Semantics of the Unified Modeling Language

Tony Clark, Andy Evans
{a.n.clark,a.s.evans}@comp.brad.ac.uk,
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 a 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 a 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 [3], Rumbaugh [12] and Jacobson's [9] popular object-oriented modelling methods. It is rapidly emerging as a de facto standard for the modelling of such systems. The UML provides the usual set of diagrammatical modelling techniques found in many object methods: object diagrams, state diagrams, object interaction diagrams, etc. 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 novel and 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), this is consists of five core UML concepts: Common concepts (types, etc); Structural Elements (types and relationships); Behavioural Elements (state machines and interactions); View Elements and Standard Elements.

Recognition of the importance of formality by the UML authors is clearly an important step forward, as it offers the opportunity of giving a precise description to the UML. It 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, etc. For example:

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.

Extensibility: 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.

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

Unfortunately, in its present form the semantic model for the UML is insufficient to achieve these goals. Certainly, the meta-model effectively defines a syntax for the language. However, semantic (meaning) properties of the UML 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 complex domain specific aspects of the model. Moreover, by attempting to make the UML as generic as possible, the authors have created a large, overly complex model (over 100 pages text and diagrams), which may prove to be extremely difficult or impossible to formalise. Indeed, experience of the development of a semantic basis for the Core Object Model [8], (small in comparison to the UML), shows the great difficulties associated with the use of informal descriptions for this purpose [5].

As an illustration of these deficiencies consider the collaboration diagram shown below:

```
          1:m1
---------  ->   ---------
|   A   | ------ |   C   |
---------       ---------
    |
    | | 2:m2
    | V
---------
|   B   |
---------
```

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 three 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 C there will be a subsequent message m_2 from the same instance of A to an instance of B; or

2. for every message m_2 which is sent from an instance of A to an instance

of B there must have been a previous message m_1 sent between the same instance of A and an instance of C.

3. both 1 and 2.

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 work is to consider some of the important issues surrounding the identification of a suitable framework for formalising the UML. In particular, we are interested in developing 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 amenable to automation using a model checking tool or theorem prover;
- should 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.

In the remainder of the paper, we discuss some of the issues surrounding the development of a suitable and practical framework for the UML, modal logics are proposed as such a framework, we give a case study which is expressed as a UML design and given a semantics in terms of a modal logic, we analyse the case study in terms of its properties and then we conclude by giving an overview of related work.

2 A Semantics for UML

The Unified Modeling 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 *static* and *dynamic*.

A static UML model expresses information about the structure of objects and classes within the system. The structure is described using relationships which can be of a specific type, for example object attributes, and which can be constrained, for example using cardinalities.

Static UML models also define the signature of classes in terms of attribute and method types. The signature of a class serves to constrain the type of objects which can be related to its instances.

Static UML models also define a specialisation relationships between classes. 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.

Essentially, a static UML model 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.

Dynamic UML models express information about the dynamic behaviour of the program which is being designed. Typically, these models define the 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 objects 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.

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.

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.

At the core of a UML design is the description of a collection of objects 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 and the value of one of its attributes. Note that several objects may be traversed to get from the object to the attribute value.

- the relationship between an object and an object which is present during an activation of one of its methods. This relationship may arise due to a parameter value, the return value of a message during the activation or a freshly created object.
- the relationship between a class and its instances and a class and its super-classes.
- the relationship which occurs when a message is sent. The participants in the message are: the source and target objects; the message name; and any data which is included in the message. Correspondingly, there is a relationship which occurs when an object replies to a message, the participants are: the source and target objects of the reply; the message which is being replied to; and any data which is sent in reply to the original message.

A UML *state* is defined to be a collection of relations. The components of the relations are: objects; classes; and messages. Objects are simply unique identifiers where basic data items such as integers represent their own identifiers. Classes and messages are represented by their names.

The state contains as many relations as are required to describe a snapshot of the required software process. However, following the analysis above, there are a number of relations which must be present and some constraints on legal states. The relations are as follows:

- *class* a set of classes.
- *object* a set of objects.
- *message* a set of messages.
- *instance* a two place relation between objects and classes.
- *send* a four place relation. An element of the relation (o_1, o_2, m, v) contains an object o_1 which is the source of the message, an object o_2 which is the target of the relation, m which is a message, and, a data item v which is the message data.
- *reply* a four place relation. An element of the relation (o_1, o_2, m, v) contains an object o_1 which is the source of the reply, an object o_2 which is the target of the reply, a message m which was originally sent from o_2 to o_1 and which is producing the reply, and, a data item v which is the reply data.

