

# Concurrency and Refinement in the Unified Modeling Language

Jim Davies and Charles Crichton

*Oxford University Computing Laboratory,  
Wolfson Building, Parks Road,  
Oxford, OX1 3QD UK*

---

## Abstract

This paper shows how a formal notion of refinement may be defined for models, and model components, expressed in the Unified Modeling Language (UML). A formal, behavioural semantics is given to combinations of class, object, and state diagrams, using the notation of Communicating Sequential Processes (CSP); this semantics is adequate for the analysis of concurrent, communicating behaviour, and induces a notion of refinement for UML based upon existing notions of traces and failures refinement for CSP.

---

## 1 Introduction

The evolution of the Unified Modeling Language has tremendous significance for the field of software engineering. The definition of a single framework in which to place the sketches, diagrams, pictures, and formulae that shape and communicate our understanding of complex systems is a welcome development indeed, not least for advocates of precise, mathematical techniques.

Many explanations have been put forward for the perceived failure of *formal methods* over the last twenty years—the apparent reluctance of developers to make greater, more explicit use of mathematical techniques—but perhaps the most convincing is the lack of such a framework: prospective users of these methods have lacked guidance on where and how to apply them.

In this paper, we show how UML can serve as a framework for the application of mathematical techniques: specifically, those with a notion of process refinement. At the same time, we show how these techniques support the application of UML, by providing a formal notion of refinement for models written in the various graphical notations that make up the language.

---

<sup>1</sup> Email: [Jim.Davies@comlab.ox.ac.uk](mailto:Jim.Davies@comlab.ox.ac.uk), [Charles.Crichton@comlab.ox.ac.uk](mailto:Charles.Crichton@comlab.ox.ac.uk)

To use UML in this way, we require a formal semantics for those parts of the language that we wish to support. The authors of UML have chosen not to present a formal, mathematical semantics; however, the ‘natural language semantics’, part of the language definition, is clear enough that we may define a mathematical semantics of our own.

Our semantics will be expressed in the language of machine-readable CSP [6]. This concrete variant of Hoare’s Communicating Sequential Processes [2] is itself given a mathematical semantics in terms of sets and sequences, but—and this is closer to our purpose here—can also be used as input to animation and refinement-checking tools.

The paper begins with a brief review of the machine-readable variant of CSP. This is followed, in Section 3, by a review of those parts of UML that we have chosen to support: class diagrams, object diagrams, statechart diagrams, and interaction diagrams. As we introduce each type of diagram, we explain how it can be given a machine-readable CSP semantics.

In Section 4, we show how the existing definition of UML can be extended to support a more comprehensive approach to concurrency: one in which it is possible to describe the concurrent invocation of two or more methods upon a single object. This approach may be advantageous even where such concurrent invocation is impossible.

Finally, in Section 5, we show how the notion of refinement in CSP induces a notion of refinement for UML, based upon our machine-readable CSP semantics, and how this notion of refinement can be applied in the analysis and verification of models expressed in UML. The paper ends with a brief discussion of the issues raised.

## 2 Notation

In CSP, processes are defined in terms of the occurrence and availability of atomic, synchronous, communications: these are called CSP *events*. Each set of events, or *channel*, must be declared;

```
channel c : A . B
```

declares a set of *compound* events, each of the form  $c.a.b$ , where  $a$  is drawn from the set  $A$  and  $b$  is drawn from the set  $B$ .

The process **STOP** can perform no events: it represents the end of a pattern of behaviour. The process **SKIP** can do nothing but terminate; future behaviour is determined by the expression following the next sequential composition symbol `;`.

The process  $a \rightarrow P$  is ready to perform the event  $a$ ; if this event is performed, the future behaviour of this process is described by term  $P$ . The query symbol, `?`, denotes a choice of events: the process  $c?x \rightarrow P$  is ready to perform any event of the form  $c.x$ ; if this process performs a particular event  $c.a$ , then  $x$  takes the value  $a$  for the rest of the current scope.

The symbol  $\square$  denotes an external choice of behaviours: a menu of possible interactions.  $\sim$  denotes an internal choice, which may behave as any one of its arguments; it may represent run-time nondeterminism, or simply the absence of further information in the description. Each of these operators may appear in indexed form.

Processes are composed using a binary parallel operator, which specifies the set of events to be shared: the set of events that can occur only if performed simultaneously by both processes. In CSP, sharing does not entail hiding: shared events remain visible, and may be subsequently shared with other parallel components.

The expression  $P \parallel_A Q$  denotes the parallel combination of two processes,  $P$  and  $Q$ , sharing every event in the set  $A$ . The extended comprehension  $\{c.a, d.b\}$  denotes the set of all events whose names begin with  $c.a$  or  $d.b$ : this is particularly useful in defining parallel combinations.

The hiding operator internalises sets of events: the expression  $P \setminus A$  denotes a process that behaves exactly as  $P$ , except that events from the set  $A$  are no longer visible: they may not be shared with, and do not require the cooperation of, other processes.

The `let ... within ...` construct allows for the scoping of process names; definitions made between `let` and `within` apply only for the term immediately following `within`.

### 3 A formal semantics for UML

A *model* in UML is a collection of diagrams, illustrating different aspects of a design, together with related properties or requirements. Each of these diagrams conveys some information about the architecture, attributes, and behaviour of the system being modelled. As might be expected, the UML documentation asserts

*Every complex system is best approached through a small set of nearly independent views of a model. No single view is sufficient.*

We might expect, too, that a given model could convey more or less information about a system: models can be created at different levels of abstraction.

Before we think about comparing two different models of the same system, it is important to address the following assertion, also from the UML documentation

Every model may be expressed at different levels of *fidelity*.

Here, *fidelity* means something other than abstraction: it signals something essential about the way in which UML is used; some models are not intended to be faithful descriptions. It may be that some of the diagrams are simply sketches, and that certain lines, boxes, and annotations are imprecise, inaccurate, or even unintentional.

There is a close semantic link between *fidelity* and *formality*, both in natural language and in a more restrictive, mathematical sense: there is little to be gained from the construction of a formal semantics for those aspects of a model that are not faithful to the design; for these aspects, or perhaps for the entire model; an informal semantics will suffice.

When a model is intended as a faithful description of a system, a formal semantics can be useful indeed: we can use such a semantics to check that the various components of the model are mutually consistent, or that the system as described would exhibit particular properties; we may even use it to generate, automatically, a suite of tests that may be run upon any implementation [7].

To construct the formal semantics of a model, we require a mapping from the graphical constructs of UML to some mathematical domain: this mapping is a formal equivalent of the existing informal semantics for the language itself—an explanation of what each construct means.

