

Triggering Active Objects*

O.M. Nierstrasz

Abstract

Active objects are concurrent, active entities based on the object-oriented paradigm. We present a model for understanding active objects based on the remote procedure call, and on the notion of *activities*, which capture a single-thread flow of control between objects. We also present simple mechanisms for creating activities, interleaving and delaying activities, and for constructing atomic actions and concurrent subactivities. We show how these mechanisms can be used to capture very general forms of *triggering*. Our model for active objects, and the mechanisms for manipulating activities are embedded in *Hybrid*, a concurrent, object-oriented language. The model is also useful for understanding and dealing with deadlock in such systems.

1 Introduction.

The object-oriented approach is an increasingly popular approach for enhancing reusability of code through abstraction, instantiation, inheritance, and homogeneity [Nier86 Nyga86 Stef85]: Objects are persistent entities with a well-defined interface for handling requests, and a hidden realization. Objects with the same interface are instances of *types* (or “classes”), and may inherit properties from parent types. In fully object-oriented models, types are objects themselves, and may be dynamically instantiated.

Less well-recognized is the opportunity that objects suggest for understanding concurrent and distributed systems. We present a model for object-oriented systems in which objects are the active entities, and we present a small set of high-level primitives for communication and synchronization between active objects. We show how an object-oriented programming language that adheres to this model and is equipped with its primitives, lends itself well to expressing solutions to a variety of concurrency problems, and, in particular, event-driven behaviour, or “triggering”.

Our model is based on the simple observation that an object, having a well-defined interface and a hidden realization, resembles a “server” process handling requests from other processes. We view an object environment, then, as consisting of a number of “active objects”, all potentially executing in parallel. These objects may communicate, providing services to one another through their interfaces.

*In *Objects and Things*, ed. D.C. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, March 1987, pp. 43-78.

Typically one speaks of objects exchanging “messages”, though this terminology is used more to emphasize object independence than to suggest an actual implementation strategy: one requests service of an object by “passing it a message” rather than by executing a procedure with the object as a parameter. The paradigm is meant to illustrate that objects are not just passive entities, but have some freedom in deciding how to handle requests. We take this one step further, and suggest that objects should be seen as concurrent, active entities, with the message-passing paradigm extended somewhat to enable flexible, though highly-structured communication between objects. We do not suggest, however, that message-passing be the only correct means of implementing communication between objects, although it is an obvious and natural approach to take.

Our model is based on the notion of objects as “servers”, with the remote procedure call as the basic model for all communication between objects. By this we mean that an object may execute a blocking send of a *call* message to invoke an operation of another object, or a send a *return* message in response to a call of an operation. There are no explicit receives. Objects that are not blocked waiting for a return and are not otherwise occupied are ready to receive calls to their operations.

We note that objects may be structured, being composed of other objects. This is a fundamental assumption of the object-oriented approach, since one is free to re-use instances of objects in different settings. Since some objects may then be considered “parts” of other objects, it is reasonable that an object with many parts may not wish all of those parts to be independently concurrent. We call a “unit of concurrency” a *domain*. A domain corresponds to a top-level, active object with possibly many dependent parts. One may think of a domain as a persistent process, with the property that the operations of objects in the domain are mutually exclusive, i.e., there can only be one thing executing in a domain at a time. If we say that objects are “active” when they are busy handling a request, then it is easy to see that the objects in a domain cannot all be active at once. In fact, objects may spend much of their lives being “inactive”, waiting for a request to be made of them. Of course, it is up to the programmer to decide which objects should be independently active and therefore be mapped to different domains.

The paradigm of the remote procedure call suggests that there are “threads of control” passing through objects and between domains, defined by calls and returns. We call such a thread of control an *activity*. Activities can be dynamically instantiated by invoking a special kind of “top operation” called a *reflex*. A reflex is invoked by sending a *start* message. It is different from a call, since it is intended to start a new thread of control, and is therefore non-blocking.

To this basic remote procedure call model, we add the notion of *queues* to capture activities that are temporarily suspended due to the unavailability of a domain, and the notion of *delegation*, which allows an object performing a call to switch its attention to another activity rather than be forced to wait for the return. Objects may exercise a fine degree of control over when certain operations may be invoked by declaring *delay queues* which may be explicitly *closed* and *opened*. Remote procedure calls are thus available as a default mode of controlling the interaction of concurrent activities, but delay queues and delegation are available as mechanisms for expressing more general kinds of interaction not otherwise possible.

For example, it is possible to capture the notion of *triggers* by representing trigger conditions with delay queues, and the triggers themselves by activities suspended on the delay queues. The actions of the trigger may be part of the delayed operation, or of operations to be subsequently invoked. Calls to these operations are delegated to prevent the object from blocking while the trigger activity

waits.

Many classical concurrency control problems, such as that of reading and writing a bounded buffer, or of maintaining a constraint over several objects, can be interpreted as triggering problems, and thus be easily solved using the mechanisms we propose. We give examples of solutions to several such problems using objects programmed in *Hybrid*, a language based on our model of active objects.

We also show how our model of active objects can be further enhanced to capture the notions of concurrent *subactivities* and of nested *atomic actions*. In a distributed environment with real concurrency, both concepts are arguably fundamental, and should be included in our model. As before, the remote procedure call remains the basic model for communication, and thus communication and synchronization of active objects is highly-structured and regular.

We observe how deadlock can be understood in the model, and therefore how various ways of handling deadlock can be mapped into it. We finish by drawing comparisons with other models for communication and synchronization between concurrent processes, notably monitors and guarded commands.

2 Activities and Domains.

The set of active objects in an object environment is partitioned into a set of *domains*, each of which is a single concurrent entity. Every domain has a *root* object, and other objects in the domain are *subobjects*, or “parts” of the root. An object handling a request is said to be “active”. Two objects in the same domain are *dependent*, and may not be active simultaneously. Two *independent* objects may be concurrently active according to the rules that we shall describe. An active object is always active on behalf of some *activity*. An activity may be thought of as a token representing transfers of control between cooperating objects.

Every domain has a queue for receiving requests to the objects it contains. In addition, there may be a number of *delay queues* for delaying calls to certain operations that are not always available to service requests. Every operation uses at most one queue to receive its call requests, however operations belonging to the same object may share delay queues.

Every domain has a set of one or more *queues* at which the objects it contains may receive requests to perform operations. Objects and operations may share queues, but every operation is attached to at most one queue. The set of all queues is partitioned by domains, that is, operations of independent objects cannot share a queue. In our programming language *Hybrid*, objects in the same domain share a single queue for all incoming requests, except when programmers explicitly require extra queues for certain operations.

A domain may be *idle*, *active* or *blocked*. Objects in an *idle* domain are free to accept requests to perform operations from other objects. When a request is accepted, the domain is *active* on behalf of the activity associated with the request. Objects in an active domain are free only to accept requests on behalf of that activity (and hence only from dependent objects). When an object in an active domain issues a request to an independent object, the domain becomes *blocked*, and waits for the request to return. Objects in a blocked domain may only accept requests associated with

the activity upon which the domain is blocked (i.e., recursive *calls* may be accepted in addition to the awaited *return*).

An activity may be either *active* or *suspended*. There is always a one-to-one correspondence between active activities and active domains. A suspended activity corresponds to a request (or return) which has not yet been accepted, and is always associated with a particular queue of some domain to which that request has been addressed. Note that both domains and activities are process-like, since either can be active or not active. Domains, however, correspond more closely to the accepted notion of a process, since they exhibit a clear notion of a local state which changes over time. An activity, on the other hand, is a thread of control which may pass through *several* domains. Nevertheless, we shall avoid the use of the term “process” and allow the reader to draw his own conclusions.

