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 an important step forward, since it offers the opportunity of giving a precise description to the UML. Given the standing of the UML authors, not only is the description 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

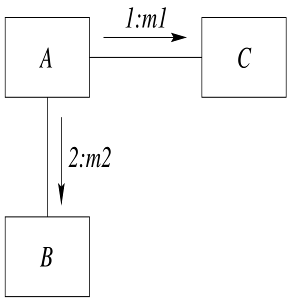
Extendability: To enable the soundness of any extensions to the UML to be verified (as encouraged by the UML authors);

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, in its present form the semantic model for the UML is insufficient to achieve these goals. Many parts of the meta- model are described using lengthy paragraphs of often ambiguous informal English, or are missing entirely. For example, copious use of the word 'Uninterpreted' is made to describe the more complex aspects of the model[2]. Moreover, by attempting to make the UML as generic as possible, the authors have created a large, complex model (over 100 pages text and diagrams), which may prove to be extremely difficult or impossible to formalise.

As an illustration of these deficiencies consider the following object collaboration diagram:



This shows three classes, A, B and C. Class A is related to both B and C which allows instances of class A to send messages to instances of the related classes. Such a diagram would be used to show the collaborations which are necessary to achieve a particular task; the example could be part of the design of an operation provided by instances of A.

The messages which are sent between classes in collaboration diagrams can be ordered. The example shows two messages m_1 and m_2 which are ordered so that message m_1 is sent before message m_2.

The information which is expressed in a collaboration diagram is very important when designing an operation. However, the exact semantics of such a diagram is open to question. This could be dangerous, since a software developer may not interpret the diagram in the way the software designer intended. For example, there are at least two possible interpretations for the dependencies between messages in the diagram:

1. for every message m_1 which is sent from an instance of A to an instance of B there will be a subsequent message m_2 from the same instance of A to an instance of C; or

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

[2] Experience of developing a semantic basis for the Core Object Model, (small in comparison to the UML), illustrates the dangers associated with the use of informal language for describing complex meta-models [5].

2. for every message m_2 which is sent from an instance of A to an instance of C there must have been a previous message m_1 sent between the same instance of A and an instance of B.

These two interpretations have very different implications. For example, the first means that a message m_1 cannot occur without a message m_2; whereas a message m_2 could happen without a message m_1. The second interpretation means the opposite. If this type of confusion is proliferated throughout a system design, it is highly likely that the software designer will not implement the system behaviour which is intended by the designer; a *precise semantics* for UML diagrams will eliminate such confusions.

Given the shortcomings of the present UML semantics, the aim of our current work is to consider some of the important issues surrounding the identification of a suitable framework for formalising the UML. In particular, the aim is to develop a framework which is much simpler and more convincing than the current meta-model. In essence it aims to:

- provide a set of sound rules for manipulating UML diagrams in a precise and formal way;
- determine when a particular UML diagram is a valid refinement of another diagram;
- permit non-trivial proofs of UML diagrams. These proofs should be automat-able using a model checking tool or theorem prover;
- be compliant with other emerging industry standard models, such as the Core Object Model.
- be transparent to the user - providing a reference source for consultation when needed

This paper presents an initial attempt to provide a suitable formal model for the UML based on the above requirements. The next section begins by exploring and introducing the UML in an informal way to determine its underlying semntic 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.

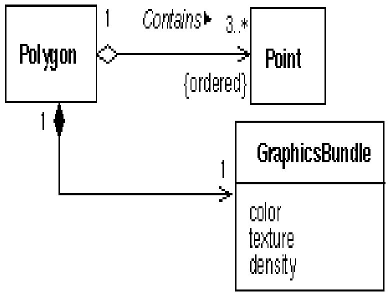
2.1 Structural Models

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*[3] 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.

[3]The 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 interchangebly.

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.