There are many ways in which we might define this mapping, depending upon the purpose to which the resulting semantics will be put. As is the case with programming languages, a likely purpose for a formal semantics is to predict and reason about patterns of behaviour. Architectural and static properties are relatively easy to deduce; it is behaviour—and in particular, *concurrent behaviour*—that is harder to comprehend.

For that reason, we will concentrate upon the construction of a formal semantics that is adequate for the analysis of behaviour: one that tells us how the system as described may evolve, in terms of interactions visible at its interface.

To illustrate this construction, we choose a simple example: a two-party communication protocol whose only dynamic property of interest (at our chosen level of abstraction) is flow control. The transmitter waits for an acknowledgement that a message has been output by the receiver before accepting another for transmission.

### 3.1 Class diagrams

A class diagram identifies the entities in the system, and describes the potential relationships between them. It provides structure to a behavioural semantics—giving the signatures of objects—and places constraints upon reachable configurations, but does not directly describe the behaviour of any component.

The class diagram for our simple protocol is given in Figure 1; it identifies the two main protocol components— a transmitter and a receiver—as well as two abstract, interface entities—a user and a listener. The association between the transmitter and the receiver tells us that each is capable of referring to the other.

Each class box may have up to three partitions below the name box, listing attributes, operations, and signals, respectively. Objects may communicate by

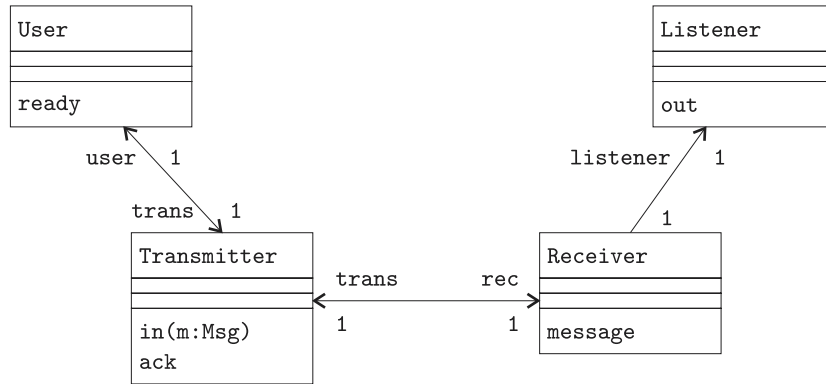


Fig. 1. A class diagram for a simple protocol

calling operations on one another, or by sending signals: operations may be synchronous; signals are always asynchronous, being passed via a notional queueing mechanism.

### 3.2 Object diagrams

Object diagrams are class diagrams in which only instances are present. Such a diagram can be used to describe a particular state of the system, or to characterise a region of the state space—each object may be annotated with a constraint upon attribute values. In a behavioural semantics, an object diagram can be used to describe an initial configuration.

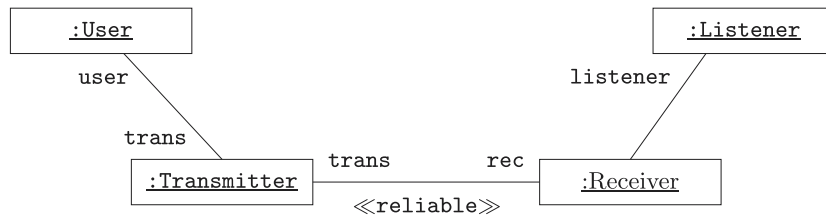


Fig. 2. An object diagram for a simple protocol

The fact that two objects can refer to each other does not mean that one should be able to call the operations of another: it may be that the two objects are connected via some communication medium. In the object diagram of Figure 2, the association instance (or *link*) between transmitter and receiver is stereotyped as `<<reliable>>`, to indicate that signals are reliably transmitted between the objects.

### 3.3 State diagrams

A state diagram shows how an object will react to the arrival of an *event*, by performing a sequence of actions, possibly accompanied by a transition from one named state to another. An event represents the receipt of a signal, or

the effect of an operation call. An action represents the sending of a signal, or the call of an operation.

A transition may be annotated with an event, a guard, and an action expression. The transition begins, or fires, with the occurrence of the trigger event. If there is a guard, it is evaluated before the action list is considered—should it prove to be false, no change in state will take place; in a sense, the transition is cancelled.

If there is no guard, or if the guard is true, then the exit actions of the source state are performed, followed by the actions of the transition itself, and then, finally, the entry actions of the target state. If two outgoing transitions are enabled at the same time, then either may fire.

The state diagram for a class describes a parameterised communicating process: a model may contain several instances of the same process, one for each object of that class; we will call each instance an *object process*. The abstract events of the process description denote the reception of (UML) events and the performance of actions.

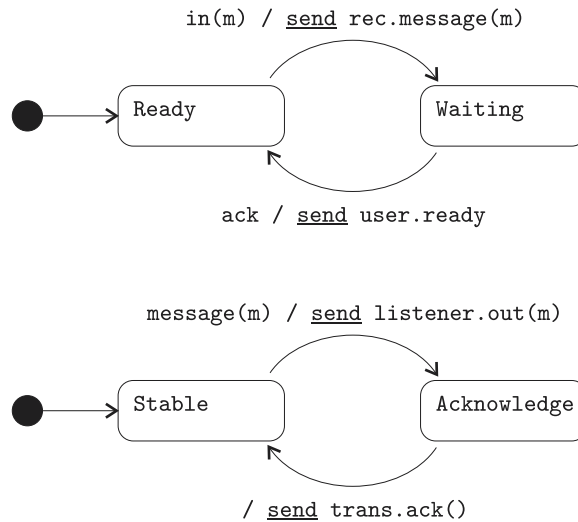


Fig. 3. State diagrams for the transmitter and receiver classes

Figure 3 shows two state diagrams: one—the upper diagram—for the transmitter class, and another for the receiver. A transmitter object can be in one of two states: it is ready for a new message, or it is waiting for an acknowledgement.

There is no blocking or rejection of (UML) events: provided that it is not engaged in an action sequence, an object process must be ready to accept any event. If an accepted event does not appear in the corresponding part of the state diagram, then it is simply discarded. If an `in` event is accepted while the object is in the `Waiting` state, then it (and the corresponding message) will be discarded; similarly, `ack` is accepted, but has no effect if the object is in the `Ready` state.

### 3.4 *Actions in state diagrams*

An essential feature of the state diagram semantics is the *run-to-completion* assumption: the actions resulting from each event must be completed before the next event can be processed. Without this, we would need additional states and transitions to represent the arrival of events during action sequences; the resulting diagrams would be unreadable.

The form of the above process reflects this: the CSP events `trans.ack` and `trans.in.m`—each representing the acceptance of a UML event, are unavailable if the process is waiting for either of the events `send.rec.receive.m` or `send.user.ready` to be performed.