A *delay queue* is a queue may be either *open* or *closed*. Objects may only accept requests on queues that are open. Every operation of every object in a domain must be attached to some queue associated with that domain. Return messages and calls to operations that do not use a delay queue are associated with a single queue for the domain which is always open. Delay queues are always associated with just one object in a domain, and that queue may be opened and closed only by the operations of that object. (A delay queue is, in fact, an instance variable of the object whose operations use it.) Notice that an object whose operations are at some time all attached to closed queues can never again open any of those queues. Such an object may be a source of *deadlock* in a system.

We shall now formally define the state of an object environment in terms of domains, activities, queues and operations. In the sections that follow, we shall give examples illustrating how event-driven behaviour can be captured in a programming language like Hybrid which adheres to this model.

For simplicity we shall assume that each domain is “flat”, and contains only a single root object. The operations of that object then resemble the procedures of a module (or a monitor). The reader should keep in mind, however, that the model generalizes to structured objects.

The *activity state* of an object environment is represented by a tuple:

$$S = \langle D, A, Q, P, M \rangle$$

where

$$D = \{\dots, d_i, \dots\}$$

is a set of *domains*,

$$A = \{\dots, a_j, \dots\}$$

is a set of *activities*,

$$Q = \{\dots, q_k, \dots\}$$

is a set of *queues*,

$$P = \{\dots, p_l, \dots\}$$

is a set of *operations*, and M is a set of mappings. The sets are finite and disjoint, but may change in number from state to state.

The mappings in M are as follows:

$$\begin{aligned} \text{status} : D &\rightarrow \{\text{idle}, \text{active}, \text{blocked}\} \\ \text{status} : A &\rightarrow \{\text{active}, \text{suspended}\} \\ \text{status} : Q &\rightarrow \{\text{open}, \text{closed}\} \end{aligned} \quad (1)$$

We adopt the convention that:

$$\begin{aligned} D_I &= \{d \in D \mid \text{status}(d) = \text{idle}\} \\ D_A &= \{d \in D \mid \text{status}(d) = \text{active}\} \\ D_B &= \{d \in D \mid \text{status}(d) = \text{blocked}\} \\ A_A &= \{a \in A \mid \text{status}(a) = \text{active}\} \\ A_S &= \{a \in A \mid \text{status}(a) = \text{suspended}\} \\ Q_O &= \{q \in Q \mid \text{status}(q) = \text{open}\} \\ Q_C &= \{q \in Q \mid \text{status}(q) = \text{closed}\} \end{aligned}$$

Also, every domain d has a queue q_d which is peramanently *open*.

Next, we have:

$$\text{active} : A_A \leftrightarrow D_A \quad (2)$$

that is, there is a bijective mapping between active activities and active domains.

Every suspended activity is associated with a queue:

$$\text{susp} : A_S \rightarrow Q \quad (3)$$

Every queue is associated with a domain:

$$\text{domain} : Q \rightarrow D \quad (4)$$

and every operation uses a unique queue:

$$\text{uses} : P \rightarrow Q \quad (5)$$

We say that Q_d is the set of queues associated with domain d , and P_d the set of operations associated with d , that is:

$$\begin{aligned} Q_d &= \{q \in Q \mid \text{domain}(q) = d\} \\ P_d &= \{p \in P \mid \text{domain}(\text{uses}(p)) = d\} \end{aligned}$$

For every queue there is some (possibly empty) set of operations associated with the same domain that may open that queue:

$$\text{open} : Q \rightarrow 2^P \quad (6)$$

such that:

$$\forall q \in Q_d, \text{open}(q) \subseteq P_d$$

Obviously, this is only relevant for delay queues.

Every domain that is blocked is blocked on behalf of some activity:

$$\text{blocked} : D_B \rightarrow A \quad (7)$$

Finally, there is a mapping from any domain d and activity a to the number of *returns* that domain is waiting for on behalf of that activity.

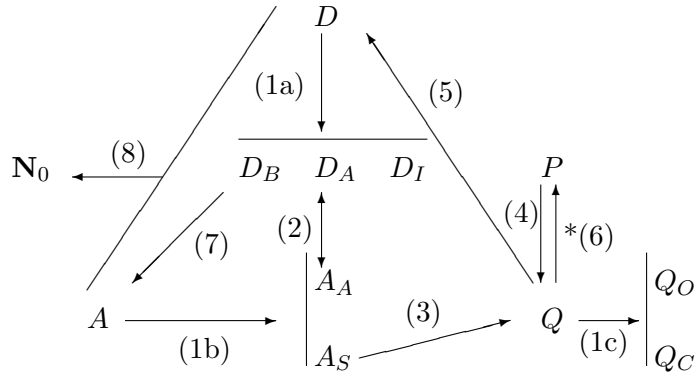
$$\text{returns} : D \times A \rightarrow \mathbf{N}_0 \quad (8)$$

(This represents the depth of the return stack for a (d, a) pair.)

The activities which are “pending” for a domain can be summarized in the derived mapping:

$$\text{pending}(d) \equiv \{a \mid \text{returns}(d, a) > 0\}$$

The mappings we have defined are summarized in the following diagram. Recall that (6) maps to sets of operations.



To aid in the visualization of an activity state, we shall represent domains by open circles, activities by labelled tokens (small circles), and queues by open-sided squares attached to domains. A active activity is represented by a token in a circle, and a suspended activity by a token in an open square. An idle domain is represented by an empty circle, and a blocked domain by a circle with a minus sign, optionally labelled with the name of the activity it is blocked on. A closed queue is indicated by an extra line on the closed side of the open square. When a domain is blocked, we draw an arrow from that domain to the queue that received the request causing the domain to block.

The example in figure 1 shows a suspended activity a_1 in a closed queue of an idle domain d_2 . The domain d_1 is blocked on activity a_1 , and is in fact waiting for the very request upon which a_1 is suspended.

An activity state may change as a result of the following events:

1. The *acceptance* of a queued request.
2. The execution of a *call*, a *return* or a *delegated* call.

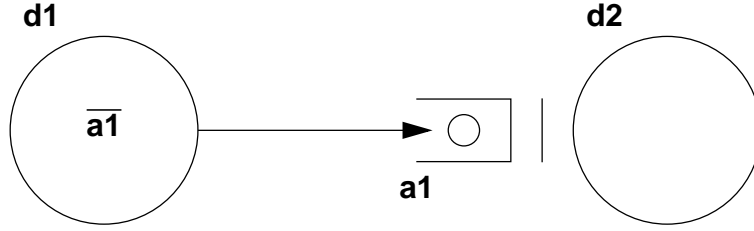


Figure 1: A suspended activity.

3. The *starting* or *ending* of an activity.
4. The *creation* or *destruction* of a domain.
5. The *opening* or *closing* of a delay queue.

The effect of these events on the state is generally obvious. Opening or closing a delay queue only affects the *status* mapping, for example. We shall take a detailed look at a couple of events, however, since some care is required to understand exactly what they entail.

Acceptance of requests may occur when there are blocked or idle domains with activities suspended on their queues. In particular, a domain d may accept a request from activity a if:

$$\begin{aligned} & a \in A_S, \text{ susp}(a) = q \in Q_O \\ \wedge & d = \text{domain}(q) \end{aligned}$$

and either:

$$\begin{aligned} & (1) \quad d \in D_I \\ \text{or} & \\ & (2) \quad d \in D_B \wedge a = \text{blocked}(d) \end{aligned}$$

Actually, an idle domain must accept the *first* such request in a queue, though we do not represent this ordering in the model we describe here. A blocked domain, however, can only accept requests from the activity it is blocked on, so some “filtering” of new requests must take place.

