Modeling Language’ (UML) [2] is a language for modelling object systems based on a unification of Booch, Rumbaugh and Jacobson’s popular object-oriented modeling methods. It is rapidly emerging as a de facto standard for the modelling of such systems. The UML provides many of the diagrammatical modelling techniques found in most modern object methods, such as object diagrams, state diagrams and object interaction diagrams. However, the aim of the UML is to provide a standardised, conformant language, which can be used in preference to the plethora of notations, diagrams, and other presentation conventions that have emerged in the object-oriented and structured methods domain in recent years.

There are good reasons for believing that the UML may achieve its aims of providing a standard notation for OO. Already, there is significant and widespread momentum growing within industry towards its adoption: most CASE vendors are already tailoring their tools to support it, whilst much of the new OO literature is seeking to make itself “UML compliant”. In addition, the UML has recently been submitted to the Object Management Group as a candidate for standardisation, where it is widely believed it will be accepted.

One interesting aspect of the UML, is the recognition by its authors of the need to provide a precise description of its semantics. Their intention is that this should act as an unambiguous description of the language, whilst also permitting extensibility so that it may adapt to future changes in object-oriented analysis and design. The approach used to describe these semantics is to give a meta-model of the UML. Thus, UML notations are used to describe UML semantics. In the present version (1.0), the meta-model consists of five core UML concepts: Common concepts (basic types); Structural Elements (types and relationships); Behavioural Elements (state machines and interactions); View Elements and Standard Elements.

By recognising the importance of formality the UML authors have clearly made a revolutionary step forward; never before has rigour played such an important part in the development of an industrial modelling method or language. Given the standing of the UML and its authors, not only are the semantics likely to be widely used and accepted as a consequence, it also likely to encourage the future use of such descriptions as a part of all method development.

Another advantage of providing the UML with a formal description is that may also provide the basis for the introduction of other valuable verification and validation techniques previously only enjoyed by formal specification

notations such as Z, B and VDM. These are some of the ways in which the UML could benefit from a formal foundation:

Clarity: To act as a reference - if at any point, there is confusion over the exact meaning of a particular UML component, reference can be made to the formal description to verify its semantics;

Equivalence and Consistency - To provide an unambiguous basis from which to compare and contrast the UML with other techniques and notations, and for ensuring consistency between its different components

Extendability: To enable the soundness of any extensions to the UML to be verified (as encouraged by the UML authors);

Refinement: To allow correctness of design steps in the UML to be verified and precisely documented. In particular, it should enable *design patterns*¹ to be checked for correctness. Once checked, a particular pattern can be used again and again without having to re-check it;

Proof: To allow justified proofs and checks of important properties of a system described in the UML, for example safety and liveness properties. Within an industrial context, this could form a basis for automatic proof techniques.

Unfortunately, the current UML semantics are not sufficiently formal to realize many of these benefits. Although much of the syntax of the language has been defined, and some static semantics given, dynamic semantics are mostly described using lengthy paragraphs of often ambiguous informal English, or are missing entirely. Furthermore, little consideration has been paid to important issues such as proof, compositionality and rigorous tool support. Another problem is the large scope of the language, all of which must be dealt with before the language is completely defined.

Given the UML's intended role as a modelling notation standard it is imperative that it has a well-founded semantics. Only once such a semantics is provided can the UML be used as a rigorous modelling technique. Moreover, the formalization of UML constructs can lead to a deeper understanding of OO concepts in general, which in turn can lead to the more mature use of OO technologies. Such insights can be gained by exploring consequences of particular interpretations, and by observing the effects of relaxing and/or tightening constraints on semantic models.

The aim of this paper is to present recent work, embarked on in a collaborative project, on the formalization of the UML. Part of the long term objectives of this project are to develop a *reference manual* for the UML. This will give a precise description of core components of the language and provide inference rules for analyzing their properties. This will then form a foundation for answering more general questions regarding the rigorous use of the UML by:

- providing a set of sound rules for manipulating UML diagrams in a precise and formal way;
- determining when a particular UML diagram is a valid refinement of another diagram;
- permitting non-trivial proofs of UML diagrams. These proofs should be automat-able using a model checking tool or theorem prover;
- offering compliancy with other emerging industry standard models, such as the Core Object Model.