For asynchronous, signal communication this is a minor concern: we may assume that each transmission medium is always ready to accept messages, and hence that any events in the object input queue will be processed; *call* communication, however, is quite a different matter.

In UML, a call action represents the sending of a message to an object: the invocation of an operation. Such an action may be described as either *synchronous* or *asynchronous* (some texts, and tools, use the terms *blocking* and *non-blocking*).

Each call action gives rise to a corresponding call event. A synchronous action is not completed until the action sequence triggered by the corresponding call event is completed. The target object may send an implicit signal, carrying a return value.

The run-to-completion assumption tells us that such a signal could not be passed using the event queueing mechanism: the calling machine is unable to process another event until the current call action, at least, has been completed.

In our behavioural semantics, a synchronous call is modelled to a pair of CSP events: one denoting the call itself, and another denoting the return. The calling process waits for the second event before proceeding.

In contrast, an asynchronous action completes as soon as the corresponding call event is accepted; no return signal is needed (or possible). An asynchronous call is modelled as a signal CSP event, shared between calling and called object processes.

### 3.5 *Interaction diagrams*

Sequence diagrams and collaboration diagrams present a series of interaction instances, represent the transmission of messages among a set of objects in a temporal order. In addition, a sequence diagram may show the lifelines of the objects involved in the interactions.

The information content of a basic interaction diagram, in terms of our behavioural semantics, is a single trace: a sequence of abstract events. In an abstract model, in which we might expect to find a degree of nondeterminism, this tells us very little about system itself.

A collection of diagrams might present further information: telling us, for example, that one of a small set of interactions *must* be possible, or that a particular interaction is guaranteed to complete. The information content of such a collection is a pair of sets of traces: a set of *good* traces that the system might exhibit, and a set of *bad* traces that it must not.

The sequence diagram of Figure 4 illustrates a desirable behaviour of the system: one in which a single message *m* is passed from the **User** to the **Listener** via the two objects of our protocol. (For our simple, deterministic protocol, this is a good characterisation).

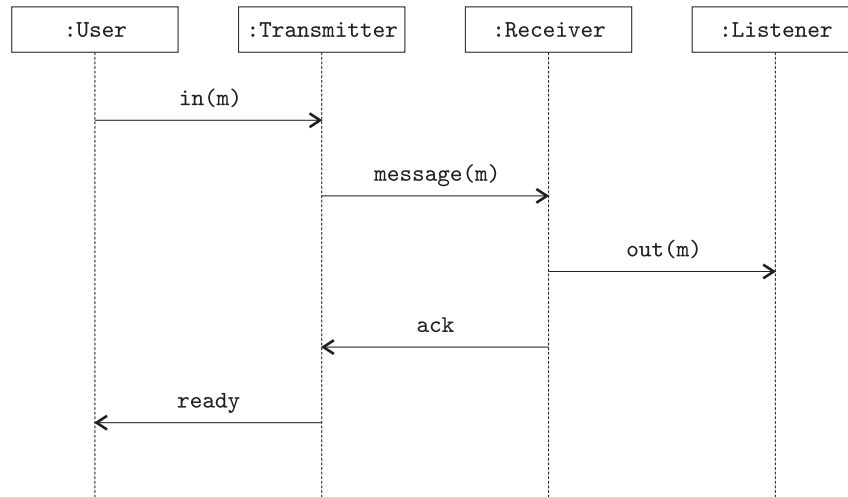


Fig. 4. A sequence diagram

Without the additional knowledge that this is the only cycle of interaction possible for our system, the information content of this diagram is simply that of the trace  $\langle \text{in.m}, \text{message.m}, \text{out.m}, \text{ack}, \text{ready} \rangle$ .

### 3.6 Activity diagrams

The activity diagram notation serves two quite different purposes in UML, describing the control flow of operations, and capturing activity and data flow across nodes. Our interpretation of an activity diagram will vary according to the purpose that it serves.

If an activity diagram is used to specify an operation, we map it to a communicating process: one that starts with an event representing an invocation, and ends with an event representing a return. Between these two points, it may perform any number of send or call actions, and process other events; this semantics has much in common with that of state diagrams.

If an activity diagram is used to describe data flow, or a pattern of distributed activity, then we may map it to a communicating process or, if the activities are linear patterns of actions or events, a set of traces. In the latter case, the information content is similar to that of an interaction diagram.

## 3.7 Models

As well as giving a semantics to individual diagrams, we can give a semantics to a complete model of a system, expressed as a collection of class, object, and state diagrams. We can construct a communicating process to represent the behaviour of each class, based upon the information content of object and state diagrams.

Each class process acts as a process factory, creating instances of object processes on demand. For example, a process factory for the transmitter class would take the following form:

```
TransmitterClass =
  call?ref!transmitter.new ->
    return.ref.transmitter.new?next ->
      ( TransmitterClass
        |||
        ( Transmitter(next)
          |[ {| receive.*.trans |} ]|
          Queue(next) ) )
```

The `new` operation is (potentially) available to any other object; a reference to the calling object, `ref`, links the call–return pair; the reference of the new transmitter object (`next`) is returned to the caller.

The `new` operation is unique, in that the return value is determined not by the object (or class) process, but by the underlying system. In our semantics, this contribution would be described by a single process whose only role is to constrain the value of `next` in events of the form `return.a.b.new.next`.

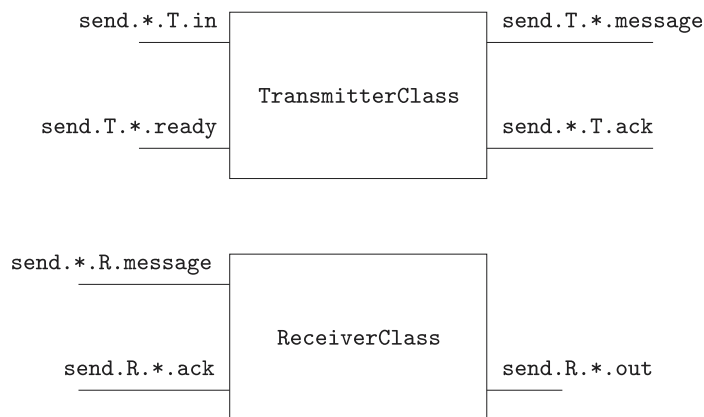


Fig. 5. Class connectivity

The remaining operations and signals from the class diagram are enough to define the connectivity of the factory processes. In our simple protocol example—if we regard the two classes `User` and `Listener` as part of the environment of the system being modelled—then the class diagram corresponds to the pair of process factories in Figure 5.