A request may be a call, a return, a start (new activity) or a delegated call. The mappings affected are *status*, *active*, *susp*, and possibly *blocked* and *returns*. The new state S' will have new mappings:

$$\begin{aligned} \text{status}' &= \text{status} \cup \{(d, \text{active}), (a, \text{active})\} \\ &\quad \setminus \{(d, \text{status}(d)), (a, \text{status}(a))\} \\ \text{active}' &= \text{active} \cup \{(a, d)\} \\ \text{susp}' &= \text{susp} \setminus \{(a, q)\} \end{aligned}$$

In addition, if $d \in D_B$, then:

$$\text{blocked}' = \text{blocked} \setminus \{(d, a)\}$$

and if the request is a *return*, then the request must be sent to the queue q_d , and:

$$\text{returns}' = \text{returns} \cup \{(d, a, n - 1)\} \setminus \{(d, a, n)\}$$

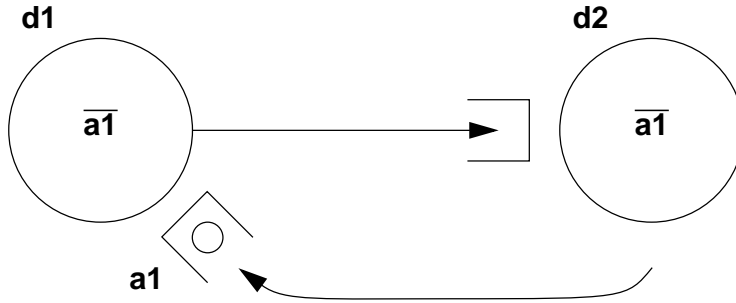


Figure 2: A recursive call.

where $n = \text{returns}(d, a)$.

Note that blocked domains may accept calls in addition to returns, since there is always a possibility of recursion, as in figure 2. Such recursive calls should never be made to operations using queues that may be closed, since a deadlock situation could arise: a domain cannot accept calls on a closed queue, and a blocked domain can only accept requests from the activity it is blocked on. Also note that a blocked domain can never accept a *start* request, since such requests are always associated with new activities (and hence never the one the domain is blocked on). A start request can only be accepted when the domain is idle again.

Executing a *call* affects the same mappings, and results in:

$$\begin{aligned}
 \text{status}' &= \text{status} \cup \{(d, \text{blocked}), (a, \text{suspended})\} \\
 &\quad \setminus \{(d, \text{active}), (a, \text{active})\} \\
 \text{active}' &= \text{active} \setminus \{(a, d)\} \\
 \text{susp}' &= \text{susp} \cup \{(a, q)\} \\
 \text{blocked}' &= \text{blocked} \cup \{(d, a)\} \\
 \text{returns}' &= \text{returns} \cup \{(d, a, n + 1)\} \setminus \{(d, a, n)\}
 \end{aligned}$$

where q is the queue used by the called operation and $n = \text{returns}(d, a)$.

When an operation performs a *return* statement, the new status of the returning domain depends on the value of $\text{returns}(d, a)$. If it is zero, the domain becomes idle, otherwise it becomes blocked. The *status*, *active*, *susp* and *blocked* mappings are modified in the obvious way, however the *returns* mapping is not affected. (It is relevant to the calling domain, not the domain being called.)

A *delegated call* is like a call, except that the new status of the domain is *idle* instead of *blocked*, and the *blocked* mapping is unaffected. The effect of this is to permit the domain to switch to another activity while the call is pending. This is normally impossible since an object calling another would ordinarily block, preventing other activities from getting in. Note that when a call is delegated, the eventual *return* message can only be accepted when the domain is idle. If the calling domain has switched to another activity, the *return* must be queued and wait.

When a *reflex* is invoked, a new activity a' is started. The *start* request is similar to a *call*, except that the caller does not block and will never receive a *return*. The new activity state is characterized

by:

$$\begin{aligned} A' &= A \cup \{a'\} \\ susp' &= susp \cup \{(a', q)\} \end{aligned}$$

where q is the queue used by the reflex.

When an activity *ends*, the new state is such that:

$$\begin{aligned} A' &= A \setminus \{a'\} \\ status' &= status \cup \{(d, idle)\} \setminus \{(d, active), (a', active)\} \\ active' &= active \setminus \{(a', d)\} \end{aligned}$$

When a new domain d' is created, its status is *idle*, and it acquires a set of operations and open queues. The mappings *domain*, *uses*, and *open* must be modified to reflect the relationship between the new domain and its queues and operations. The *returns* mapping is modified so that:

$$\forall a \in A, returns'(d', a) = 0$$

The destruction of a domain may only take place if it has no further *pending* activities (as defined above), and there are no further activities suspended on any of its queues. Destruction must coincide with a *return* or an *end*, since the activity token of that domain must be released.

We shall now see how an object-oriented programming language that adheres to this model and provides the primitives we have described can be used to concisely express event-driven behaviour in concurrent systems.

3 Triggers.

Triggers appear in various settings, perhaps most commonly in databases and operating systems. In databases, triggers are used to maintain constraints after updates have been made. In operating systems, the most obvious examples of triggers are device interrupts, exceptions, and user-generated signals. In all cases, triggers are of the general form:

$$condition \rightarrow action$$

This rather loose notion of triggers does not tell us when the condition is to be evaluated, nor does it indicate whether the action is expected to affect the value of the condition (i.e., predicate). In a rule-based system, for example, every such trigger might be a rule, and rules could be repeatedly fired until all their trigger conditions are false. In other settings, the trigger might disappear after the action is successfully performed.

If we consider the case where the trigger condition is initially false, then one is in fact waiting for an *event* that may make the condition become true. It is only when such an event takes place that one need re-evaluate the condition. In the case of database triggers, the condition is a violation of some consistency constraint, and the action is a procedure to restore the constraint. The condition is evaluated whenever any of the data objects involved in the constraint is updated. In fact, in the case of some constraints, there may be only certain kinds of updates that may cause the constraint

to be violated. For example, with the constraint:

$$x + y > z$$

only events that increment x or y , or decrement z should cause the constraint to be checked.

We suggest that the activity model we have described in the previous section is powerful enough to capture a very general notion of triggering using a small number of constructs. These constructs, when embedded in a programming language, such as *Hybrid*, enable a programmer to compactly express triggering of active objects. Furthermore, we suggest that the ability to express triggers is crucial to the success of a programming language that takes concurrency seriously.

The general idea is to capture each trigger in an activity suspended on a delay queue, where that queue is closed if the trigger condition is false. Events (operations) that cause the condition to become true may open the queue. Delegation may be used if objects wish to be triggered on more than one event, or if a trigger condition depends on different events, and thus requires several suspended activities to wait for these events. Once a queue is opened, the natural scheduling of active objects, i.e., through the implicit *accepts* of requests, will cause the suspended activities to (eventually) become active again. The action associated with a trigger condition may be encoded either in the operation attached to the delay queue, or in an operation to be invoked after the delayed operation returns.

An arbitrary trigger condition is a predicate over a collection of objects, where each object is a variable over a state space. Events which cause these objects to change state may in turn cause the trigger condition to become true. We consider the following list of triggering examples as being fairly representative of the range of triggers that can possibly be of interest:

1. Await the availability of a reusable resource.
2. Await arbitrary state from a finite state space.
3. Await the “next” consumable resource.
4. Await “next” state from a totally ordered state space.
5. Await arbitrary state from a countable state space.
6. Await multiple events.

In cases 1 through 5 we consider more and more general events that one can wait for, and in case 6 we consider waiting for multiple events at once. We shall show how each of these triggering examples can be encoded in Hybrid. First we shall present a brief summary of Hybrid to aid in the presentation of the example solutions.