For example, suppose that in the current state there is a class *Point*, with a single instance o whose attributes are x and y, with values of 10 and 20 respectively. This state is represented as:

$$
\begin{array}{lll}
\{ & \{Point\}, & class \\
 & \{o\}, & object \\
 & \{(o, Point)\}, & instance \\
 & \{(o, 10)\}, & x \\
 & \{(o, 20)\}, & y \\
\} & &
\end{array}
$$

notice that the attributes x and y are represented as relations in the state.

Messages are described as elements of the *message* relation in the current state. For example, if O is a class, whose instance o_1 sends a *move* message to an instance o_2 of *Point*:

$$
\begin{array}{lll}
\{ & \{O, Point\}, & class \\
 & \{o_1, o_2\}, & object \\
 & \{move\}, & message \\
 & \{(o_1, O), (o_2, Point)\}, & instance \\
 & \{(o_1, o_2, move, (20, 10))\}, & send \\
\ldots & & \\
\} & &
\end{array}
$$

note that a state represents a snapshot in time; any messages contained in a state are assumed to occur at that instant.

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

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, as the reply from a message, or by directly creating the target.

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 entries in the relations corresponding to attributes of the class.

Object-oriented designs are state based. Each state contains information about objects and their current attribute values. As execution proceeds, the values of the attributes may change as a result of messages being passed between objects. A state change can occur as a result of the following:

- an attribute is updated in an object. In this case the relation defining the attribute 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.

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 Strategies

We have chosen to explore two approaches to the UML semantics: meta-level semantics [7] and translational. Both of these approaches are amenable to mathematical proof methods. We view UML as a notation to be used in the design stages of software development. As such, proof methods are highly desirable in order to verify a UML design.

The meta-level approach addresses the UML design space by giving a formal description of the UML language in Z [13] including how it can be represented and manipulated. This approach will be useful for investigating design options and for clearly indicating how to construct, transform and extend UML diagrams. This approach is not discussed further in this paper, see [7] for further details.

The translational approach 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).

A graph of states corresponds very closely to the semantics of a modal logic. Such a logic is comprised of a language and a model based semantics. The language expresses facts which are satisfied by a collection of models. Our claim is that UML diagrams are sugared modal logic formulas and that the model based semantics of modal logic provides a suitable model for the UML.

The modal logic which is used as the semantics of UML diagrams is standard. The relation is reflexive and transitive. It is beyond the scope of this paper to fully define the logic, however the following brief description gives an overview. The language which is used is an extension of first order predicate calculus. The extensions which are used in this paper are as follows:

- $\Box F$ meaning that F is always true in the future.
- $\Diamond F$ meaning that F is true sometime in the future.
- $F\,\mathbf{B}\,G$ meaning that F is true before or at the same time as G.
- $\otimes F$ meaning that F is true sometime in the past.
- $\mathbf{N}F$ meaning that F is true subsequently.
- $\mathbf{P}F$ meaning that F is true previously.
- $\mathbf{A}F$ meaning that F is true at all times in the past and future.

For example, to express that there is a *Point* class whose instances all have attributes x and y we have the following UML diagram:

```
-------------
|   Point   |
-------------
|   x:int   |
|   y:int   |
-------------
```

which is sugar for the following modal logic formula:

$$\begin{array}{l} \Box\, class(Point) \\ \Box\, class(int) \\ \Box \forall o : Point \bullet \exists n_1, n_2 : int \bullet x(o, n_1) \wedge y(o, n_2) \end{array}$$

and consequently the following is an example interpretation of the UML diagram:

$$\begin{array}{ll} \{\{Point, int\}, & class \\ \ \{o_1, o_2, \ldots, 1, 2, \ldots\}, & object \\ \ \{(o_1, Point), \ldots, (1, int), \ldots\}, & instance \\ \ \{(o_1, 1), (o_2, 4), \ldots\}, & x \\ \ \{(o_1, 3), (o_2, 9), \ldots\} & y \\ \} & \end{array}$$

4 An Example Application

In order to describe the semantics of UML diagrams we will use an example application derived from [14]. The application provides concrete examples of the various constructs. The chosen application is very simple, but provides scope for most of the main UML models. In particular, the application will require concurrent execution.