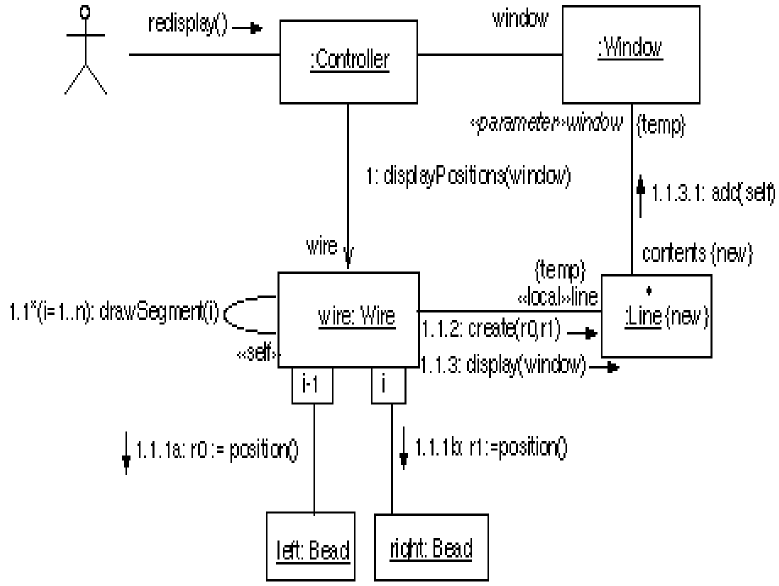


Essentially, a class diagram serves to constrain the possible collection of legal system configurations. Any program which implements the design must be in a legal system configuration at all times.

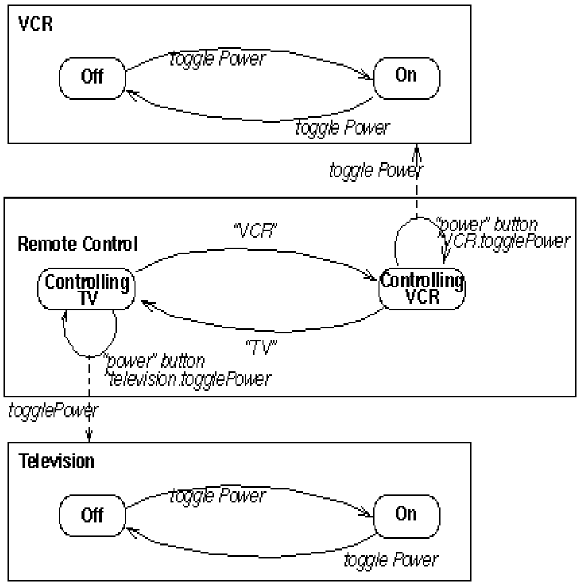
2.2 Behavioural Models

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.



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 Refinements

The UML design notation addresses the development process by allowing different diagram components to be related using *refinement* relations. Relating two diagrams by refinement indicates that one diagram is to be viewed as being derived from the other. The derivation may involve increasing the level of detail or may arise from instantiation.

2.4 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 though 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 is 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.

Note, that there are a number of different semantic interpretations that can be found in the UML meta-model regarding messages. For example, messages may be asynchronous (send and reply are separate), or synchronous (both send and reply occur at the same time) or timed (there is a time delay on acceptance of a reply)

The semantic model of UML must reflect the state based nature of object-oriented designs. Provision must be made for expressing snapshots of object states and also expressing object states changing in response to messages. This is achieved by defining a semantic model as *a graph of states*. Each state in the graph defines a snapshot during the execution of the system. The edges in the graph represent state changes.

3 Formalisation

This section present 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 the *object diagram* is the key representation used in the UML. All other diagrams represent views on this 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]. This has been chosen because it captures a precise description of the Object Manamgement 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*, which also permits 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.

$$
\begin{array}{|l}
\hline
\textit{CoreUMLModel} \\
\textit{Types} \\
\textit{Values} \\
\textit{Operations} \\
\textit{Behaviours} \\
\textit{Associations} \\
\textit{Hierarchy} \\
\textit{Inheritance} \\
\hline
\end{array}
$$

In addition, *a semantics* is given to the specification by formalising the meaning of a class as a *set of possible object values*. Underlying this semantics is the idea that each of these values represents a particular state of an object instance. This will facilitate the formalisation 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:

$$[\textit{TypeName}]$$

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*:

$$
\begin{array}{|l}
\hline
\textit{Types} \\
\textit{oTypes}, \textit{nTypes}, \textit{abstract} : \mathbb{F}\, \textit{TypeName} \\
\hline
\mathsf{disjoint}\ \langle \textit{oTypes}, \textit{nTypes} \rangle \\
\textit{abstract} \subseteq \textit{oTypes} \\
\hline
\end{array}
$$

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 Values

It is assumed that there is a given set:

$[Value]$

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

$$
\begin{array}{l}
\textit{TypeValues} \\
\hline
oTypes, nTypes : \mathbb{F}\ TypeName \\
oValues, nValues : TypeName \rightarrowtail\!\!\!\!\!\!+\ \mathbb{P}\ Value \\
\hline
\mathsf{disjoint}\ \langle \mathrm{ran}\ oValues, \mathrm{ran}\ nValues \rangle \\
\mathrm{dom}\ oValues = oTypes \\
\mathrm{dom}\ nValues = nTypes \\
\hline
\end{array}
$$

This expresses the fact that every object type and non-object type is associated with a set of possible values. The constrsint states that these values are disjoint.