3.1 Hybrid.

Hybrid is an object-oriented programming language that adheres to our model of activities and active objects. Objects are structured, and “top-level objects” are root objects of a domain. Objects

in different domains are independent, and may execute independently on behalf of different activities. Objects are strongly-typed, meaning that an object type determines the operations supported by its instances. One must therefore know the type of an object before attempting to invoke an operation. The type of an object may, however, be determined at run-time, so that objects can be manipulated fairly flexibly despite strong-typing. Types are not disjoint, in particular, a *subtype* consists of objects that support *at least* the operations of its parent types, and possibly more.

A definition of a type consists of the keyword *type*, a name for the new type, type parameters, a type specification, and an optional *private* part that provides a realization for instances and their operations. Type parameters are used to parameterize instances of the new type. For example:

```
var a : array [1..10] of integer ;
```

declares a variable *a* of type *array*, with type parameters 1..10 and *integer*. This notion is generalized to programmer-defined types. The type specification determines the interface of the newly-defined type through the use of *type constructors* and previously-defined types. For example:

```
type matrix [xtype, ytype] of stuff :
  array [xtype] of
    array [ytype] of stuff ;
```

defines a new type *matrix* in terms of the *array* constructor. The type *matrix* is essentially a new constructor which must be parameterized whenever instances are declared. In this case the operations of *matrix* objects are inherited from *array*, and from the type parameters.

One may also define abstract types by using the *abstract* constructor, and providing a list of operation stubs to define the interface of the new type. In this case, in order to create instances, one must also supply a *private* part to the definition, following the type specification, which determines a realization for each of the operations, and for the internal state of the instances. It is possible to define operations that may be invoked using identifiers, or using infix or prefix operators (such as +, −, etc.).

Expressions in Hybrid are of the general form:

```
<target> <operation> <argument>
```

The *target* is typically a variable name or a constant, but may be an expression identifying an object. The *argument* may also be an expression. If the *operation* name is an identifier, it is invoked using “dot” notation, i.e.:

```
stack.push(frame) ;
```

Infix and prefix operators are invoked in the obvious way. There are no postfix operators, to simplify the parsing rules in the presence of operator-overloading. Note that prefix operators take no arguments, the only “argument” being the target object itself, as in the expression $-x$.

Simple statements consist of an expression that will evaluate to *nil*, followed by a semi-colon. All statements terminate either with a semi-colon, or with a matching right brace. Control statements will either have obvious semantics, or will be explained together with the examples.

An new activity is created in Hybrid by invoking a *reflex*, a specially declared operation that does not return a value. It is invoked with an exclamation mark instead of a dot:

```
mailbox ! cleanup ()
```

The “caller” continues to run independently.

A new object is created by invoking the *create* operation on a *type* object. An object identifier is returned. To create a new domain, one must *export* a newly-created object, thus creating a new root object (and, consequently, a new domain):

```
winOid := export ( window.create )
```

Every domain implicitly has a shared queue for receiving requests to any of its subobjects. Delay queues must be explicitly declared as variables of type *delay*, and may be attached to operations by declaring in the *private* part of the type definition that an operation *uses* that queue. Several operations (or reflexes) may be attached to the same queue, but an operation may use no more than one queue. A delay queue may be opened or closed using the operations *open* and *close*. A delay queue is initially open.

An operation invocation may be *delegated* by bracketing the invocation expression with the *delegate* construct:

```
delegate(<target> <operation> <argument>)
```

After the *target* and *argument* expressions have been evaluated, the *operation* is invoked, and the caller does not block to wait for the return. The return is accepted only when the calling domain is idle again. The value of a delegated expression is exactly the same as that of a normal (undelegated) expression. Only the flow of control is affected. An object may delegate a call to an operation of an object in the same domain, in which case the current activity is suspended on that operation’s queue, and the domain may switch to another activity.

3.2 Examples.

Our first triggering example is of a resource that may be exclusively “acquired” by an object for a period of time. In this case there is only one state that one may wait for, namely *available*, and one event, namely *release*, which will make the resource available. The trigger condition and action are bundled up in the single operation *acquire*, as is shown in example 1.

Example 1: A resource manager.

```
type resource :
  abstract {
    acquire : → ;
    release : → ;
  } ;
```

```

private
{
  var available : delay ; # initially open
  var holder : oid of object ;
  acquire : → ;
    uses available ;
  {
    holder := caller ;
    available.close ;
    return ;
  }
  release : → ;
  {
    if ( caller =? holder ) {
      available.open ;
      holder := nil ;
    }
    return ;
  }
}

```

In this example, resources would be subtypes of the type *resource*, and inherit the operations *acquire* and *release*. *Caller* is a *pseudo-variable* in Hybrid that discloses the object identifier of the object that has invoked an operation. This example is somewhat simple-minded since useful resources have other operations in addition to *acquire* and *release*. Since there is nothing to prevent an object from invoking these operations without first performing an *acquire*, it is clear that the other operations must also filter calls and check the *caller* against the *holder* of the resource. In fact, in the case of resources, it is more appropriate to think of an *activity* as acquiring the resource. As such, we shall find in a later section that the notion of an *atomic action* is better for expressing medium-term exclusive access than that provided by operations, and that the deadlock issues can then be better understood.

In the second example, objects wish to be informed of a change to a particular state from a finite state space. In our example, one may wait for the condition “mouse button is up”, or “mouse button is down”. Notice that again we use one queue to stand for each awaited condition. Also notice that an *awaitUp* executed when the button state is *up* will return immediately. The *init* operation of the *button* object assumes that the button is initially *up*, and accordingly closes the *downQueue*. This example can be easily generalized to other kinds of “button” objects with more than two states.

Example 2: A mouse button.

```

type buttonState : enum { up , down } ;

type button :
  abstract {
    init : → ;
    state : → buttonState ;
    awaitUp : → ;
    awaitDown : → ;
    up : → ;
    down : → ;
  } ;
private
{
  var upQueue , downQueue : delay ;
  var currentState : buttonState ;
  init : → ;
  {
    downQueue.close ;
    return ;
  }
  state : → buttonState ;
  {
    return ( currentState ) ;
  }
  awaitUp : → ;
  uses upQueue ;
  {
    return ;
  }
  awaitDown : → ;
  uses downQueue ;
  {
    return ;
  }
}

```

```

up : → ;
{
  currentState := up ;
  downQueue.close ;
  upQueue.open ;
  return ;
}
down : → ;
{
  currentState := down ;
  downQueue.open ;
  upQueue.close ;
  return ;
}
}

```

The third example is of a bounded buffer. In this case the states to wait for are *notEmpty* and *notFull*. The example bears comparison to a solution of the same problem using monitors given in [Hoar74]. The most important difference is that running activities are never suspended using delay queues. Activities may become suspended upon a call or return, but not when they open or close a delay queue. The semantics of open and close are thus quite different from those of *wait* and *signal*, despite a superficial similarity. We shall discuss the differences more fully in the section on other models.