This paper presents an initial attempt to provide a suitable formal model for the UML based on the identification of some of the core UML concepts. The next section begins by exploring and introducing the UML in an informal way to determine its underlying semantic foundation. A formal description is then presented in section 3. Issues surrounding the development of the formal model are then presented and other work reviewed.

2 An Informal Tour of the UML

The Unified Modelling Language (UML) is defined as a collection of graphical models which express different properties of an object-oriented design. The two most important types of model are the *structural* and *behavioural* models.

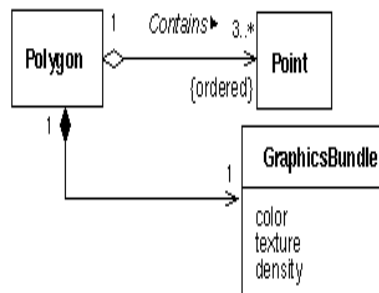
¹Design patterns are characteristic structures of classes or objects which can be reused to achieve design goals in a very elegant manner.

2.1 Structural Models

The structural model expresses information about the structure of classes and objects (instances) within the system, such as their relationships to one another, their attributes and their constraints. The most important structural model, and indeed the central model of the UML, is the *class diagram*. A class diagram is a two dimensional graph containing the following components:

- a collection of *types* or *classes*² from which a set of object instances can be drawn.
- a set of distinct *values* that the instances of a class may take.
- descriptions of the operations owned by classes and their parameters.
- a set of *association* relationships which link and constrain the cardinality of instances.
- *generalisation* hierarchies between classes or types. This relationship allows new classes to be defined as extensions of existing classes. A class which is extended is referred to as a *super-class* and the class which is the extension is referred to as its *sub-class*. Instances of a class may also be viewed as instances of its super-classes.

The following is given as a small example of a class diagram, in which the classes *Polygon*, *Point* and *GraphicsBundle* are related to each other by two associations, one of which is named. Here, the named association *Contains* relates exactly one instance of *Polygon* to three or more (ordered) instances of *Point*. The unfilled diamond denotes that the association is an *aggregation* and represents a whole/part relationship between the classes. The other association is a composite association, which once created cannot be changed.



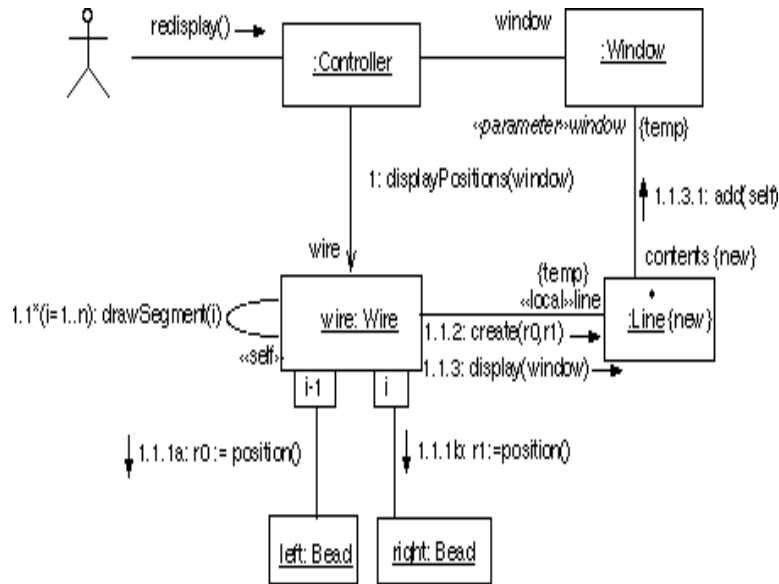
Essentially, a class diagram serves to constrain the possible collection of legal system configurations. Any program which implements the design must be in a legal system configuration at all times.