In the diagram, we use  $T$  and  $R$  to denote the sets of all references to transmitter and receiver instances, respectively. The multiplicity constraints and role names in the class diagram allow us to go further, and identify  $T$  with the set  $\{\mathbf{trans}\}$  and  $R$  with the set  $\{\mathbf{rec}\}$ .

The other diagrams identify the source and target of each signal and operation, producing a combination of object processes with the connectivity shown in Figure 6.

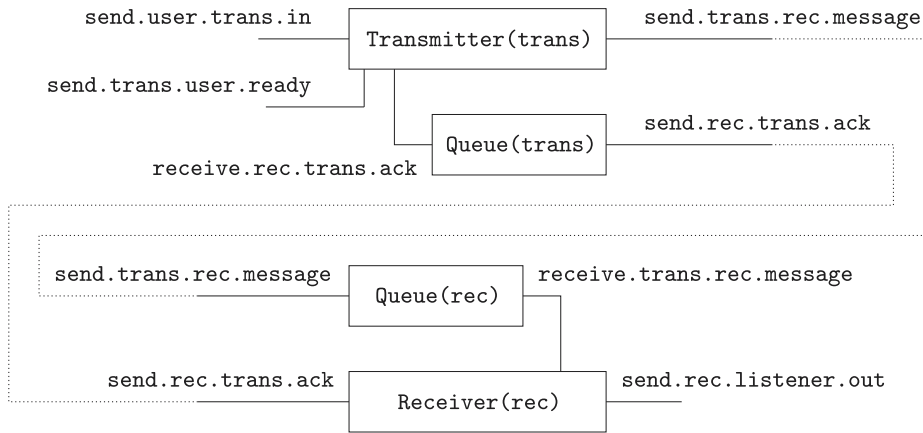


Fig. 6. Object connectivity

In our simple example, no class processes are required: the semantics is described by a pair of object processes, each with its own input queue. The behavioural semantics of our simple protocol model is shown in Figure 7.

## 4 Concurrency

In an object-oriented design, it is natural to assume that several objects may be evolving concurrently: two or more objects might receive an event simultaneously; the resulting sequences of actions might be arbitrarily interleaved. The extent to which this might be possible is an important characteristic of most designs; this importance is reflected in the design of our behavioural semantics, and in our formulation of a theory of refinement.

### 4.1 Inter-object concurrency

The extent of inter-object concurrency in a design may be constrained by model components that address threading properties:

- objects can be described as *active* or *passive*; an active object having its own thread;
- threads may be modelled explicitly, by including suitable thread classes in the model;

```

datatype Object = trans | rec | user | listener
datatype Signal = ready | message | ack | in | out
datatype Data = data | none
channel send, receive : Object . Object . Signal . Data

Transmitter(trans) =
  let
    Ready =
      ( receive.user.trans.in?m ->
        send.trans.rec.message.m -> Waiting )
      []
      receive.rec.trans.ack.none -> Ready
    Waiting =
      ( receive.rec.trans.ack.none ->
        send.trans.user.ready.none -> Ready )
      []
      receive.user.trans.in?m -> Waiting
  within
    Ready

Receiver(rec) =
  let
    Stable =
      ( receive.trans.rec.message?m ->
        send.rec.listener.out.m -> Acknowledge )
    Acknowledge =
      send.rec.trans.ack.none -> Stable
  within
    Stable

Queue(object) =
  send?other!object?signal?d ->
  receive!other!object!signal!d -> Queue(object)

System =
  ( Transmitter(trans)
    [] {| receive.rec.trans, receive.user.trans |} [] Queue(trans) )
  [] {| send.trans.rec, send.rec.trans |} []
  ( Receiver(rec) [] {| receive.trans.rec |} [] Queue(rec) )

```

Fig. 7. Semantics for the simple protocol

- we can use deployment diagrams to associate model components with nodes, and limit the number of threads at each node.

In our behavioural semantics, this corresponds to adding a process in parallel to constrain the degree of concurrency across the system.

As an illustration of this approach, consider an alternative version of our simple flow-control protocol in which our objects communicate through operation calls, and in which the previous separate transmitter and receiver functionality are provided by a single class, **Transceiver**. A class diagram for this new model is presented in Figure 8.

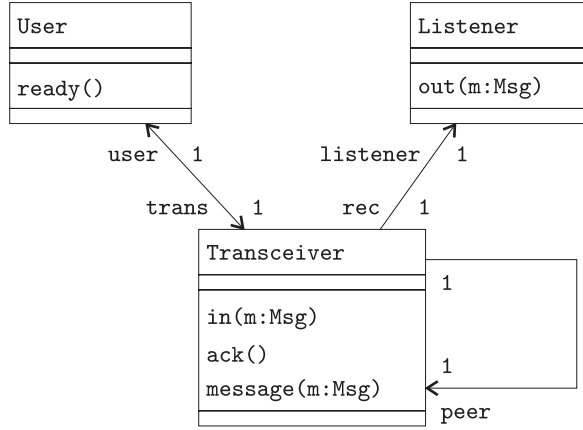


Fig. 8. Transceiver protocol class diagram

The state diagram for the **Transceiver** class—shown in figure 9—has three states, **Ready**, **Acknowledge**, and **Waiting**. In the first of these, an object is ready to accept an operation call corresponding to either the arrival of a **message** from a peer, or the input of message from the environment, for subsequent transmission to a peer.

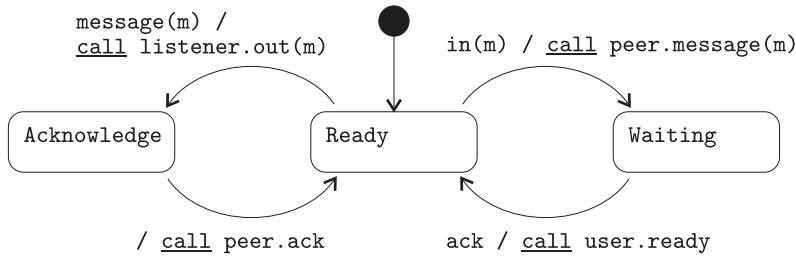


Fig. 9. Transceiver protocol state diagram

A call of **message** leads to a subsequent call of the **out** operation on a listener object, and then a call of **ack** on a peer. Until both of these calls have been completed, the transceiver object will accept no further events—this is the run-to-completion assumption.

This design has the interesting property that concurrent calls of **in**—by two users, upon two peer transceiver objects—can lead to deadlock. However, it could be that concurrency is constrained in our protocol component, to the extent that at most external operation call can be in progress at any one time.

This constraint corresponds to the assumption of a single thread of control, passing through the component as the protocol is executed; this corresponds

in turn to the addition of a process `Thread` to our semantics. A single copy of `Thread` in parallel with the objects, sharing every `call` or `return` event, will limit activity to a single object.