Example 3: A circular bounded buffer.

```

# bound is constrained to be an enumerated type
type buffer of ( itemType , bound :< enumType ) :
  abstract {
    write : itemType → ;
    read : → itemType ;
  } ;
private
{
  var notFull , notEmpty : delay ;
  var bufferSpace : array [ bound ] of itemType ;
  var readIndex , writeIndex : bound ;

```

```

init : → ;
{
  notEmpty.close ;
  # bound is an enumerated type
  # (i.e., a type object)
  # whose first element is bound.first.
  readIndex := bound.first ;
  writeIndex := bound.first ;
  return ;
}
write : ( item : itemType ) → ;
  uses notFull ;
{
  bufferSpace [ writeIndex ] := item ;
  switch ( writeIndex ) {
    case bound.last { writeIndex := bound.first ; }
    default { writeIndex := writeIndex.succ ; }
  }
  if ( writeIndex =? readIndex ) {
    notFull.close ;
  }
  notEmpty.open ;
  return ;
}
read : → itemType ;
  uses notEmpty ;
{
  var item : itemType ;
  item := bufferSpace [ readIndex ] ;
  switch ( readIndex ) {
    case bound.last { readIndex := bound.first ; }
    default { readIndex := readIndex.succ ; }
  }
  if ( readIndex =? writeIndex ) {
    notEmpty.close ;
  }
  notFull.open ;
  return ( item ) ;
}
}

```

The buffer example can be generalized to other kinds of “stream” objects that manage streams of

inputs and outputs. It can also be seen as a variation of example 1 above, in which the resource to be acquired is the next item in the stream.

The fourth example is somewhat more interesting, and makes use of both delegation and reflexes in addition to delay queues. In this case we wish to trigger an activity upon an “update”, that is, we are interested in the “next” state of some object. Here the problem is that the notion of “next” is not well-defined, since there may be many objects interested in updates, each with their own idea of the “previous” state. A solution is to record the state explicitly in a variable that strictly increases with each update. An object may then ask to be triggered upon the update of some explicit *previous* state. In the example, we see that *awaitUpdate* takes an argument which is the previous known state.

Example 4: Awaiting updates.

```

type updateCount : integer ;
type updateAbleObject :
  abstract {
    state : → updateCount ;
    update : → updateCount ;
    awaitUpdate : updateCount → ;
  } ;
private
{
  var updates := 0 : updateCount ;
  var uponUpdate : delay ;
  init : → ;
  {
    uponUpdate.close ;
  }
  state : → updateCount ;
  {
    return ( updates ) ;
  }
  update : → updateCount ;
  {
    updates += 1 ;
    uponUpdate.open ;
    self ! wasUpdated ;
    return ( updates ) ;
  }
}

```

```

awaitUpdate : ( previous : updateCount ) → ;
{
  if ( previous <? updates ) {
    return ;
  }
  else { return ( delegate ( self.awaitNext ) ) ; }
}
awaitNext : → ;
  uses uponUpdate ;
{
  return ;
}
# Shares queue with awaitNext()
reflex wasUpdated : ;
  uses uponUpdate ;
{
  uponUpdate.close ;
  # reflexes don't return
}
}

```

The state of an *updateAbleObject* may be sampled using the *state* operation. Every *update* causes the state variable to be increased and reports the new state. An *awaitUpdate* requires the previous known state as an argument. This means that the argument must be seen in order to decide whether to return or not. Unfortunately this means that the *awaitUpdate* call must be accepted *before* one knows if an update has taken place, and hence a delay queue on this operation would be of no help. Instead, *awaitUpdate* examines its arguments, and then *delegates* the call to the delayed operation *awaitNext*. This operation is not part of the visible interface of the object, only being available to other operations in the *private* part. The *awaitNext* operation delays all activities synchronized with the true current state of the object. (Note that an optimization is possible if we permit the notion of *forwarding* calls: in this case *awaitNext* could return directly to the original caller instead of returning first to *awaitUpdate*.)

Finally, when an update takes place, it is necessary to purge all the activities suspended on the *uponUpdate* queue. The queue is opened in the *update* operation, and a new activity is spawned to close the queue when all the suspended activities have been triggered. This is accomplished by having the *wasUpdated* reflex use the same queue as *awaitNext*. This means that *wasUpdated* request effectively “flushes” out the calls to *awaitNext*.

Note that there is no synchronization problem since only one activity may be active in the domain at a time. It is therefore impossible for a later call to *awaitNext* to get in “ahead of” the request to *wasUpdated* while *update* is, in fact, still executing.

Our fifth example is that of waiting for an arbitrary state from a countable state space. We shall

not give this example in full, since the approach is similar to that of the previous examples. The difference with the fourth example is that the object whose state is being monitored must maintain a *table* of triggers, one for each awaited state.

A concrete example would be a *clock* object that agrees to wake up objects at a requested time. The clock object would have an operation *awaitTime*, whose argument is some possible state of the clock, and a *tick* operation that causes it to change state. Whenever an *awaitTime* request is made, the clock object checks if that time has passed. If so, it returns immediately, otherwise it checks its table to see if a *triggerObject* exists for the awaited time. If not, it creates such an object and adds it to its table. Then it delegates the call to the *awaitTrigger* operation of the *triggerObject*. When a *tick* occurs, the clock checks to see if there is a *triggerObject* waiting for the new state in its table. If so, it invokes the *trigger* operation of the *triggerObject*, and the *terminate* reflex. As in example 4, the *terminate* reflex shares a delay queue with the *awaitTrigger* operation, so as to “flush” out all waiting calls.

Example 5: A trigger object.

```

type triggerObject :
  abstract {
    awaitTrigger : → ;
    trigger : → ;
    terminate : → ;
  } ;
private
{
  var delayQueue : delay ;
  init : → ;
  {
    delayQueue.close ;
  }
  awaitTrigger : → ;
  uses delayQueue ;
  {
    return ;
  }
  reflex terminate : → ;
  uses delayQueue ;
  {
    quit ;
  }
}

```

```

    trigger : → ;
    {
        delayQueue.open ;
    }
}

```

Here we only show the definition of the *triggerObject*, since the *clock* object itself is quite similar to the *updateAbleObject* of the last example. In fact, one could have implemented the *updateAbleObject* using a single instance of a *triggerObject* instead of a table of them.

Our last example is intended to show how one can construct triggers depending on multiple conditions. In example 6 we see a *line* object which is attached to two *point* objects upon creation. It is constrained so as to continue to be attached to those points, even if the points move. The constraint requires that the line redraw itself whenever the points move. The line and the two points may all be independent objects. This means that they may be created and manipulated independently, and various line objects may be attached to the same point. Furthermore, the line object must be capable of responding to requests even while it is waiting for the points to move. The “trick” is to create one activity for each awaited event that the constraint depends on, and to delegate the calls that will be delayed waiting for those events. The example assumes that each *point* is an *updateableObject* as in example 4, and therefore keeps track of its updates. The line object starts a reflex for each point that awaits an update of that point, and maintains the constraint when an update occurs.

Example 6: A constrained line object.

```

type updateablePoint :
  record {
    var poid : oid of point ;
    var pstate : updateCount ;
  } ;
type line init (oid of point,oid of point) :
  abstract {
    redraw : → ;
    # other operations ...
  } ;
private
{
  var end : array[1..2] of updateablePoint ;

```

```

init : (p1, p2: oid of point) → ;
{
  end[1].poid := p1 ;
  end[1].pstate := p1.state ;
  end[2].poid := p2 ;
  end[2].pstate := p2.state ;
  self.redraw ;
  self ! watchEnd(end[1]) ;
  self ! watchEnd(end[2]) ;
  return ;
}
reflex watchEnd : (n: 1..2) ;
{
  end[n].pstate :=
    delegate(end[n].awaitUpdate(end[n].pstate)) ;
  self.redraw() ;
  self ! watchEnd(n) ; # repeat
}
redraw : → ;
{
  # code for drawing a line
  return ;
}
}

```

This example is not complete as it does not deal with the issue of synchronizing the line with the two points. The *redraw* operation may need to take care that it is informed of a particular state of each endpoint. If the points are moving very quickly, many intermediate states may be lost. Whether this is important or not depends, of course, upon the application. In the following section we shall briefly discuss the issue of atomic actions and subactivities, and how they can help to solve some of these problems.