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

$$
\begin{array}{l}
\textit{InitialValues} \\
\hline
TypeValues \\
oInitial : TypeName \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{P}\ Value \\
\hline
\mathrm{dom}\ oInitial = oValues \\
\forall\, o : oTypes \bullet \\
\qquad oInitial(o) \subseteq oValues(o) \\
\hline
\end{array}
$$

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

$Values \mathrel{\widehat=} TypeValues \land InitialValues$

3.3 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].

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

It is now possible to define a schema *Operations*1. This expresses the fact that every object type provides an *interface*, consisting of a set of signatures:

$$
\begin{array}{l}
\textit{Operations1} \\
\hline
oTypes : \mathbb{F}\ TypeName \\
interface : TypeName \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{F}\ Signature \\
\hline
\mathrm{dom}\ interface = oTypes \\
\hline
\end{array}
$$

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.

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

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

$$
\begin{array}{|l}
\hline
\textit{Operations2} \\
Values \\
interface : TypeName \rightarrowtail \mathbb{F}\, Signature \\
behaviour : TypeName \rightarrowtail \mathbb{P}\, Transition \\
\hline
\mathrm{dom}\, behaviour = oTypes \\
\forall\, o : oTypes \bullet \\
\quad \forall\, t : behaviour(o) \bullet \\
\quad\quad \{t.before, t.after\} \subseteq oValues(o) \\
\forall\, o : oTypes \bullet \\
\quad \forall\, s : interface(o) \bullet \\
\quad\quad \mathrm{ran}(s.parameters \frown s.returned) \subseteq oTypes \cup nTypes \\
\hline
\end{array}
$$

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 object or non-object types.

The complete *Operations* schema is specified thus:

$$Operations \mathrel{\widehat{=}} Operations1 \land Operations2$$

4 Relationships

The UML describes two types of relationships: *associations*, which model relationships between instances of types and *generalisations*, 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:

$$[AssociationName, RoleName]$$

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

$$Bool ::= t \mid f$$

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

$$
\begin{array}{|l}
\hline
\textit{Role} \\
\textit{name} : \textit{RoleName} \\
\textit{owner} : \textit{TypeName} \\
\textit{multiplicity} : \mathbb{P}\,\mathbb{N} \\
\textit{isAggregate}, \textit{isChangeable} : \textit{Bool} \\
\hline
\textit{isChangeable} = t \Rightarrow \textit{isAggregate} = t \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\hline
\textit{Associations}1 \\
\textit{Values} \\
\textit{associations} : \mathbb{F}\,\textit{AssociationName} \\
\textit{aRoles} : \textit{AssociationName} \mathrel{+\!\!\!\!\rightarrow} \operatorname{seq} \textit{Role} \\
\hline
\operatorname{dom} \textit{aRoles} = \textit{associations} \\
\forall\, a : \textit{associations} \bullet \\
\qquad \#(\textit{aRoles}(a)) = 2 \land \\
\qquad \{n : \mathbb{N} \bullet (\textit{aRoles}(a)(n)).\textit{owner}\} \subseteq \textit{oTypes} \land \\
\qquad \#\{n : \mathbb{N} \mid (\textit{aRoles}(a)(n)).\textit{isAggregate} = t\} \leq 1 \\
\hline
\end{array}
$$

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:

$$
\begin{array}{|l}
\hline
\textit{AssociationsMeaning}1 \\
\textit{Associations}1 \\
\textit{associations} : \mathbb{F}\,\textit{AssociationName} \\
\textit{aValues} : \textit{AssociationName} \mathrel{+\!\!\!\!\rightarrow} \textit{Value} \leftrightarrow \textit{Value} \\
\hline
\operatorname{dom} \textit{aValues} = \textit{associations} \\
\forall\, a : \textit{associations};\ t_1, t_2 : \textit{TypeName} \mid \\
\qquad t_1 = (\textit{aRoles}(a)(1)).\textit{owner} \land \\
\qquad t_2 = (\textit{aRoles}(a)(2)).\textit{owner} \bullet \\
\qquad\qquad \textit{aValues}(a) \in (\textit{oValues}(t_1) \leftrightarrow \textit{oValues}(t_2)) \\
\hline
\end{array}
$$

The second constraint of the schema states that the values of the association must belong to the values of the object type it links.