2.2 Behavioural Models

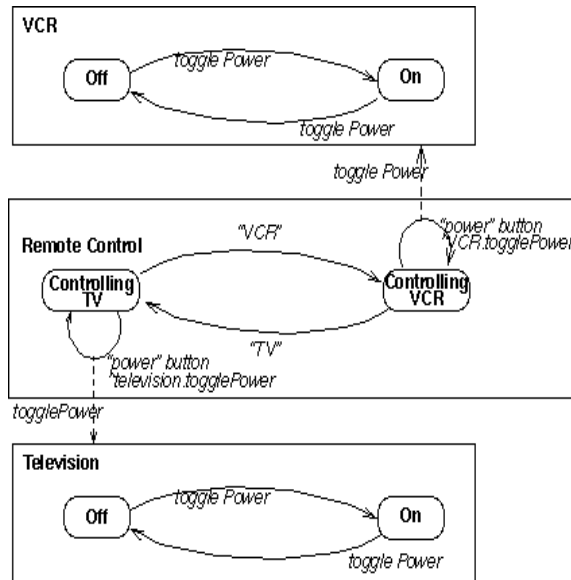
The Behavioural model expresses information about the *dynamic* behaviour of the program which is being designed. Typically, this model describes behaviour in terms of state change and message passing, which must occur in response to some system event. There are two important types of dynamic model: *collaboration diagrams* and *state transition diagrams*; together these diagrams express the required behaviour of the system.

A collaboration diagram expresses information about the *message interactions* which occur between *object instances* as viewed by an external observer. It is an external view because it does not define the internal state changes which occur at each of the collaborating objects. Typically, a collaboration diagram will express a sequence of messages which will occur due to one of the participating objects receiving a particular message. Messages may only be sent between objects which are somehow related. For example, the following diagram gives an example of a collaboration diagram for part of a diagram editor. Here, messages between instances are denoted by arrows. The messages that implement an operation are numbered along with the operation they implement.

²The distinction made between types and classes in the UML is that a class represents a “realisation” of a type. However, for our purposes it is sufficient to assume that types and classes are semantically equivalent - a class inherits all properties of a type. The terms ‘instances’ and ‘object instances’ are also used interchangeably.



A state transition diagram expresses information about the internal state changes which occur at a particular class of objects. At any instant in time, an object is in a particular state and may receive a message from another, related, object. The particular state and message will determine the action which is taken by the receiver and its new state. Again, the following diagram gives a simple example a typical UML state diagram with message passing:



2.3 Core Semantics

Given the above models, it is possible to informally identify some of the core components of the UML model. These will be examined in a more formal context later in the paper. At the core of a UML design is the description of a collection of classes, their instances and the relationships between them. These relationships will correspond to a number of distinct components of a concrete implementation including:

- the relationship between an object instance and its value.

- the relationship between a class and its instances and a class and its super-classes.
- the relationship between instances which are involved in an association
- the relationship between a class and the operations it offers
- the relationship between an operation and its signature (parameters)
- the relationship between a class and its behaviour. This can be expressed in terms of the effect the operation has on the values of its instances and the messages it passes.

Legal states are determined by a number of constraints which prevent a state from expressing an impossible situation. These constraints are:

1. messages may only be sent between objects which are related. Intuitively, this corresponds to the restriction that, in order for one object to send another object a message, the sender must be able to reference the target. The reference may occur through an attribute, through a method parameter, or through an association.
2. all objects must be the instance of at least one class. An object may be the instance of multiple classes when they are related by inheritance.
3. all instances of the same class must have values belonging to the set of possible values of the class.

Overall, UML designs are inherently state based. Each state contains information about object instances and their current attribute values. As execution proceeds, the values of the attributes may change as a result of messages being passed between objects and operations being executed. A state change, or transition, can occur as a result of the following:

- the value of an object is changed. In this case the relation defining the object's current value must be changed.
- a message arrives at an object. In this case there may be parameter values which are related to the target of the message for the duration of the message activation.
- an object is created. In this case the object must be related to its initial attribute values.
- an object is destroyed. In this case the object must be removed from all the relations in which it participates.
- a reply is sent. In this case the parameter values are (possibly) no longer related to the target of the message and the sender of the message becomes related to the reply value.

3 Formalization

This section presents work currently being completed by the group on developing a usable formal model for the UML. Its aim is to answer the question: what is the simplest, smallest and most convincing formal model, that captures the essential properties of the UML? The model presented is hopefully a step in this direction. It is intended to make it small and simple by recognising that many of the modelling techniques used by the UML simply present different 'views' of the same underlying object model. For example, object interaction diagrams simply capture the order of operation invocation between object instances.