The resulting semantics is shown in Figure 10; a formal analysis of this semantics, using the FDR refinement checker, confirms that the possibility of deadlock has been eliminated. Unsurprisingly, if we allow two threads of control to access the protocol component—the combination of two linked transceivers—concurrently, then deadlock becomes possible: the process

```
System =
  ( Transceiver(peer1,peer2,user1,listener1)
    [| {| call.peer1.peer2, return.peer1.peer2,
          call.peer2.peer1, return.peer2.peer1 |} |]
    Transceiver(peer2,peer1,user2,listener2) )
  [| {| call, return |} |]
  ( Thread ||| Thread )
```

halts after the trace

```
< call.user1.peer1.in.data, call.user2.peer2.in.data >
```

Similar assumptions may be made about the propagation of signals. For example, we might adopt a synchrony hypothesis, similar to that of Esterel, in which we assume that any propagation and consequential actions would be complete before the environment of the system is ready to attempt another interaction.

With asynchronous communication, any problems in concurrent design would manifest themselves not in terms of deadlock, but in terms of lost or spurious messages. Nevertheless, our assumptions about threading may be represented in exactly the same way; this time, however, we constrain occurrences of `send` and `receive`, not `call` and `return`.

#### 4.2 Intra-object concurrency

The UML documentation admits the possibility of multiple, concurrent calls of operations upon a single object. Each operation has a concurrency attribute, which may take one of three values:

- *sequential*: concurrent calls on the same object are forbidden;
- *guarded*: concurrent calls are permitted, but each call will be blocked until the previous call has completed;
- *concurrent*: concurrent calls are permitted, and the corresponding sequences of transitions and actions may be interleaved.

This information can be carried forward from the model to an implementation, using features such as the `synchronized` mechanism in Java.

However, the possibility of concurrent invocation multiplies the number of states that may be of interest; any operation may be associated with a

```

datatype Object = peer1 | peer2 | user1 | user2 | listen1 | listen2
datatype Operation = message | in | out | ack | ready
datatype Data = data | none
channel call, return : Object . Object . Operation . Data

Transceiver(this,peer,user,listener) =
  let
    Ready =
      (call.user.this.in?m ->
        call.this.peer.message.m -> return.this.peer.message.none ->
        return.user.this.in.none -> Waiting )
      []
      (call.peer.this.ack.none -> return.peer.this.ack.none -> Ready)
      []
      (call.peer.this.message?m -> return.peer.this.message.none ->
        call.this.listener.out.m -> return.this.listener.out.none ->
        call.this.peer.ack.none -> return.this.peer.ack.none ->
        Ready)

    Waiting =
      (call.peer.this.ack.none ->
        call.this.user.ready.none -> return.this.user.ready.none ->
        return.peer.this.ack.none -> Ready)
      []
      call.user.this.in?m -> return.user.this.in.none -> Waiting
  within
    Ready

Thread =
  let
    Active(this) =
      (call!this?next?oa?da -> Active(next))
      []
      (return?prev!this?ob?db -> Active(prev))
  within
    call?start?object?op?data -> Active(object)

System =
  ( Transceiver(peer1,peer2,user1,listener1)
    [| {| call.peer1.peer2, return.peer1.peer2,
        call.peer2.peer1, return.peer2.peer1 |} |]
    Transceiver(peer2,peer1,user2,listener2) )
  [| {| call, return |} |] Thread

```

Fig. 10. Semantics of the transceiver model

state space of its own, describing its progress through the various actions and conditions that its performance may entail. The effect of an action may depend upon the value of temporary variables or objects—the local state of the current invocation—as well as the value of object attributes.

If we attempt to include information about the progress of multiple invocations of different operations within a single state chart, the resulting diagram will be unreadable for all but the simplest of systems.. Even the behaviour of a class with a single operation could become difficult to describe if this operation had more than two points at which it reads from, or writes to, the attributes of the object.

(Such an approach would require also a relaxation of the run-to-completion assumption for state diagrams: if a single diagram is to describe the effect of concurrent invocations, then the underlying state machine must be able to accept a second call event before the action sequence corresponding to the first has been completed.)

The only practical solution is to use a different diagram for each *compound* operation: that is, for each operation that has more than one point of interaction with the shared state. (The effect of any other, atomic operation can be represented as a single transition, triggered by a call event, on the main state diagram). The behavioural semantics of an object is then the parallel combination of the processes corresponding to each of the diagrams.

As an example of this approach, consider how we might describe the behaviour of objects of the `Printer` class shown in Figure 11.

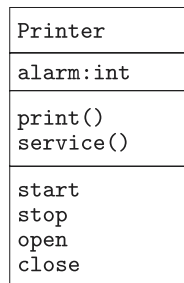


Fig. 11. Printer class diagram

The intention is that all interaction with a printer object should be through the two operations `print` and `service`. However, these do not appear in the state diagram of Figure 12.

Instead, the state diagram shows how the state space of a printer object can be partitioned into two regions: `Idle` and `Printing`. The difference between the two lies in the effect of the `open` event: if the printer is `Printing`, then this event leads to a state in which all events will be ignored, and both operations will be blocked; this last effect is achieved by setting the single attribute `alarm` to a non-zero value.

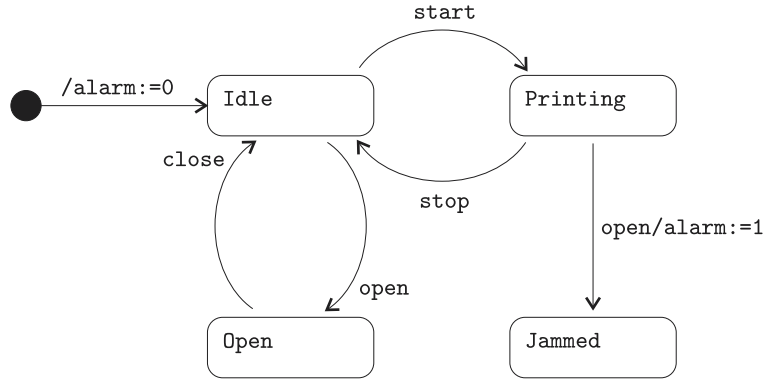


Fig. 12. Printer state diagram

Each operation is described by a separate state diagram: both are shown in Figure 13. The labelling of the initial transitions is a (suggested) notational short-cut, indicating that the invocation should be blocked if this condition is not true. The same result can be produced by adding an additional state, immediately following the initial one, with a guarded choice of completion transitions.