The telecommunications protocol, Signaling System No. 7, is an international standard for signaling between telephone exchanges. The protocol controls a network of signaling points between the exchanges. The protocol arranges that messages received from the first exchange will be passed on to the second exchange via a fixed route through the network. It is important that the ordering of messages is preserved by the protocol.

A message which is received at the input of a signaling point will eventually be produced at the output. At any given time, the signaling point will have received a number of messages and will have transmitted a number of messages. A signaling point cannot transmit a message which has not been received but may have received messages which it has not yet transmitted. Messages which have been received but not transmitted are currently being processed by the signaling point.

5 Design and Analysis of Application

The first stage of object-oriented analysis is to identify the major classes which are present in the requirements, the major relationships between these classes and all externally observable messages.

At a high level of abstraction, it is possible to view the required system as a single object which receives and transmits packets. Both a telephone exchange and a protocol will be viewed as special cases of objects which process packets. Therefore, we require classes describing packets and packet processors; this is described on the following UML diagram:

```
-----------------------   --------------
|    ProcPack         |   |   Packet   |
-----------------------   --------------
|  receive(x:Packet)  |
-----------------------
```

The class `Packet` simply states that there is a collection of objects which are distinct from instances of all other classes. The class *ProcPack* defines that there are a collection of objects which support a single message `receive` which receives packets for subsequent processing. The semantics of this diagram are as follows:

$$\Box class(ProcPack)$$
$$\Box class(Packet)$$
$$\Box \forall o_1, o_2 : ProcPack, o_3 \bullet send(o_1, o_2, receive, o_3) \Rightarrow instance(o_3, Packet)$$

Packets are received and transmitted by packet processors in response to messages. In order to send messages, objects must be related. This is expressed on a UML diagram by linking the appropriate classes and naming the relations. Packet processors are pipelined so that both the source and target of a message is a packet processor:

```
------------  next0,1------------
| ProcPack |---------| ProcPack |
------------         ------------
```

The relation `next` links two packet processors so that they can communicate using message passing. The semantics of this relationship is defined by a corresponding modal formula. For each packet processor there is either 0 or 1 packet processors which are related using `next`. This is expressed as follows:

$$\Box \forall o_1, o_2 \bullet next(o_1, o_2) \Rightarrow instance(o_1, ProcPack) \wedge instance(o_2, ProcPack) \wedge o_1 \neq o_2$$
$$\Box \forall o_1, o_2, o_3 : ProcPack \bullet next(o_1, o_2) \wedge next(o_1, o_3) \Rightarrow o_2 = o_3$$

The relation `next` is used to define the message passing behaviour between packet processors. Suppose that we have $next(o_1, o_2)$ and $next(o_2, o_3)$ where o_1 and o_3 are telephone exchanges and o_2 is a protocol system. The first telephone exchange o_1 sends a message `receive` to the protocol o_2 with a packet p. The protocol object will start to process the packet and on completion will send the packet to the telephone exchange o_3. This is expressed on the following diagram:

```
          1:receive(p)
------------ -----> ------------        ------------
| ProcPack |--------| ProcPack |--------| ProcPack |
------------        ------------ -----> ------------
                                2:receive(p)
```

However, this diagram simply expresses that messages which are received by a protocol object at its input will eventually be produced at its output. It is not possible in the UML to express the relative orderings which occur when different packets are processed. Using modal logic we can be more specific:

$$\begin{array}{l} \Box \forall o_1, o_2, o_3 : ProcPack \bullet \\ \quad \forall p : Packet \bullet \\ \quad\quad (next(o_1, o_2) \wedge next(o_2, o_3) \wedge send(o_1, o_2, receive, p)) \Rightarrow \\ \quad\quad\quad \Diamond send(o_2, o_3, receive, p) \end{array}$$