The specification presented here is an *extension* of the Core Object Model specification developed by Houston and Josephs [8], which has been written in Z [13]. This has been chosen because it captures a precise description of the Object Management Group's emerging standard for objects. Assuming the UML is accepted for standardisation by the group, it is likely that compliancy will be required between the core model and the UML meta-model. Thus, by showing compliancy with this important industry standard it is hoped to make the UML model convincing.

Houston's and Joseph's model is essentially a description of the syntax of a simple object model. It imposes a number of requirements on compliant systems. Such a system must be able to maintain a class library in which:

- there are a number of *types* available to the system;
- *operations* are associated with each type;
- the types form a *hierarchy*;
- *inheritance* of operations from supertypes takes places

As mentioned above, it is the intention of our work to describe a *core* UML model as opposed to every aspect of the UML meta-model. Based on the informal semantics identified in the previous section, the following additional requirements are identified. These require that:

- each type is associated with a set of possible *values*. In the case of object types these values will represent the object *instances* of the type.
- types are related by *associations*.
- operations are associated with *behaviours*, and also permit message passing.

Each of the above concepts will be considered in turn. The result will be a *Z schema* for the Core UML Model that includes seven schemas that correspond to the above concepts.



The overall approach to describing the *semantics* is to give a denotation to each of the concepts in *Z*. This will facilitate the formalization of important concepts such as behaviour, which is modelled as a set of state changes (transitions).

Taken together these concepts broadly capture the intent of the following parts of the UML Semantics document: section 4 (Common Types); sections 5 and 6 (Structural Elements - Types, Classes, Instances and Relationships); section 11 (Behavioural Elements: Interactions).

3.1 Types

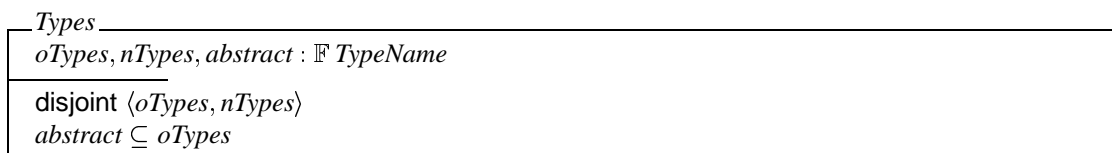
First, the object types that are available are specified as presented in [8].

It is assumed that there is a given set:

$[TypeName]$

from which the names of all types can be drawn.

The types available in the model consist of object types, non-object types (e.g. numbers) and abstract types. To express this three finite sets of type *TypeName* are declared in the schema *Types*:



The two constraints in the predicate-part of the schema state that:

- an object type cannot have the same name as non-object type.
- any abstract types must be object types.

3.2 Instances

Having introduced types, the meaning of an instance can be described.

It is assumed that there is a given set:

$[Instance]$

from which the identities of all instances can be drawn.

An instance is an instantiation of a type, and has a unique identity:

$ \begin{array}{l} \textit{Instances} \\ \hline oTypes : \mathbb{F} \textit{TypeName} \\ oInstances : \mathbb{F} \textit{Instance} \\ instances : \textit{TypeName} \rightarrow \mathbb{F} \textit{Instance} \\ \hline \text{dom } instances = oTypes \\ \bigcup (\text{ran } instances) = oInstances \\ \forall i, j : \text{dom } instances \mid i \neq j \bullet \\ \quad instances(i) \cap instances(j) = \emptyset \end{array} $
--

The constraint of the schema states that each instance is associated with exactly one type.

3.3 Values

A type defines the values of its constituent instances. The value of an instance consists of the value of its attributes at some point in time.

To express this the following given sets are assumed:

$[Attribute, \mathcal{V}]$

from which the set of all attributes and values of interest can be drawn.