4 Subactivities and Atomic Actions.

The concept of *delegation* provides us with a simple mechanism for switching activities within a domain. It is not, however, very practical for controlling precisely *which* activities to switch between. Furthermore, it can be rather cumbersome to spawn multiple activities whose results must later be gathered together before another activity may continue.

Consider, for example, an object acquainted with a set of independent objects, such as our line object connected to two points, or an *organism* object whose *weight* is the sum of the weights of its (independent) *component* objects. If the components are capable of truly executing in parallel,

then we could take advantage of this by spawning a subactivity for each subweight to be gathered. Presently we would have to use reflexes to create the subactivities and use a delay queue to gather the results.

Example 7: Subactivities using reflexes.

```

type organism [ intRange ] :
  abstract {
    weight : → units ;
  } ;
private
{
  var sumQueue : delay ;
  var child : array[intRange] of oid of component ;
  var totalWeight : units ;
  var notDone : integer ;
  weight : → units ;
  {
    var n : integer ;
    n := intRange.first ;
    notDone := 1 + intRange.last - n ;
    doneQueue.close ;
    totalWeight := 0 ; # units
    loop {
      check (n :? intRange) {
        self ! subWeight(n) ;
      }
      else { break ; }
      n += 1 ;
    }
    return(delegate(self.awaitDone)) ;
  }
  awaitDone : → units ;
  uses doneQueue ;
  {
    return(totalWeight) ;
  }
}

```

```

reflex subWeight : (n:intRange) ;
{
  totalWeight += delegate(child[n].weight) ;
  notDone -= 1 ;
  if (notDone ==? 0) {
    doneQueue.open ;
  }
}
}

```

In example 7 we see how this can be done. The “parent” activity releases control of the *organism* by delegating its call to *awaitDone*. At this point control is switched to the subactivities started in the *subWeight* reflex. Each subactivity delegates its call to the *weight* operation of a component, and thus enables switching to the next subactivity. When they are all done, the delay queue is opened, and the parent activity is triggered.

The *check* statement in Hybrid is used to perform a run-time type-check of a variable and cast that variable to the indicated subtype. In this case, the variable *n* is tested for membership in the integer range subtype, and temporarily cast it to that subtype if the check succeeds. It can then be passed as a valid argument to *subWeight*.

The problem with this solution is that other activities are not prevented from gaining access to the *organism*. In particular, suppose that there are two pending calls to *weight*. When the first delegates its call to *awaitDone*, control may be switched to the second call of *weight*, rather than to the *subWeight* subactivities. A slightly more elaborate solution is required to solve this problem.

Instead, we suggest that the notion of a *subactivity* is important enough to be supported by a special notation. The solution in this case would be as in example 8.

Example 8: Explicit subactivities.

```

type organism :
  abstract {
    weight : → units ;
  } ;
private
{
  var child : array[intRange] of oid of component ;
  var totalWeight : units ;

```

```

weight : → units ;
{
  var n : integer ;
  n := intRange.first ;
  totalWeight := 0 ; # units
  coloop {
    check (n :? intRange) {
      activity {
        totalWeight +=
          delegate(child[n].weight) ;
      }
    }
    else { break ; }
    n += 1 ;
  } # wait here ...
  return(totalWeight) ;
}
}

```

Here the subactivities are created using the *activity* construct within a *coloop* scope. When a subactivity delegates a call, control is transferred to the parent, rather than to an arbitrary activity. When the coloop is terminated with a *break* statement, the parent waits for the subactivities to terminate, transferring control to each one in turn until all are done. Control is never passed to another peer activity, unless the *parent* itself delegates a call.

This example shows how an arbitrary number of similar subactivities can be started. Of course one may sequentially initiate a sequence of dissimilar subactivities by repeated use of the *activity* construct. If only one such sequence is required, the *coblock* statement can be used instead of *coloop*. Again, the parent waits until all of the subactivities have terminated.

To formally understand concurrent subactivities, we require an additional mapping *parent*:

$$parent : A \rightarrow A \tag{9}$$

The inverse mapping is:

$$child : A \rightarrow 2^A$$

and the closure is:

$$descendant(a) \equiv child^+(a) = \bigcup_{i \geq 1} child^i(a)$$

The events that must be handled differently from before are:

1. The *acceptance* of a queued request.
2. The execution of a *delegated* call.

3. The *starting* or *ending* of a subactivity.
4. *Breaking* out of a coblock or coloop.

Now, when an activity a spawns a child a' , the parent is suspended on a newly-created queue q_a , and the child immediately becomes active. The mappings *parent*, *active*, *status* and *susp* are updated in the obvious way, as well as A and Q . The new queue q_a may be opened (and closed) by the operation spawning the subactivity.

Subactivity calls and returns are handled as before. When a activity a delegates a call, however, the domain only becomes idle if a has no parent. Otherwise the domain becomes *blocked* on the parent. If the queue q_a is open, the parent may then become active again.

When a parent activity a *breaks* out of a coblock or coloop, it is suspended on the queue q_a , and the domain blocks on a . Furthermore, if a still has children, the queue q_a is closed.

If a domain d is blocked on activity a , we now allow d to accept requests from a or any of its descendants. As a consequence, d will not deadlock when it blocks on a and a is suspended on the closed queue q_a . This also means that when a subactivity delegates a call, the domain may switch either to the parent or to any of its other children or descendants.

Finally, when a subactivity ends, it opens its parent's queue if there are no other children. The parent will thus become active again when its children terminate.

The other problem that we have already alluded to earlier in this paper is that of medium-term mutual exclusion for an object. Objects provide short-term mutual exclusion through the activity mechanism. While a call to an operation is pending, no other calls can be serviced unless they are recursive calls. Delegation allows us to relax this criterion by enabling controlled switching between activities. If, on the other hand, we wish to ensure mutual exclusion for a number of calls, then we must delay all calls not related to the activity requiring the medium-term access. We can simulate this requirement using delay queues as in example 1 above, however this requires that all objects to be used during the critical section must be capable of being "acquired" or locked in this fashion. Appropriate delay queues must be present for all those objects.

Instead, we propose that the notion of *atomic actions* be incorporated into our framework. Atomic actions are a familiar concept from systems with many concurrent accesses to shared objects. There are programming languages that provide built-in constructs for designing atomic actions, such as *Argus* [Lisk83]. Since our objects are active, they already provide a low granularity of concurrency control. Medium-term mutual exclusion can be accomplished by convention if objects agree to delay competing activities for the duration of an *atomic* statement. The mechanism entails a form of *two-phase locking*, in which objects returning from a call of an operation go into a *ready* state in which they are willing to accept further calls from the calling activity, or either an *accept* or (possibly) an *abort* request.

Atomic actions may be nested, creating a special kind of subactivity. A *ready* object is willing to accept calls from the atomic activity that engaged it, or any of its subactivities. At each point that an atomic subactivity is engaged, an object may be required to *checkpoint* its state, if there is a possibility of the action being aborted. If the activity makes a *commit* request, the checkpointed state is discarded, and the object is released from the activity. Alternatively, if an *abort* request is made, the checkpointed state is restored, and the object is released. The checkpoints can be

made in a variety of ways, ranging from making a full copy of the previous state, to making careful partial copies, or logging the changes made. Of course, if it is known in advance that the atomic subactivity will not abort, there is no need to do any checkpointing.

It should be clear that atomic subactivities and delegation are incompatible, since one explicitly forbids switching of activities, and the other enables it. Delegation within an atomic subactivity would allow that object to be seen in an “inconsistent” state, thus violating the atomicity requirement. One may, however, delegate a call to an operation with an atomic action inside it, since the delegating object is not (at least initially) involved in that action.