which states that every packet received by a protocol object will eventually be produced. Furthermore, we wish to state that the relative order of packets which are produced by a protocol object is the same as the order in which they were received. This is difficult to express in the UML, however in modal logic:

$$\begin{aligned}
&\Box \forall o_1, o_2, o_3 \bullet \\
&\quad \forall p_1, p_2 : \mathit{Packet} \bullet \\
&\qquad (\mathit{next}(o_1, o_2) \wedge \mathit{next}(o_2, o_3) \wedge \\
&\qquad\quad \mathit{send}(o_1, o_2, \mathit{receive}, p) \,\mathbf{B}\, \mathit{send}(o_1, o_2, \mathit{receive}, p_2)) \Rightarrow \\
&\qquad\qquad \Diamond(\mathit{send}(o_2, o_3, \mathit{receive}, p_1) \,\mathbf{B}\, \mathit{send}(o_2, o_3, \mathit{receive}, p_2))
\end{aligned}$$

The final step in the high-level abstract system description is to identify two different refinements of `ProcPack`. The first is a telephone exchange which is of interest simply as a source and target of packets. The second is a telecommunications protocol. These are expressed on the following diagram:

```
                   ------------
                   | ProcPack |
                   ------------
                        ^
                        |
           --------------------
           |                  |
---------------------   ------------
| TelephoneExchange |   | Protocol |
---------------------   ------------
```

which states that every instance of `TelephoneExchange` and `Protocol` may also be viewed as an instance of `ProcPack`. The semantics of this diagram are given as:

$$\begin{aligned}
&\Box \mathit{class}(\mathit{TelephoneExchange}) \\
&\Box \mathit{class}(\mathit{Protocol}) \\
&\Box \forall o : \mathit{TelephoneExchange} \bullet \mathit{instance}(o, \mathit{ProcPack}) \\
&\Box \forall o : \mathit{Protocol} \bullet \mathit{instance}(o, \mathit{ProcPack})
\end{aligned}$$

The next stage of design is to refine the definitions in order to add detail. We add an attribute to the class `Protocol` which is used to store the messages while they are being processed. The attribute is called `queue` and is of type `Pqueue`:

```
--------------------
|     Protocol     |
--------------------
|   queue:Pqueue   |
--------------------
| receive(p:Packet)|
--------------------
```

the semantics of the extension is given as a modal formula which defines a class `Pqueue` and an attribute `queue`:

$$\Box \forall o_1 : \mathit{Protocol} \bullet \exists! o_2 : \mathit{Pqueue} \bullet \mathit{queue}(o_1, o_2)$$
$$\Box \mathit{class}(\mathit{Pqueue})$$

A `Pqueue` object is a queue which contains packets. New packets are added at the head of the queue and removed from the tail. The storage for the queue is defined to be a list[2] which may subsequently be refined into an array or some other concrete data structure. The UML definition of a `Pqueue` is as follows:

```
--------------------------
|         Pqueue         |
--------------------------
| store : list of Packet |
--------------------------
|      add(p:Packet)     |
|      rem():Packet      |
--------------------------
```

The semantics of this diagram are as follows:

$$\Box \forall o : \mathit{Pqueue} \bullet \exists! l \bullet \mathit{seq}(l, \mathit{Packet}) \wedge \mathit{store}(o, l)$$
$$\Box \forall o_1 : \mathit{Pqueue}; o_2, p \bullet \mathit{send}(o_2, o_1, \mathit{add}, p) \Rightarrow \mathit{instance}(p, \mathit{Packet})$$
$$\Box \forall o_1 : \mathit{Packet}; o_2, p \bullet \mathit{reply}(o_1, o_2, \mathit{rem}, p) \Rightarrow \mathit{instance}(p, \mathit{Packet})$$

When an instance of `Protocol` is sent a packet it will add it to the head of the queue. We do not know how long this will take since the packet may be processed by the prototcol first. This is expressed on a UML diagram as follows:

```
                           2:add(p)
 1:receive(p) ------------ -------> ----------
------------> | Protocol |----------| Pqueue |
              ------------          ----------
```

the semantics of the diagram is as follows:

$$\begin{array}{l} \Box \forall o_1; o_2 : \mathit{Protocol}; o_3 : \mathit{Pqueue}; p : \mathit{Packet} \bullet \\ \quad \mathit{send}(o_1, o_2, \mathit{receieve}, p) \wedge \mathit{queue}(o_2, o_3) \Rightarrow \\ \qquad \Diamond \mathit{send}(o_2, o_3, \mathit{add}, p) \end{array}$$