A *Value* associates attributes to values:

$Value == Attribute \rightarrow \mathcal{V}$

The value of each instance must belong to those defined for its owning type:

$ \begin{array}{l} \textit{TypeValues} \\ \hline \textit{Instances} \\ tValues : \textit{TypeName} \rightarrow \mathbb{F} \textit{Value} \\ iValue : \textit{Instance} \rightarrow \textit{Value} \\ \hline \text{dom } tValues = oTypes \\ \text{dom } iValue = oInstances \\ \forall t : oTypes \bullet \\ \quad \{i : instances(t) \bullet iValue(i)\} \subseteq tValues(t) \end{array} $

Of these values, some will represent the set of initial values that a particular object type may be permitted to have when created:

$ \begin{array}{l} \textit{InitialValues} \\ \textit{TypeValues} \\ t\textit{Initial} : \textit{TypeName} \rightarrow \mathbb{P} \textit{Value} \\ \hline \text{dom } t\textit{Initial} = o\textit{Types} \\ \forall o : o\textit{Types} \bullet \\ \quad t\textit{Initial}(o) \subseteq t\textit{Values}(o) \end{array} $
--

The *Values* of the model are described by the conjunction of the above schemas:

$$\textit{Values} \hat{=} \textit{TypeValues} \wedge \textit{InitialValues}$$

3.4 Operations

The given-set:

[*OpName*]

is assumed, from which the names of all operations can be drawn.

The signature of an operation consists of the operation's *name*, the types of its *parameters* and the types of its *returnedvalues*. This is the same as in [8].

$ \begin{array}{l} \textit{Signature} \\ \textit{name} : \textit{OpName} \\ \textit{parameters, returned} : \textit{seq} \textit{TypeName} \end{array} $
--

It is now possible to define a schema *Operations1*. This expresses the fact that every object type provides an *interface*, consisting of a set of signatures:

$ \begin{array}{l} \textit{Operations1} \\ o\textit{Types} : \mathbb{P} \textit{TypeName} \\ \textit{interface} : \textit{TypeName} \rightarrow \mathbb{P} \textit{Signature} \\ \hline \text{dom } \textit{interface} = o\textit{Types} \end{array} $
--

Unlike the Core Object Model, the UML *does* permit overloading of operations, and therefore there are no further constraints required on *interface*.

The behaviour of an object type is now described as an extension to the Core Object Model. Object behaviour is modelled as a set of transitions that denote the progression of an object from one value to another. Transitions consist of a 6-tuple consisting of input and output parameters, an operation name, a transition from a before value to an after value and a message denoting the operation's external interaction.

It is firstly assumed there is the given-set:

[*Message*]

from which the set of all messages can be drawn. No further detail regarding the structure and meaning of messages are given here.

$ \begin{array}{l} \textit{Transition} \\ \textit{in, out} : \textit{seq} \textit{Value} \\ \textit{opname} : \textit{OpName} \\ \textit{before, after} : \textit{Value} \\ \textit{message} : \textit{Message} \end{array} $

The following schema describes the relationship between an object type and its behaviours:

<i>Operations2</i>
<p><i>Values</i></p> <p><i>interface</i> : <i>TypeName</i> $\rightarrow \mathbb{F}$ <i>Signature</i></p> <p><i>behaviour</i> : <i>TypeName</i> $\rightarrow \mathbb{P}$ <i>Transition</i></p>
<p>$\text{dom } \textit{behaviour} = \textit{oTypes}$</p> <p>$\forall o : \textit{oTypes} \bullet$</p> <p style="padding-left: 20px;">$\forall t : \textit{behaviour}(o) \bullet$</p> <p style="padding-left: 40px;">$\{t.\textit{before}, t.\textit{after}\} \subseteq \textit{tValues}(o)$</p> <p>$\forall o : \textit{oTypes} \bullet$</p> <p style="padding-left: 20px;">$\forall s : \textit{interface}(o) \bullet$</p> <p style="padding-left: 40px;">$\text{ran}(s.\textit{parameters} \cap s.\textit{returned}) \subseteq \textit{oTypes}$</p>

The two constraints state that:

- each transition of an object type must belong to its set of possible values.
- the signature of an operation must contain valid types.

The complete *Operations* schema is specified thus:

$$\textit{Operations} \hat{=} \textit{Operations1} \wedge \textit{Operations2}$$

4 Relationships

The UML describes two types of relationships: *associations*, which model relationships between instances of types and *generalizations*, which model supertype/supertype hierarchies. Although the UML supports n-ary associations only binary associations will be considered here.