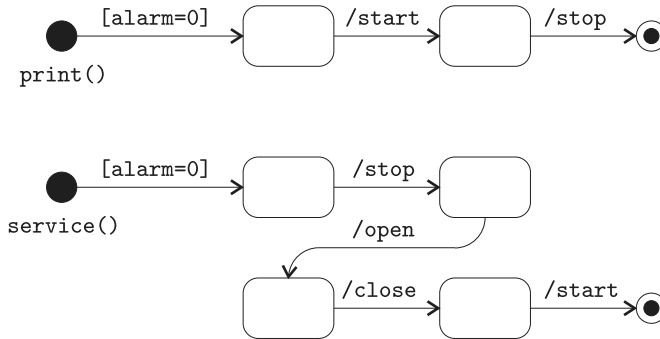


Fig. 13. Printer operation state diagrams

In this example, the two operations do not make assignments directly, but instead act upon the shared state by sending signals. Apart from fulfilling the original need for a model that can describe concurrent invocation, this approach has the additional advantage of separating two concerns: the definition of (an abstraction of) the state, and the description of the operations.

In our semantics, the processes corresponding to operations are interleaved: they do not interact directly. They share *local* signal events with a process representing the state diagram, and **set** and **get** events with a process representing the object attributes. The local signals are treated as simple synchronisations, although—if this were a particular concern—we could interpose buffer processes to capture the effect of different memory models.

If we define an ‘invocation factory’ process for each operation, then the state space of the resulting parallel combination will be infinite. To facilitate

analysis, we will usually wish to place a limit upon the number of concurrent invocations of each operation. In the semantics shown in Figure 14, we allow at most one invocation of `print`, and at most invocation of `service`.

A quick analysis of the semantics reveals that a deadlock is possible: all interaction ceases after the external trace `<call.print,call.service>`, corresponding to (amongst others) the internal trace

```
<call.print, set.alarm.zero, get.alarm.zero, call.service,
  get.alarm.zero, stop, start, open, set.alarm.one, close,
  start, stop, call.print, get.alarm.one, call.service,
  get.alarm.one>
```

The deadlock occurs because the object has entered the `Jammed` state. The state diagram allows for this, although it is clearly not a state that the object is intended to reach. Its presence in the model allows us to detect that it might occur, and also that it can safely be avoided: having identified a problem, we have only to revisit the design and place a limit upon invocations.

This can be done in one of two ways: we may make an assumption about threading, as we did for the transceiver protocol example; or we may define a UML model in which the concurrency attributes of the operations are set to *guarded* or *sequential*. The effect upon the semantics is the same: a process is introduced to limit the occurrences of `call` and `return` events.

## 5 Refinement

Our mapping of UML models to CSP processes defines not one, but a variety of semantics. The composition of this mapping with any of the semantic models defined for the language of CSP [6] defines a behavioural semantics for UML in terms of sets and sequences: in particular, we have both a *traces* and a *failures-divergences* semantics for the language of diagrams.

If  $M$  is a model expressed in UML,  $\mathcal{S}$  denotes our mapping from UML to CSP, and the semantic functions for the traces and failures-divergences models are  $\mathcal{T}$  and  $\mathcal{F}$  respectively, then  $\mathcal{T}(\mathcal{S}\llbracket M \rrbracket)$  and  $\mathcal{F}(\mathcal{S}\llbracket M \rrbracket)$  denote the trace and failures semantics of the model, respectively.

Each of the semantic models is associated with a refinement ordering, defined by reverse inclusion upon the semantic sets. These can be used to define notions of refinement for UML: for models  $M1$  and  $M2$ , we define

$$\begin{aligned} M1 \sqsubseteq_{\mathcal{T}} M2 &\Leftrightarrow \mathcal{T}(\mathcal{S}\llbracket M2 \rrbracket) \subseteq \mathcal{T}(\mathcal{S}\llbracket M1 \rrbracket) \\ M1 \sqsubseteq_{\mathcal{F}} M2 &\Leftrightarrow \mathcal{F}(\mathcal{S}\llbracket M2 \rrbracket) \subseteq \mathcal{F}(\mathcal{S}\llbracket M1 \rrbracket) \end{aligned}$$

where  $\sqsubseteq_{\mathcal{T}}$  denotes traces refinement, and  $\sqsubseteq_{\mathcal{F}}$  denotes failures-divergences refinement, for the language of UML.

```

datatype Attribute = alarm
datatype Value = zero | one
datatype Operation = print | service
channel start, stop, open, close
channel call, return : Operation
channel set, get : Attribute . Value

Attributes =
  let
    Alarm(value) = set.alarm?new -> Alarm(new)
                  [] get.alarm!value -> Alarm(value)
  within
    set.alarm?initial -> Alarm(initial)

State =
  let
    Idle =
      start -> Printing [] open -> Open
      [] stop -> Idle [] close -> Idle
    Open =
      close -> Idle
      [] open -> Open [] start -> Open [] stop -> Open
    Printing =
      stop -> Idle [] open -> set.alarm.one -> Jammed
      [] start -> Printing [] close -> Printing
    Jammed =
      start -> Jammed [] stop -> Jammed [] open -> Jammed
      [] close -> Jammed
  within
    set.alarm.zero -> Idle

Print =
  call.print -> get.alarm?value ->
    ((value == zero) & start -> stop -> return.print -> Print)

Service =
  call.service -> get.alarm?value ->
    ((value == zero) & stop -> open ->
      close -> start -> return.service -> Service)

System =
  (((Print ||| Service) [| {start, stop, open, close} |] State)
  [| {| set, get |} |]
  Attributes) \ {| set, get, start, stop, open, close |}

```

Fig. 14. Semantics of the printer model

### 5.1 Refactoring

The most obvious use of a refinement ordering is to compare two alternative designs for the same component. We regard each design as a separate ‘model’ in UML—in this case, we are using the word ‘model’ to mean a collection of diagrams describing the structure and behaviour of a single component—and simply compare the semantics of the two models.

One CSP process is a refinement of another only if they present the same interface to their environment: if one process is able to perform an event that does not even appear in the description of the other, then the refinement relationship cannot hold. To make a sensible comparison, we may need to conceal events in one or both of the process descriptions.

So that this concealment may be performed automatically, we may label operations and events in the class diagram as `{observable}`. (The extension mechanisms of UML are designed for exactly this kind of semantic hint.) Only `{observable}` operations and events will be considered in our refinement check; other events will be concealed using the hiding operator.

It is necessary also to rename the external channels in our CSP descriptions: in our semantics, each compound event records the identity of calling and called objects; these references need to be anonymised for the sake of comparison.

As an example of how we might use the refinement ordering to compare two models, consider the alternative representation of the transceiver protocol pictured in the class diagram of Figure 15. The pair of transceivers from the original protocol are replaced here by a single object; the resulting model represents the design at a higher level of abstraction.

