Active Objects in Hybrid[1]

Oscar Nierstrasz[2]

Abstract

Most object-oriented languages are strong on reusability or on strong-typing, but weak on concurrency. In response to this gap, we are developing *Hybrid*, an object-oriented language in which objects are the active entities. Objects in Hybrid are organized into *domains*, and concurrent executions into *activities*. All object communications are based on remote procedure-calls. Unstructured *sends* and *accepts* are forbidden. To this the mechanisms of *delegation* and *delay queues* are added to enable switching and triggering of activities. Concurrent subactivities and atomic actions are provided for compactness and simplicity. We show how solutions to many important concurrent problems, such a pipelining, constraint management and "administration" can be compactly expressed using these mechanisms.

1. Introduction.

The idea of applying object-oriented approaches to the programming of concurrent systems has been gaining popularity in recent years. Some of the object-oriented languages that have mechanisms for dealing with concurrency are: Orient84/K [Toko86], ConcurrentSmalltalk [Yoko86], ABCL/1 [Yone86] and Emerald [Blac86]. It is very appealing to imagine environments of highly independent active objects for solving concurrent and distributed problems. Previous work in object-oriented systems, however, has emphasized reusability of objects, as in Smalltalk [Gold83], Objective C [Cox86], C++ [Stro86] and Lisp with flavors [Wein81], and, more recently, strong-typing of objects in Smalltalk [John86] and in Owl [Scha86], with little or no attention paid to concurrency. Work on concurrent "objects" has been mostly limited to actor languages [Hewi77, Byrd82, Ther83] and other kinds of systems supporting message-passing processes, such as Thoth [Gent81].

Hybrid is an object-oriented programming language in which objects themselves are the active entities. We believe that, with some care, concurrency can not only be easily accommodated within an object-oriented framework, but that the object-oriented approach is exceptionally well-suited for structuring concurrent activity. In this paper we shall attempt to demonstrate this by presenting an object-oriented model for concurrency, a small set of primitives which can be embedded into a language such as *Hybrid*, and a number of examples indicating how these primitives can be used to compactly express solutions to various concurrent programming problems.

Concurrency models have traditionally taken one of two views for communication and synchronization of concurrently executing processes: either that of processes synchronizing accesses to a shared memory, or that of processes communicating by passing messages (whether synchronously or asynchronously) [Andr83]. If we accept that a programming paradigm and a con-

1. In OOPSLA '87 Proceedings, ACM SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 243-253.

2. Author's address: Centre Universitaire d'Informatique, 12 rue du Lac, CH-1207 Geneva, Switzerland. E-mail: oscar@cui.unige.ch, oscar@cgeuge51.bitnet. Tel: +41 (22) 787.65.80. Fax: +41 (22) 735.39.05.

currency model should be mutually supportive, then it is clear that we must take care in choosing how to view concurrency in the context of object-oriented programming.

The fundamental concept of the object-oriented approach is that objects are encapsulations, providing access to hidden data through a set of visible operations.[1] Objects can thus be viewed as "servers," responding to requests via message-passing. A concurrency model that views objects as active entities passing messages has the advantage that message-passing can be used both for communication *and* as a basis for synchronization. When objects are viewed as passive entities, on the other hand, one must use additional mechanisms for synchronizing access to objects, such as semaphores (as in Smalltalk), or *waits* and *signals* as in monitors [Hoar74]. Furthermore, shared memory models do not generalize well to distributed environments. The advantage of the shared memory view over message-passing, however, is that the threads of control are explicit, whereas messages do not necessarily entail a transfer of control.

We propose a hybrid model in which objects are potentially active entities that communicate by message-passing. Message-passing operations are based on remote-procedure calls, and always entail either a transfer of control, or the creation of a new thread of control, or *activity*. A "unit of concurrency" is called a *domain*, and may contain one or more top-level objects. In addition, we require mechanisms for deciding when an operation may be invoked, thus effectively scheduling activities attempting to invoke those operations, and when to relax the remote-procedure call protocol by permitting the interleaving of activities.