4.1 Binary Associations

It is assumed that there are the given sets:

$$[\textit{AssociationName}, \textit{RoleName}]$$

from which the names of associations and roles can be drawn. The following enumerated is introduced to represent boolean values:

$$\textit{Bool} ::= t \mid f$$

A role connects the end of an association to a type:

<i>Role</i>
<p><i>name</i> : <i>RoleName</i></p> <p><i>owner</i> : <i>TypeName</i></p> <p><i>multiplicity</i> : $\mathbb{P}\mathbb{N}$</p> <p><i>isAggregate, isChangeable</i> : <i>Bool</i></p>
<p>$\textit{isChangeable} = t \Rightarrow \textit{isAggregate} = t$</p>

Each role has a name, an owning type and multiplicity which denotes the range of allowable cardinalities that an association role may have. A multiplicity is a subset of the natural numbers. The boolean values *isAggregate* and

isChangeable indicate whether the role represents the whole (owning) part of a aggregation or composite aggregation. The constraint of the schema states that a composite aggregation is also an aggregation, but not necessarily the reverse.

The schema *Associations* expresses the relationship between associations and roles.

<i>Associations1</i>
Values $associations : \mathbb{F} AssociationName$ $aRoles : AssociationName \rightarrow seq Role$
$dom aRoles = associations$ $\forall a : associations \bullet$ $\#(aRoles(a)) = 2 \wedge$ $\{n : \mathbb{N} \bullet (aRoles(a)(n)).owner\} \subseteq oTypes \wedge$ $\#\{n : \mathbb{N} \mid (aRoles(a)(n)).isAggregate = t\} \leq 1$

The constraints state that:

- An association has exactly two roles (as required by a binary association)
- Roles must be owned by object types.
- At most one role can belong to an aggregation.

The meaning of an association is a relation between object instances:

<i>AssociationsMeaning1</i>
$Associations1$ $aValues : AssociationName \rightarrow Instance \leftrightarrow Instance$
$dom aValues = associations$ $\forall a : associations; r_1, r_2 : Role \mid$ $r_1 = (aRoles(a)(1)) \wedge$ $r_2 = (aRoles(a)(2)) \bullet$ $0 \in r_1.multiplicity \Rightarrow ran(aValues(a)) \subseteq instances(r_2.owner) \wedge$ $0 \notin r_1.multiplicity \Rightarrow ran(aValues(a)) = instances(r_2.owner) \wedge$ $0 \in r_2.multiplicity \Rightarrow dom(aValues(a)) \subseteq instances(r_1.owner) \wedge$ $0 \notin r_2.multiplicity \Rightarrow dom(aValues(a)) = instances(r_1.owner)$

The second constraint of the schema describes the relationship between the instances it associates and the instances of the types it links. If the multiplicity of particular role contains a '0' then it is optional whether the instances of the opposite role may participate in the association. If the multiplicity is conditional (contains no '0') then the instances of the opposite role must participate in the association.

The second schema describes the semantic implication of associations in terms of the instances they relate.

$ \begin{array}{l} \text{AssociationsMeaning2} \\ \text{AssociationsMeaning1} \\ \forall a : \text{associations} \bullet \\ \quad a\text{Values}(a) \in \\ \quad \{f : \text{Instance} \leftrightarrow \text{Instance} \mid \\ \quad \quad (\forall x : \text{dom}f \bullet \\ \quad \quad \quad \#\{y : \text{ran}f \mid x \underline{f} y\} \in (a\text{Roles}(a)(2)).\text{multiplicity}) \wedge \\ \quad \quad (\forall y : \text{ran}f \bullet \\ \quad \quad \quad \#\{x : \text{dom}f \mid x \underline{f} y\} \in (a\text{Roles}(a)(1)).\text{multiplicity})\} \\ \\ \forall a : \text{associations}; n : \mathbb{N} \bullet \\ \quad (a\text{Roles}(a)(n)).\text{isChangeable} = t \Rightarrow \\ \quad (a\text{Roles}(a)(n)).\text{multiplicity} = \{1\} \end{array} $
--

The first constraint of the schema is just a way of saying that the multiplicity of an association constrains the number of object instances which may participate in it. The second constraint captures the requirement made in the UML that the multiplicity of a composite aggregation must equal one.