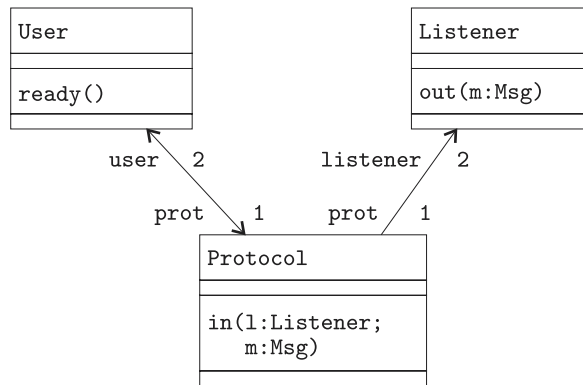


Fig. 15. Protocol class diagram

The operation state diagram for the `in` operation, Figure 16, shows how a protocol object responds to a call of its single operation, `in`, by calling the `out` operation of a listener object 1. The identity of the listener is a parameter of the `in` operation, as is the message `m`.

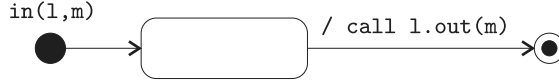


Fig. 16. Protocol operation state diagram

As there is no object state information of interest, the behavioural semantics of the new model is given simply by the operation process corresponding to this diagram:

```
InOperation =
  call?u!prot!in?l.m ->
    In(u,prot,in) ||| InOperation
```

```
In(u,prot,in) =
  call!prot!l!out!m -> return!prot!l!out!none ->
  return!u!prot!in!none -> STOP
```

With a suitable object diagram (omitted here), we can infer the connectivity information that we require to complete the semantics:  $u$  can be either `user1` or `user2`, and that  $l$  can be either `listener1` or `listener2`.

If we rename the `in` and `out` channels to anonymise the protocol and transceiver objects, but not the users or listeners, and hide the non-**observable** operations `message` and `ack`, then we might expect to find that this model is correctly refined by the transceiver model of Figure 8. However, this is the case only if we adopt the same assumption regarding threading.

The single-protocol and dual-transceiver models have the same behaviour only if each call of `in` must be completed—the return event must occur—before the next call is made. If concurrent invocations are possible, then the dual-transceiver model can deadlock, whereas the single-protocol model cannot: there is no shared state, and each invocation of `in` is processed independently.

If this is our intention, then the reader might be forgiven for wondering whether it would be possible to forgo the separation imposed in Section 4.2, and simply present a single ‘object’ state diagram for the `Protocol` class, as in Figure 17.

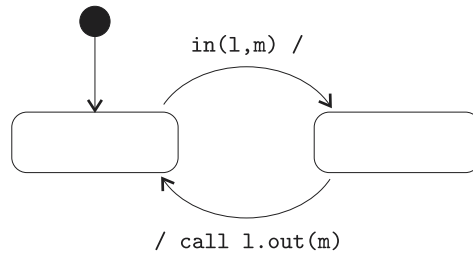


Fig. 17. Protocol object state diagram

However, this would describe a subtly different model, and *not* the one that most users of the notation would expect to find. The semantics of this diagram is

```
ProtocolState =
  call?u!prot!in?l.m -> return!u!prot!in!none ->
    call!prot!l!out!m -> return!prot!l!out!none ->
    ProtocolState
```

In this process, the `return` for `in` must occur before `out` can be called: quite the opposite of what we might expect or intend. To produce the desired result, we must either abandon synchronous operation calls, or replace the implicit return mechanism with an explicit return signal.

## 5.2 Consistency

Another use of the refinement orderings is to check that the information content of an interaction diagram is consistent with that of the model defined by the class and state diagrams. For example, we might wish to check that a sequence diagram describes a pattern of behaviour that could be observed of any implementation matching the current design.

The sequence diagram of Figure 4 may be associated with the following trace of our simple, asynchronous protocol:

```
< in.m, message.m, out.m, ack, ready >
```

The object diagram in Figure 2 allows us to identify the participant objects, yielding the following trace of compound events:

```
< send.user.trans.in.m, send.trans.rec.message.m,
  send.rec.listener.out.m, send.rec.trans.ack.none,
  send.trans.user.ready.none >
```

To test that this is indeed a trace of the protocol, we have only to check that the model semantics is trace-refined by the process

```
Sequence1 =
  send.user.trans.in?m -> send.trans.rec.message.m ->
    send.rec.listener.out.m -> send.rec.trans.ack.none ->
    send.trans.user.ready.none -> STOP
```

We have incorporated one piece of information from the diagram that may not have been explicit: the system should be capable of performing such a trace for any message value `m`.

Naturally, we will need to hide any event in our semantics that is not mentioned in the diagram. In this case, the `receive` events are not relevant:

```
System1 =
  System \ {| receive |}
```

The following refinement check succeeds, confirming that the information in the diagram is consistent with the model.

```
assert System1 [T= Sequence1
```

Here, we are checking not that this sequence of events *will* be performed, merely that the design (expressed by the model) is consistent with the possibility of such a sequence occurring. A more sophisticated interaction diagram, or a collection of such diagrams considered together, would require a different kind of refinement check.

To check that a particular sequence of actions *will* occur, we need to construct the most nondeterministic process for which this is the case, and check that this process is refined by the model semantics. This refinement check would use the failures–divergences, not the traces, ordering.

For example, a collection of suitably annotated interaction diagrams might represent a requirement that the sequence of actions

```
< send.trans.rec.message.m,
   send.rec.listener.out.m,
   send.trans.user.ready.none >
```

will *always* follow the action `send.user.trans.in.m`, in sequence, and that no subsequent `send.user.trans.in` can appear until they have done so.

To establish that any implementation is guaranteed to meet this requirement, we must check that the model semantics is a failures–divergences refinement of the following process:

```
Sequence2 =
  let
    Initial =
      send.user.trans.in?m -> Insist(m)
      |~|
      send.trans.rec.message?m -> Initial
      |~|
      send.rec.listener.out?m -> Initial
      |~|
      STOP
    Insist(m) =
      send.trans.rec.message.m ->
        send.rec.listener.out.m ->
          send.trans.user.ready.none ->
            Initial
  within
    Initial
```

Observe that this process does not insist that any combination of events should be initially available, merely that the prescribed sequence must follow any occurrence of `send.user.trans.in`.

Before we can carry out the refinement check, however, we must conceal any actions that are outside the scope of the diagram. In this case, we need to hide the set `{| receive, send.rec.trans |}`. Furthermore, we must use the information from the object diagram to set a communication context for the semantic process:

```
Context =
  [] a : {| send.user.trans.in.data,
          send.rec.listener,
          send.trans.rec,
          send.rec.trans,
          send.trans.user |} @ a -> Context
```

This process allows any number of occurrences of the prescribed `send` events; more importantly, it is unable to perform any events that do *not* correspond to links in the object diagram: for example, it is unable to perform the event `send.user.rec.message.data`. If we define

```
SystemInContext =
  System
  [| {| send |} |]
  Context
```

then we might expect our refinement check to succeed.