In order that a protocol object can send a packet to a telephone exchange it must first be removed from the queue. This is shown in the following diagram:

[2]Note that for the purposes of this paper we assume that lists are part of the design notation.

```
                   1:p:=rem()
     ------------  ---------> ----------
     | Protocol |-------------| Pqueue |
     ------------             ----------
          | |
          | | 2:receive(p)
          | |
          | V
---------------------
| TelephoneExchange |
---------------------
```

The messages shown in this diagram are all internal, *i.e.* there is no external event which initiates them. We therefore have a choice when determining the semantics as to which message takes precedence over the other. For example, we could define that if ever there is a `rem` message there will be a subsequent `receive` message. Alternatively, whenever there is a `receive` message there must have been a previous `rem` message. Since the `rem` messages are part of the internal workings of the protocol, we take the second option as the semantics:

$$
\begin{array}{l}
\Box \forall o_1 : \mathit{Protocol}; o_2 : \mathit{Pqueue}; o_3 : \mathit{TelephoneExchange}; p : \mathit{packet} \bullet \\
\quad \mathit{send}(o_1, o_2, \mathit{receieve}, p) \wedge \mathit{next}(o_1, o_3) \wedge \mathit{queue}(o_1, o_2) \Rightarrow \\
\qquad \otimes(\mathit{reply}(o_2, o_1, \mathit{rem}, p))
\end{array}
$$

The behaviour of queues of packets is defined using a state transition diagram. A packet queue can be in one of two states: `Empty` or `NonEmpty`. Transitions between these states occur when a queue receives messages `add` and `rem`. The behaviour is defined as follows:

```
                              add(p)/store.append(p)
                                     -------
                                     |     |
                                     |     V
---------  add(p)/store.append(p)  ------------
| Empty | -----------------------> | NonEmpty |---
---------                          ------------  | rem()
    ^                                   |  ^     | [store.length()>1]
    |                                   |  ------- /store.pop()
    -------------------------------------
                 rem()[store.length()=1]/store.pop()
```

The semantics of the diagram are as follows:

$$\begin{array}{l}\Box\forall o_1;o_2 : Pqueue; l; p : Packet\bullet\\ \quad send(o_1,o_2,add,p)\wedge store(o_2,l)\Rightarrow\\ \quad\quad \mathbf{N}store(o_2,append(l,[p]))\end{array}$$

$$\begin{array}{l}\Box\forall o_1;o_2 : Pqueue; l; p : Packet\bullet\\ \quad send(o_1,o_2,rem,())\wedge store(o_2,append(l,[p]))\Rightarrow\\ \quad\quad \mathbf{N}(reply(o_2,o_1,rem,p)\wedge store(o_2,l))\end{array}$$

6 Design Properties

Once a UML design has been constructed and formalised using modal logic formulas it is possible to establish design properties. One essential property for any design is *consistency* which means that no design components contradict each other. A logic formula is consistent when at least one model exists for the formula; this can be checked using a technique called *semantic tableau*.

Modifications to a UML design often constitute *refinements*. If a UML deign is viewed as a constraint on the set of models which meet the design then a refinement results in two designs where the set of models which meet the design after the refinement is a sub-set of those which meet the design before the refinement. Since the meanings of UML designs are given by modal logic formulas this is equivalent to defining that a refinement is a modification which results in the meaning after the modification implying the meaning before the modification. A detailed discussion of this topic is outside the scope of this paper.

Other properties of the design will depend upon the design itself. For example we may want to establish that a particular object ends up in a particular state when it is sent a message. These properties can be expressed as theorems in the modal logic and then deduction techniques can be used to prove the theorem. A suitable technique is *natural deduction*.

It is beyond the scope of this paper to give a detailed analysis of the general properties of UML designs. However the next two sections give sketches of how consistency and a design property can be established.