The schema *Associations* is the conjunction of the above schemas:

$$\begin{array}{l}
 \text{Associations} \hat{=} \\
 \quad \text{Associations1} \wedge \\
 \quad \text{AssociationsMeaning1} \wedge \\
 \quad \text{AssociationsMeaning2}
 \end{array}$$

4.2 Hierarchy

There is a distinguished type called *Object*:

$$| \quad \text{Object} : \text{TypeName}$$

The schema *Hierarchy* expresses the supertype relationship between object types, and is that given in [8]:

$ \begin{array}{l} \text{Hierarchy} \\ o\text{Types} : \mathbb{F} \text{TypeName} \\ o\text{SuperTypeOf} : \text{TypeName} \leftrightarrow \text{TypeName} \\ \\ o\text{SuperTypeOf} \in (o\text{Types} \leftrightarrow o\text{Types}) \\ o\text{SuperTypeOf} * (\{ \text{Object} \}) = o\text{Types} \\ \text{disjoint} \langle o\text{SuperTypeOf}^+, o\text{SuperTypeOf}^{+\sim} \rangle \end{array} $

The constraint of the schema are just a way of saying that the collection of object types forms a directed acyclic graph with root *Object*.

4.3 Inheritance

Finally, a type inherits all the operations supported by its supertypes:

<p><i>Inheritance</i></p> <p>$oTypes : \mathbb{F} \text{ TypeName}$ $oSuperTypeOf : \text{ TypeName} \leftrightarrow \text{ TypeName}$ $interface : \text{ TypeName} \rightarrow \mathbb{F} \text{ Signature}$</p> <hr/> <p>$\forall S, T : oTypes \bullet$ $S \mapsto T \in oSuperTypeOf \wedge$ $(interface(S)) \subseteq (interface(T))$</p>

It is not clear from the UML whether a type also inherits all the behaviour supported by its supertypes. If so, this could be written as follows:

<p><i>Inheritance1</i></p> <p><i>Inheritance</i> <i>Values</i> $behaviours : \text{ TypeName} \rightarrow \mathbb{F} \text{ Transition}$</p> <hr/> <p>$\forall S, T : oTypes \bullet$ $S \mapsto T \in oSuperTypeOf \wedge$ $(behaviours(S)) \subseteq (behaviours(T))$</p>

4.4 Compatible Extensions

A key aim of the UML is that it should permit “extensibility”. However, it is not made clear how an extension to the UML can be proven valid. In [8] the following simple law for proving that one object model is a compatible extension to another is given:

$$OM \vdash \text{ExtendedOM} \upharpoonright OM$$

Every type structure that meet the requirement of the object model OM must meet those of the extended object model. ExtendedOM is assumed to declare everything that OM does, but the constraints of OM may be relaxed and new concepts added.

Clearly, the UML model presented in this paper can be proven to be a valid extension of the Core Object Model, as only new concepts have been added and one property relaxed (overriding).

Similarly, once our Core UML Model is complete to our satisfaction, it will be possible to use it to validate further extension to the UML as they emerge. In this case, the property to be proved will be:

$$\text{CoreUMLModel} \vdash \text{ExtendedUMLModel} \upharpoonright \text{CoreUMLModel}$$

5 Formalization Strategies

The formalization strategy presented in this paper is one of two approaches that are being investigated in relation to the UML. The approach adopted in this paper is aimed at describing the UML at the *meta-level*. The advantages of this approach is that it will be particularly useful for investigating ambiguities in the UML meta-model and as a means of developing general rules and techniques for constructing, manipulating and refining UML diagrams.

The other approach we are investigating is the *translational* approach [6]. This attributes a semantics to each UML diagram by translating it to a formula in a modal logic. Since the modal logic has a well defined semantics, the translation provides a semantics for each diagram. Proof techniques have been developed for modal logics and can therefore be applied to the resulting formula. This approach will be useful for proving properties about the systems expressed by UML diagrams (as opposed to properties of the UML diagrams themselves).