In practice, however, our model satisfies the requirement expressed in the sequence diagram only if the user object is sufficiently well-behaved. With asynchronous communication, the user need not wait for `ready` signal before sending another `in`; since this is clearly the intention, it is reasonable to add a process to our semantics to represent the good behaviour of the user:

```
User =
  send.user.trans.in.data ->
  send.trans.user.ready.none ->
  User
```

With this additional assumption on board, the refinement checking tool confirms that the following assertion is correct:

```
Sequence2
  [FD=
    ( SystemInContext
      [| {| send.user, send.trans.user |} |]
      User )
    \
    {| receive, send.rec.trans |}
```

where `[FD=` denotes failures–divergences refinement.

## 6 Discussion

### 6.1 *On giving a formal semantics to UML*

It is important to note that we are not attempting to give a formal semantics to the whole of the language. We agree with the authors when they say,

... a completely formal specification... would have added significant complexity without clear benefit.

However, the following statement suggests that it is worth reviewing the observations made in Section 4, and particularly those of Section 4.2:

... the UML authors targeted the modeling of concurrent, distributed systems to assure the UML adequately addresses these domains.[5]

The approach described in Section 4.2, which we claim to be adequate for the description of concurrency, cannot be applied within the constraints of the current language definition: a state diagram cannot be associated with an operation, and there is no way of referring to specific invocations.

We are left, therefore, with the approach of Section 4.1, in which concurrent invocation of operations upon a single object is prohibited. Apart from the fact that this makes a three-valued **concurrency** attribute rather inappropriate, the result is that we are unable to properly describe synchronous operation calls. We might suggest that the existing formulation of UML is *not* adequate for the proper description of concurrent behaviour.

We might expect this issue to be addressed in the near future, as the capabilities of toolsets for modelling and analysis extends beyond basic syntax checking into the areas of animation, model-checking, and the generation of test suites based upon dynamic semantics. As the authors of the UML documentation acknowledge:

The dynamic semantics are described using natural language, although in a precise way so that they can easily be understood. Currently, the dynamic semantics are not considered essential for the development of tools; however, this will probably change in the future.[5, Page 2–8]

Indeed, the beta-release version of the new UML documentation includes a trace semantics for state and interaction diagrams.

### 6.2 *Automation*

It is unreasonable to expect most users of the UML notation to translate their diagrams into machine-readable CSP notation. A high degree of automation is required and—fortunately—entirely possible. We may: create and edit a model using any UML editor; export the model in XMI format; process the XMI to extract the relevant information; and generate scripts in machine-readable CSP. Indeed, the authors are already experimenting with a prototypical tool that does exactly this.

However, the real difficulty in automatic refinement checking is constraining the state space of the model semantics. The FDR tool used for CSP requires that the processes representing models are all finite state. We must therefore extend our semantic mapping to produce descriptions in which global parameters are used to limit the number of invocation events.

For example, the process corresponding to the `Transmitter` class could be defined as

```
TransmitterClass =
  let
    Class(number) =
      (number < maxTransmitters) &
      call?ref!transmitter.new ->
      return.ref.transmitter.new?next ->
      ( Class(number + 1)
        |||
        ( Transmitter(next)
          |[ {| receive.*.trans |} ]|
          Queue(next) ) )
  within
    Class(0)
```

where `maxTransmitters` is a global constant.

Similar restrictions must be placed upon the capacity of the queues associated with each object; it may also be necessary—as in the case of our simple asynchronous protocol—to make explicit assumptions about the behaviour of the environment: that is, about the context in which any implementation of the model might be used.

### 6.3 Related work

A considerable amount of research has been carried out into the formal semantics of UML. In most cases the goal has been the improved definition of the language itself, rather than the extraction and analysis of behavioural properties. Of particular interest, with regard to checking consistency, is the work described in [8], where the implicit semantics of combinations of sequence diagrams is described in terms of FSP [3].

The work reported by Ober and Stan [4] is also important. In considering the notion of *active* and *passive* objects in UML, the authors conclude that the semantics is inadequate with regard to concurrency. They also observed problems in describing concurrent invocation.

Their position is that state machines are primarily designed to respond to asynchronous signals. They observe that active objects in UML are sequential, and suggest that passive objects *cannot have a state machine*. Their solution is to focus upon an explicit treatment of threaded behaviour, in terms of *quasi-concurrent* active objects that may yield control.

This is necessarily more concrete than the approach proposed in this paper; we are able to adopt a uniform treatment of active and passive objects, there is no reason to consider thread operations, and state diagrams can be used to describe the behaviour of objects of any class.

An additional difference between the two approaches lies in the treatment of operations. Ober and Stan focus upon threads: there is no description of operation state, or progress; call events are placed in queues, and processed, along with events corresponding to thread operations, by the state machine of an active object.

Of the research in which the semantics is intended to support practical analysis, the closest to that described here is [1], in which *capsules* are used to describe architecture, state diagrams are used to describe behaviour, and the resulting descriptions are analysed using FDR. The translation is manual, but their goals are clearly the same as our own.

## Acknowledgements

The authors would like to thank the organisers of the Refinement Workshop for the opportunity to discuss this work, at a suitably early stage, with others working in the same area. They are also extremely grateful to IBM for supporting this research, under a Faculty Partnership Programme (FPA award), and to their partners in the EU-funded AGEDIS project [7].

## References

- [1] Engels, G., J. Kuster, R. Heckel and L. Groenewegen, *A methodology for describing and analysing consistency of object-oriented behavioral models*, in: *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.
- [2] Hoare, C., “Communicating Sequential Processes,” Prentice Hall, 1985.
- [3] Magee, J. and J. Kramer, “Concurrency: state models and Java programs,” Wiley, 1999.
- [4] Ober, I. and I. Stan, *On the concurrent object model of uml*, in: *Proceedings of EuroPar99*, LNCS **1685** (1999).
- [5] Object Management Group, *UML 1.4 with Action Semantics*, electronic document, OMG (2002).  
URL <http://cgi.omg.org/cgi-bin/doc?ptc/2002-01-09>
- [6] Roscoe, A. W., “Theory and Practice of Concurrency,” Prentice Hall, 1997.
- [7] The AGEDIS project, *Automated Generation and Execution of Test Suites for Distributed Systems*, (EU-IST 1999–20218).  
URL <http://www.agedis.de>

- [8] Uchitel, S., J. Kramer and J. Magee, *Detecting implied scenarios in message sequence chart specifications*, in: *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.