Furthermore atomic subactivities may spawn concurrent subactivities which use delegation to switch amongst themselves within their atomic parent. Atomicity therefore does not preclude concurrency.

Example 9: An atomic action.

```

redraw : → ;
{
  var pos1, pos2 : position ;
  atomic {
    end[1].pstate := end[1].poid.state ;
    end[2].pstate := end[2].poid.state ;
    pos1 := end[1].poid.pos ;
    pos2 := end[2].poid.pos ;
    # code for drawing a line ...
  }
  return ;
}

```

In example 9 we see how an atomic action can be used by our *line* object of example 6 to enable it to synchronize with its two endpoints. The two independent *point* objects are “acquired” by the atomic subactivity, and not released until the scope of the atomic statement terminates. All possible concurrency is eliminated within this scope, and so the line object can be sure that it is synchronized with its endpoints. One may imagine more elaborate examples required nested atomic actions, and actions that may abort for various reasons.

Two additional mappings are required to formally capture atomic subactivities:

$$atomic : A \rightarrow \{true, false\} \quad (10)$$

and

$$ready : A \rightarrow 2^D \quad (11)$$

When an atomic subactivity a' is created, it is added to A , and the mapping $atomic$ is modified to reflect the addition of new subactivity. When a domain d returns from a call made within a' it must remain *blocked* instead of becoming idle, even if $returns(d, a') = 0$. Also, $ready$ is modified to

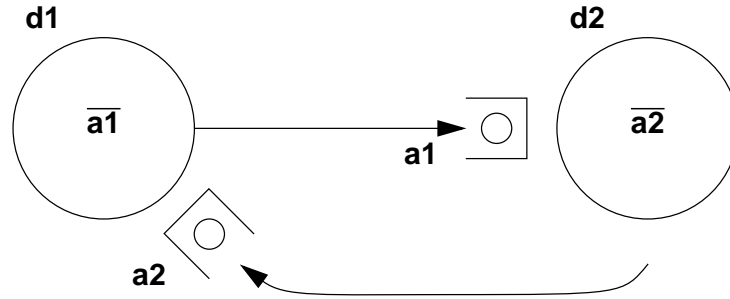


Figure 3: A simple deadlock.

reflect that d is waiting for a' to end. When a' finally commits (or aborts), the list of *ready* domains is released, and each becomes *idle* (or *blocked* on the parent, according to the rules described above).

In the next section we shall discuss the issue of deadlock within our model, and see how the notion of atomic actions can enable us to get a more complete picture of deadlock in certain cases.

5 Deadlock.

As is the case with any concurrent system, there is a possibility of deadlock between active objects. Consider, for example, the situation in figure 3, where some object in domain d_1 has called an operation of an object in d_2 . The activity a_1 in which the call was performed is suspended on a queue of d_2 . In addition, an object in d_2 has called an operation of some object in d_1 . The calling activity a_2 is suspended on a queue of d_1 . Neither object can become unblocked, and all activities suspended on their queues will remain so forever.

Typically deadlock is understood in terms of cycles in a *waits-for graph*, where processes and resources “wait for” each other [Holt72]. Since it is not obvious what the “processes” and “resources” are in our model, we shall re-interpret the notion of a waits-for graph in terms of activities, domains, queues and operations.

First we define four kinds of deadlock. We shall say that a domain is *deadlocked* if it is blocked on an activity and can never become unblocked. Similarly, we say that an activity is deadlocked if it is suspended and can never become active again. A queue is deadlocked if it is closed and can never again become open. An operation is deadlocked if it can never be executed.

Notice that a domain is only deadlocked if it is blocked and *cannot* become unblocked. We do not guarantee that it *will* become unblocked. Similarly for activities, queues and operations. We make the following observations:

1. If a domain is blocked on an activity that is not deadlocked, then that domain is not deadlocked either.
2. If an activity is suspended on an open queue of a domain that is not deadlocked, then that activity is not deadlocked either.

3. If a queue is closed, and there is some operation that will open it attached to another queue that is not deadlocked, then that queue is also not deadlocked.

In the first case, “livelock” is ruled out: if an activity is active, or can eventually become active, then we assume it can return to the blocked domain. Activities do not run forever. In the second case, a fairness condition is assumed: if a domain is not deadlocked, it can eventually become idle often enough to service all activities suspended on its open queues. Finally, we assume that a closed queue that can be opened by an operation attached to a non-deadlocked queue eventually *could* be opened by that operation. In other words, only circular forms of deadlock are considered.

We say that a domain d running or blocked on activity a *waits for* (or is “assigned to”) that activity. Of course, when an object delegates a call, the domain is not blocked, and so does not wait for the activity that was running to return. An activity a suspended on an open queue of a domain d waits for that domain. Furthermore, if activity a is suspended on a closed queue q , then it waits for that queue. A closed queue waits for the *set* of operations that may open it. Finally, by analogy with activities, an operation attached to an open queue waits for its domain, and an operation attached to a closed queue waits for that queue.

Formally, the edges of the waits-for graph are:

$$\begin{aligned}
 E = & \{(d, a) \mid d \in D_B, a = \text{blocked}(d)\} \\
 & \cup \{(d, a) \mid d \in D_A, a = \text{active}(d)\} \\
 & \cup \{(a, d) \mid a \in A_S, \text{susp}(a) \in Q_O \cap Q_d\} \\
 & \cup \{(a, q) \mid a \in A_S, q = \text{susp}(a) \in Q_C\} \\
 & \cup \{(q, p) \mid q \in Q_C, p \in \text{open}(q)\} \\
 & \cup \{(p, d) \mid p \in P_d, \text{uses}(p) \in Q_O\} \\
 & \cup \{(p, q) \mid \text{uses}(p) \in Q_C\}
 \end{aligned}$$

Observe that the waits-for graph is bipartite, with domains and closed queues in one partition, and activities and operations in the other partition. Waits-for edges may only exist between nodes in different partitions. Also observe that only closed queues may have more than one outgoing waits-for edge.

Following the proof of [Holt72], it is now easy to see that a deadlock can only occur if there is a cycle in the waits-for graph, and that a knot in the graph is a sufficient condition for deadlock. Observe that domains can be mapped to “reusable resources” which when blocked are assigned to activities, and that closed queues can be mapped to “consumable resources” with operations as their “producers”. Activities can be mapped to processes. Also note that a cycle not involving closed queues is a sufficient condition for deadlock, since a knot in which every node has only one outgoing edge is precisely a cycle.

At this point we should like the reader to note that our line of reasoning thus far explicitly ignores the fact that operations that can open queues may do so conditionally. There is no notion of the operation “waiting for” an event associated with some activity or domain. In the previous sections, examples 1 and 7 included operations of this sort. In example 1, the *available* queue was opened only when the “holder” of the resource released it. In example 7, the *doneQueue* was opened when the various subactivities completed their tasks. In all other examples, operations that opened queues did so unconditionally, once they were called.

If we formalize the notions of atomic actions and subactivities then we can also understand deadlock in these cases. In the case of concurrent subactivities, we add the following edges to the waits-for graph:

$$\{(d, a) \mid d \in D_B, a = \text{descendant}(\text{active}(d))\}$$

This captures the idea that subactivities can wake up a blocked domain.

In the case of atomic actions, there is no need to add new edges to the waits-for graph since domains continue to be blocked on atomic subactivities as long as they are *ready*. The redefinition of the way returns are handled in the context of atomic subactivities is sufficient to capture possible deadlocks. Nested atomic subactivities are already handled by the addition of the *descendant* edges into the waits-for graph.