Clearly, the two approach express common information; although at this stage we believe that the approaches are sufficiently different in nature to warrant investigating both.

Finally, another approach we are investigating is the use of diagrams to directly express mathematical properties of systems.

6 Conclusion

This paper has presented our motivation for formalizing the Unified Modeling Language (UML). The objective of our efforts is to make the UML a precise modeling notation, which can be used as the basis for rigorous software development. The benefits of this are:

- It will lead to a deeper understanding of OO concepts, which can in turn lead to more mature use of technologies.
- The UML models become amenable to rigorous analysis.
- Rigorous refinement techniques can be developed.

As an initial step in this direction, we have identified and formalized some core components of the Unified Modeling Language (UML). By concentrating on a small subset of the UML, we aim to develop a more understandable and manageable description of the language. A Z specification of some of the key components of a *Core UML Model* has been presented, based on an extension of the Core Object Model. Of particular benefit, is the ability to prove that future extensions of the UML are valid extensions of the current meta-model. It is intended to further enhance the specification to cover other important aspects of the UML, such as message passing. Already, a number of ambiguities and omissions have been found in the UML document during the development of the specification, which we are currently in the process of documenting.

There is considerable research interest in unifying formal and object-oriented development methods [11]. For example, [7] shows how message flow diagrams (equivalent to UML collaboration diagrams) can be given a semantics in terms of partial orders on events; [4] shows how the specification language Larch can be used to give a formal semantics to static object diagrams; and, [10] shows how LOTOS can be used to produce an executable object-oriented design.

All the above work is very important in developing a more precise understanding of emerging software development techniques and concepts. However, much of it is not particularly accessible to practitioners who have little knowledge of discrete mathematics. Instead, it is important that the theory is expressed in a language that is accessible to practitioners. In the case of the UML, the specifications presented here should be used to help clarify the emerging UML meta-model.

We intend to continue the work by developing case studies on UML designs and their formal counterparts. In particular, it is recognized that a compositional semantics will be needed in order to develop practical means of analysing UML models. Work is also currently ongoing on the development of a reference manual for the UML, which we hope will provide an powerful reference source and tool for users of the UML in industry.

References

- [1] J. Bicarregui, K. Lano, T. Maibaum. Towards a Compositional Interpretation of Object Diagrams. Technical Report, Department of Computing, Imperial College of Science, Technology and Medicine, 1997.
- [2] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language 1.0, Technical Report, Rational Software Corporation, 1997.
- [3] G. Booch. Object-Oriented Analysis and Design with Applications, 2nd ed. Benjamin/Cummings Publishing Company, Inc. 1994.
- [4] R. Bourdeau, B. Cheng. A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering 21(10) 1995.

- [5] A. Bryant, A. Evans, A formal basis for specifying object behaviour, In object-oriented behavioural specifications, chpt 2, eds H. Kilov and W. Harvey, Kluwer, 1996.
- [6] A.Clark, A.Evans. Foundations of the UML with Modal Logic. Presented at Workshop on ‘Making OO Methods More Rigorous’, Imperial College, 24th June, 1997.
- [7] W. Citrin, A. Cockburn, J von Kanel, R. Hauser. Formalized Temporal Message Flow Diagrams. Software Practice and Experience 25(12) 1995.
- [8] I. Houston and M. Josephs. The OMG’s Core Object Model and compatible extensions to it, Computer Standards and Interfaces, vol 17, nos 5-6, 1995.
- [9] I. Jacobson. Object-Oriented Software Engineering – A Use Case Driver Approach. Addison-Wesley, 1992.
- [10] A. Moreira, R. Clark. Adding Rigour ro Object-Oriented Analysis. Software Engineering Journal, September 1996.
- [11] A. Ruiz-Delgado, D. Pitt, C. Smythe. A Review of Object-Oriented Approaches in Formal Specification. The Computer Journal, 38(10), 1995.
- [12] J. Rumbaugh, Object-Oriented Modeling and Design, Prentice Hall, 1991.
- [13] J.M.Spivey, The Z Reference Manual, Prentice Hall, 1992.
- [14] J. Woodcock, J. Davis. Using Z Specification, Refinement and Proof. Prentice Hall, 1996.