6.1 Consistency

Consistency is established by showing that a model exists for the modal formula which is the semantics of a UML diagram. Semantic tableau is a suitable technique for establishing this property. The following example shows how a model is shown to be inconsistent by failing to complete a suitable tableau. The starting point is the formula for the design:

1$store(q, l)$
2$send(o, q, add, p)$
3$\Box\forall o_1, o_2 : Pqueue, p : Packet, l \bullet send(o_1, o_2, add, p) \wedge store(q, l) \Rightarrow$
$\quad \mathbf{N} store(q, append(l, [p]))$
4$\Box\forall o : Pqueue, l_1, l_2 \bullet store(o, l_1) \wedge store(o, l_2) \Rightarrow l_1 = l_2$
5$\mathbf{N} store(q, l)$

which states that the list of packets l used as the storage of a queue q does not change when the queue is sent an `add` message. Informally, this is not consistent with the design requirement that all new packets p are appended to the current list when an `add` message is received. The following must be true from 3:

6 $\forall o_1, o_2 : Pqueue, p : Packet, l \bullet send(o_1, o_2, add, p) \wedge store(q, l) \Rightarrow$
$\quad \mathbf{N} store(q, append(l, [p]))$

then instantiating the universally quantifier variables from 6:

7 $send(o, q, add, p) \wedge store(q, l) \Rightarrow \mathbf{N} store(q, append(l, [p]))$

now either the following must be true from 7:

8 $\neg send(o, q, add, p)$
9 $\neg store(q, l)$

which causes 9 and 1 to be contradictory; or from 7:

10 $\mathbf{N} store(q, append(l, [p]))$

Moving to the next state we get the following:

11 $store(q, append(l, [p]))$ from 10
12 $store(q, l)$ from 5
13 $\forall o : Pqueue, l_1, l_2 \bullet store(o, l_1) \wedge store(o, l_2) \Rightarrow l_1 = l_2$ from 4

Instantiating the universally quantified variables produces the following:

14 $store(q, l) \wedge store(q, append(l, [p])) \Rightarrow l = append(l, [p])$ from 13

From 14 either the following must be true:

15 $\neg store(q, l) \vee \neg store(q, append(l, [p]))$

leading to $\neg store(q, l)$ which contradicts 12 or $\neg store(q, append(l, [p]))$ which contradicts 11; or

16 $l = append(l, [p])$

which is clearly untrue. Now since we have developed all possible alternatives, we conclude that the original model is inconsistent.

6.2 Design Proof

A proof technique which is related to the method of semantic tableau is *natural deduction* which uses a collection of rules to deduce a theorem from a collection of assumptions. This is useful since the application of the rules is often consistent with human reasoning.

Consider the following formulas which are derived from the example application:

$$\begin{array}{l}\mathbf{A}\forall o_1, o_2 : \mathit{Pqueue}, l, p : \mathit{Packet}\bullet \\ \quad \mathit{reply}(o_2, o_1, \mathit{rem}, p) \wedge \mathit{store}(o_2, l) \Rightarrow \\ \qquad \mathbf{P}(\mathit{send}(o_1, o_2, \mathit{rem}, ()) \wedge \mathit{store}(o_2, \mathit{append}(l, [p])))\end{array}$$

$$\begin{array}{l}\mathbf{A}\forall o_1 : \mathit{Protocol}, o_2 : \mathit{Pqueue}, o_3 : \mathit{TelephoneExchange}, p : \mathit{packet}\bullet \\ \quad \mathit{send}(o_1, o_3, \mathit{receive}, p) \wedge \mathit{next}(o_1, o_3) \wedge \mathit{queue}(o_1, o_2) \Rightarrow \\ \qquad \mathbf{P}\mathit{reply}(o_2, o_1, \mathit{rem}, p)\end{array}$$

The first formula states that if the list of packets in a queue is l when replying to a *rem* message then it must have been $\mathit{append}(l, [p])$ for some appropriate p when receiving the *rem* message. The second formula states that when a telephone exchange is sent a *receive* message with a packet p there must have been a previous *rem* message which produced p.