Once we have a formal notion of deadlock, it is possible to cope with it in a variety of ways, either attempting to detect it when it occurs (or is “likely” to occur) and using some procedure to break the deadlock, or by avoiding the possibility of deadlock altogether [Bern81 Bada86].

Thus we see that an explicit notion of atomic actions and of subactivities not only allows us to more concisely express medium-term mutual exclusion and controlled parallelism in a programming language, but also enables us to get a clear handle on the deadlock issues involved. Delay queues continue to be useful to express the notion of triggering activities waiting for a particular event to occur.

6 Other Models.

It is instructive to compare our activity model to other formalisms for expressing communication and synchronization between concurrent processes. The differences are mainly due to our goal of attempting to understand concurrency in an object-oriented setting where objects themselves are the active entities. Processes as such do not exist in our model (unless one argues that activities are really processes). Each activity, however, is effectively part of the object whose reflex (top operation) started it. All communication in our model is based on remote procedure calls, and all synchronization is based on the idea of activities being suspended on queues.

Our model is at least as general as semaphores, since we can capture mutual exclusion at the granularity of an operation, or of a sequence of operations bundled as an atomic action. We can also capture semaphores directly using delay queues, as shown in example 1.

A more interesting comparison is with monitors [Hoar74]. Each domain is, after all, monitor-like, providing exclusive access to a set of operations of a number of abstract “objects”. We do not, however, divide our world into programs and monitors. Everything is an object, and all objects can be accessed in the same way. Objects can thus play either a passive (server) role, or an active role (calling other objects). Furthermore, activities cannot truly be seen as processes, since they have no memory – they represent only a thread of control between active objects.

We observe the following concrete differences between objects and monitors:

1. Monitors are passive, and allow several processes to enter them at once (until they encounter a *wait*), whereas domains can only be active on one activity at a time.

2. *wait* and *signal* suspend processes running in a monitor, whereas *close* and *open* do not suspend activities.
3. Activities may only be switched upon a *call* or a *return*, even when delegation is used, whereas *wait* and *signal* permit switching at arbitrary times.

One can, of course, simulate the behaviour of a *wait* by delegating a call to an operation attached to a closed queue, as was done in various triggering examples. We suggest that the activity model is better suited to an object-oriented environment, and that the restrictions concerning the points at which an activity may be suspended result in more structured code with no less generality.

Another interesting comparison is with guarded commands [Dijk75], as manifested in Hoare’s CSP [Hoar78]. Parallel commands can be simulated by creating subactivities with the *coloop* and *coblock* constructs. Input and output guards have no direct analogy since the remote procedure call is the model for all communication. A call to an operation can simulate these, however. Note that “read” and “write” can be simulated using triggers, as in example 3, presented earlier in this paper. An alternative command can be simulated by modifying example 7 so that each subactivity spawned attempts to evaluate one of the guards. The parent activity is suspended on a delay queue, and is triggered when the *first* guard succeeds. If all the subactivities return without success, the parent is still triggered, but failure is indicated. We have not argued in this paper for an explicit alternative construct for selecting between subactivities.

The main difference with CSP is that objects can support a set of operations more general than simply *input* and *output*. Furthermore, processes cannot share variables in CSP, but can only communicate using input and output commands. Objects, however, may switch between activities, and activities can thus “share” common objects.

There are other message-passing models that are of interest, including Thoth [Gent81] and Actors [Hewi77a Hewi77b]. It should be clear that our model can be mapped to one supporting a notion of message-passing processes, since domains resemble processes, and requests associated with suspended activities resemble messages. To do so would require the adoption of conventions for message-passing that associate an activity with each message, and permit only certain kinds of messages corresponding to *start*, *call*, *return*, *commit*, *abort*, and so on. Note that our use of queues suggests asynchronous message-passing in spite of the fact that a blocking send is the default for a *call* message. On the other hand, every message is associated with an activity, and since an activity can only be suspended at one place at a time, the number of messages in the system is always bounded by the number of activities. (The exception would be for *commit* and *abort*, which can be sent in parallel.)

7 Concluding Remarks.

We have presented a model for understanding communication and synchronization of *active objects* in an object-oriented environment. The model forms the foundation for concurrency in the object-oriented programming language *Hybrid*.

The model supports a notion of active objects partitioned into concurrent entities called *domains*.

Each domain corresponds to a single top-level object, possibly composed of other objects. Each object supports a set of operations, which forms that object's interface.

Concurrency is modelled through the notion of *activities*, which are tokens representing a single thread of control between active objects. Communication is modelled on remote procedure calls, with a *call* of an operation resulting in an eventual *return*. New activities are created by invoking special operations called *reflexes*. Mutual exclusion is provided at the level of operations since objects can only be active on behalf of a single activity at a time. The notion of *delegation* is introduced as a mechanism for allowing more flexible interleaving of activities whenever an object calls an operation.

Delay queues are introduced as the mechanism for delaying calls to operations that are temporarily not available. Various important forms of *triggering* can be captured by this mechanism, including triggers that wait for multiple events.

Concurrent subactivities and atomic actions are added to this framework to more compactly express solutions to certain problems that are unwieldy when solved using only reflexes and delay queues. This also allows us to capture the deadlock situations that may arise when using subactivities and atomic actions.

Numerous examples are given to illustrate how these constructs may be used in an object-oriented programming language to solve various concurrency and triggering problems. Comparisons are drawn with other models for communicating processes, notably monitors and CSP.

8 Acknowledgements.

The work described in this paper was carried out with the assistance of a postdoctoral fellowship granted by the Natural Sciences and Engineering Research Council of Canada. Its support is gratefully acknowledged.

References

- [Bada86] D.Z. Badal, "The Distributed Deadlock Detection Algorithm", ACM Transactions on Computer Systems, vol. 4, no. 4, pp. 320-337, Nov 1986.
- [Bern81] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, vol. 13, no. 2, pp. 185-221, June 1981.
- [Dijk75] E.W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs", CACM, vol. 18, no. 8, pp. 453-457, Aug 1975.
- [Gent81] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software – Practice and Experience, vol. 11, pp. 435-466, 1981.
- [Hewi77a] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, vol. 8, no. 3, pp. 323-364, June 1977.

- [Hewi77b] C. Hewitt and H. Baker, “Laws for Communicating Parallel Processes”, *Information Processing 77*, pp. 987-992, North-Holland, 1977.
- [Hoar74] C.A.R. Hoare, “Monitors: An Operating System Structuring Concept”, *CACM*, vol. 17, no. 10, pp. 549-557, Oct 1974.
- [Hoar78] C.A.R. Hoare, “Communicating Sequential Processes”, *CACM*, vol. 21, no. 8, pp. 666-677, Aug 1978.
- [Holt72] R.C. Holt, “Some Deadlock Properties of Computer Systems”, *ACM Computing Surveys*, vol. 4, no. 3, pp. 179-196, Sept 1972.
- [Lisk83] B. Liskov and R. Scheifler, “Guardians and Actions: Linguistic Support for Robust, Distributed Programs”, *ACM TOPLAS*, vol. 5, no. 3, pp. 381-404, July 1983.
- [Nier86] O.M. Nierstrasz, “What is the ‘Object’ in Object-oriented Programming?”, *Proceedings of the CERN School of Computing*, Renesse, The Netherlands, Sept 1986.
- [Nyga86] K. Nygaard, “Basic Concepts in Object Oriented Programming”, *ACM SIGPLAN Notices*, vol. 21, no. 10, pp. 128-132, Oct 1986.
- [Stef85] M. Stefik and D.G. Bobrow, “Object-Oriented Programming: Themes and Variations”, *The AI Magazine*, Dec 1985.