The second schema describes the semantic implication of associations in terms of their values.

$$
\begin{array}{|l}
\hline
\textit{AssociationsMeaning2} \\
\hline
\textit{AssociationsMeaning1} \\
\hline
\forall a : \textit{associations} \bullet \\
\quad \textit{aValues}(a) \in \\
\quad\quad \{f : \textit{Value} \leftrightarrow \textit{Value} \mid \\
\quad\quad\quad (\forall x : \textit{Value} \bullet \\
\quad\quad\quad \#\{y : \textit{Value} \mid (x, y) \in f\} \in (\textit{aRoles}(a)(2)).\textit{multiplicity}) \land \\
\quad\quad\quad (\forall y : \textit{Value} \bullet \\
\quad\quad\quad \#\{x : \textit{Value} \mid (x, y) \in f\} \in (\textit{aRoles}(a)(1)).\textit{multiplicity})\} \\
\forall a : \textit{associations};\ n : \mathbb{N} \bullet \\
\quad (\textit{aRoles}(a)(n)).\textit{isChangeable} = t \Rightarrow \\
\quad\quad (\textit{aRoles}(a)(n)).\textit{multiplicity} = \{1\} \\
\hline
\end{array}
$$

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:

$$
\begin{array}{l}
\textit{Associations} \mathrel{\widehat{=}} \\
\quad \textit{Associations1} \land \\
\quad \textit{AssociationsMeaning1} \land \\
\quad \textit{AssociationsMeaning2}
\end{array}
$$

4.2 Hierarchy

There is a distinguished type called *Object*:

$$
\begin{array}{|l}
\textit{Object} : \textit{TypeName}
\end{array}
$$

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

$$
\begin{array}{|l}
\hline
\textit{Hierarchy} \\
\hline
\textit{oTypes} : \mathbb{F}\ \textit{TypeName} \\
\textit{oSuperTypeOf} : \textit{TypeName} \leftrightarrow \textit{TypeName} \\
\hline
\textit{oSuperTypeOf} \in (\textit{oTypes} \leftrightarrow \textit{oTypes}) \\
\textit{oSuperTypeOf}^{*} \limg \{\textit{Object}\} \rimg = \textit{oTypes} \\
\mathsf{disjoint}\ \langle \textit{oSuperTypeOf}^{+}, \textit{oSuperTypeOf}^{+\sim} \rangle \\
\hline
\end{array}
$$

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:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Inheritance}} \\
\hline
oTypes : \mathbb{F}\, TypeName \\
oSuperTypeOf : TypeName \leftrightarrow TypeName \\
interface : TypeName \rightarrowtail\!\!\!\!\!\!+\;\; \mathbb{F}\, Signature \\
\hline
\forall S, T : oTypes \bullet \\
\quad S \mapsto T \in oSuperTypeOf \wedge \\
\quad (interface(S)) \subseteq (interface(T)) \\
\hline
\end{array}$$

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:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Inheritance1}} \\
\hline
Inheritance \\
Values \\
behaviours : TypeName \rightarrowtail\!\!\!\!\!\!+\;\; \mathbb{P}\, Transition \\
\hline
\forall S, T : oTypes \bullet \\
\quad S \mapsto T \in oSuperTypeOf \wedge \\
\quad (behaviours(S)) \subseteq (behaviours(T)) \\
\hline
\end{array}$$

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:

$$OM \vdash ExtendedOM \upharpoonright OM$$

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 valie extension of the Core Object Model, as only new concepts have been added and one property relaxed (overiding).

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:

$$CoreUMLModel \vdash ExtendedUMLModel \upharpoonright CoreUMLModel$$

5 Formalisation Strategies

The formalisation 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 particulary 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

The aim of this work is to produce a precise model of UML diagrams in order to facilitate the practical and well founded use of graphical methods within the software development process. We have described the benefits of having such a model and given evidence of the need for a precise UML semantics. A Z specification of some of the key components of a *Core UML Model* has been presented as a first step in this direction. This is 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 We intend to further enhance the specification to cover other important aspects of the UML such as message passing and state. 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.

In all of these cases, the specification language which is used either does not address the complete design notation or is executable and therefore in danger of leading to over-specification. We claim that the approach presented here can be used to give a formal semantics to a core part of the UML, which will be sufficient to give a clear understanding of its components.

We intend to continue this work by developing case studies as UML designs and their formal counterparts. This will lead to a general definition for the semantics of the UML and will also lead to general definitions of properties such as consistency and refinement which can be used as a basis to investigate automated methods for UML based development.

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.