Now suppose that we wish to establish the following property:

$$\begin{array}{l}\mathit{queue}(r, q) \wedge \mathit{next}(r, t) \wedge \mathbf{P}\mathit{store}(q, l) \wedge \mathit{send}(r, t, \mathit{receieve}, p) \Rightarrow \\ \quad \otimes \mathit{store}(q, \mathit{append}(l, [p]))\end{array}$$

which states that when a particular protocol r sends a *receive* message with a packet p to an exchange t then there must have been a previous state in which the queue q had a sequence of packets $\mathit{append}(l, [p])$ for some appropriate l. The proof is by natural deduction and is sketched in figure 1. The proof is expressed as a collection of boxes where each box is either a sub-proof (of a theorem from a collection of assumptions) or is a proof in a previous state (boxes marked Previously).

We are given 1 and 2 and want to prove that 19 follows. 19 is true if we can show that 18 follows from 3 – 6. 7 follows from 2 and 3 – 6. In a previous state we derive 8 from 7 and 9 from 5. Then 10 follows from 1, 11 follows from 10 and 12 follows from 8, 9 and 11. In a previous state we derive 13 from 12 and 14 from 12. Then 15 follows from 14, 16 from 15, 17 from 16, 18 from 17 and we finally conclude that 19 must be true, QED.

1 $\mathbf{A}\forall o_1, o_2 : Pqueue, l, p : Packet\bullet$
$reply(o_2, o_1, rem, p) \wedge store(o_2, l) \Rightarrow$
$\mathbf{P}(send(o_1, o_2, rem, ()) \wedge store(o_2, append(l, [p])))$

2 $\mathbf{A}\forall o_1 : Protocol, o_2 : Pqueue, o_3 : TelephoneExchange, p : packet\bullet$
$send(o_1, o_2, receive, p) \wedge next(o_1, o_3) \wedge queue(o_1, o_2) \Rightarrow$
$\mathbf{P}reply(o_2, o_1, rem, p)$

3 $queue(r, q)$
4 $next(r, t)$
5 $\mathbf{P}store(q, l)$
6 $send(r, t, receieve, p)$
7 $\mathbf{P}reply(q, r, rem, p)$

Previously
8 $reply(q, r, rem, p)$
9 $store(q, l)$
10 $\forall o_1, o_2 : Pqueue, l, p : Packet\bullet$
$reply(o_2, o_1, rem, p) \wedge store(o_2, l) \Rightarrow$
$\mathbf{P}(send(o_1, o_2, rem, ()) \wedge store(o_2, append(l, [p])))$
11 $reply(q, r, rem, p) \wedge store(q, l) \Rightarrow$
$\mathbf{P}(send(r, q, rem, ()) \wedge store(q, append(l, [p])))$
12 $\mathbf{P}(send(r, q, rem, ()) \wedge store(q, append(l, [p])))$

Previously
13 $send(r, q, rem, ())$
14 $store(q, append(l, [p]))$

15 $\mathbf{P}store(q, append(l, [p]))$
16 $\otimes store(q, append(l, [p]))$

17 $\mathbf{P} \otimes store(q, append(l, [p]))$
18 $\otimes store(q, append(l, [p]))$

19 $queue(r, q) \wedge next(r, t) \wedge \mathbf{P}store(q, l) \wedge send(r, t, receieve, p) \Rightarrow$
$\otimes store(q, append(l, [p]))$

Figure 1: The proof of a UML design property

7 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. The requirements for a semantic model for the UML have been analysed and we have shown that a modal logic is suitable. We have given an example case study and sketched how the formal semantics facilitates the analysis of various system properties such as consistency and design proofs.

There is considerable research interest in unifying formal and object-oriented development methods [11]. For example, [6] 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 modal logic can be used to give a formal semantics to the whole of the UML design notation and can support very abstract specifications. In this respect we take a similar view to [1] which uses a more sophisticated approach to address Syntropy using the Object Calculus.

We intend to continue this work by developing case studies as UML designs and their modal logic formula counterparts. This will lead to a general definition for the semantics of UML as a translation from designs to formulas. It 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] W. Citrin, A. Cockburn, J von Kanel, R. Hauser. Formalized Temporal Message Flow Diagrams. Software Practice and Experience 25(12) 1995.

[7] A. Evans, T. Clark. Foundations of the Unified Modelling Language. Second BCS-FACS Northern Formal Methods Workshop. Ilkley 1997. Springer Electronic Workshops in Computing http://ewic.springer.co.uk.

[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 to 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.