Briefly, the concepts we define are as follows:

1. *domain:* a "unit of concurrency" (i.e., a process)
2. *activity:* a thread of control
3. *reflex:* an operation that starts a new activity
4. *delay queue:* a scheduling mechanism to delay calls
5. *delegation:* a mechanism for permitting the interleaving of activities

In addition, we show how this framework can be consistently extended to accommodate *concurrent sub-activities* and *atomic actions*.

To illustrate the use of these mechanisms, we give several examples of objects that encapsulate:

1. pipelining
2. postactions
3. administration [Gent81]
4. triggering
5. constraints

1. The notion of class inheritance, though crucial to object-oriented programming, is secondary to the idea of encapsulation.

Furthermore, with the higher-level constructs for composing concurrent subactivities and atomic actions, it becomes possible to structure sets of related activities.

The examples are defined in Hybrid [Nier87a], a strongly-typed object-oriented programming language that supports the concurrency model and primitives described in this paper. We shall explain the non-obvious aspects of Hybrid during the presentation of the examples.

2. Domains and Activities.

The description that is often given of objects communicating by passing "messages" can seduce one into thinking of objects as independent message-passing processes. Listeners are often disappointed when they discover that, although objects are in some ways independent, they are not generally concurrent.

If we try to imagine how concurrency would fit into an object-oriented world, it would be quite natural to start by proposing that every object be a message-passing process. Since objects encapsulate a set of operations and a hidden representation, all we need is to structure the message-passing operations to conform to a remote procedure-call protocol: every message is either a *call* to an operation or a *return*.

This solution has a terrible problem, however, which becomes apparent when we recall that objects are structured. Would this then mean that every instance variable would be a process, and every instance variable of every instance variable a process? And finally every bit of every byte a process? It is clear that this approach (though perhaps yielding to an appealing formal "purity") has crippling practical drawbacks. There is simply too much concurrency.

To tackle this problem, rather than defining *a priori* how much concurrency there will be, we allow the programmer to define the granularity of concurrency by providing a single unit of concurrency that we call a *domain*. Intuitively, a domain bundles a collection of active objects into a single message-passing process. Typically there will be one "root" (top-level) object in a domain and a number of "sub-objects" which are instance variables and dynamically instantiated objects.

More concurrency is achieved by splitting objects up into more domains. Objects in the same domain are *dependent*. An object can only communicate with an independent object if it knows its name, or *object identifier*, which it may store as an instance variable. To create a new domain, a new object is created and *exported*. The new object becomes the root of the new domain. (If a new object is not exported, it simply becomes a dependent of the domain that created it.)

Since a domain is like a process, it can only be "doing" one thing at a time. That means that at any time at most one object may be active in a given domain. We also need to get a handle on what is happening when a domain is active. We call a single thread of control an *activity*. An activity is *started* by invoking a *reflex*. A reflex is a special kind of "main" operation in Hybrid which, when invoked, starts a new activity while allowing the parent activity to continue independently. The *start* message is a variation on the *call* message where no return is expected.

From then on, the activity passes as a thread of control from object to object in a series of *call* and *return* message-passing operations. The activity is like a token, and is always in a unique location. When one independent object calls another, the activity actually *moves* from one domain to the other.

There is a default degree of mutual exclusion provided by the remote-procedure call model. Domains may be *idle, active* or *blocked*. Objects in idle domains are capable of handling requests. Objects in active domains are busy, and can only handle requests arising from the current activity. When an object *calls* an independent object, its domain *blocks*, and objects in that domain can only accept call or return requests from that same activity. (Since a call was made, presumably there will eventually be a return.) In effect, calls are by default blocking sends. An *active* or *blocked* domain therefore causes all incoming requests to wait.

Activities may either be *active*, that is, when they are running in some active domain, or *suspended*, when they are bundled in a *start*, *call* or *return* message that cannot (yet) be accepted. The "location" of an activity is therefore always either a domain (when it is active), or a message queue (when it is suspended). Furthermore, every message is always associated with an activity, and therefore the number of messages in the system is always bounded by the number of activities. This is an obvious consequence of the fact that the total number of activities is equal to the sum of the number of active domains and the number of suspended activities (each represented by a message).

Also note that the total amount of potential parallelism is bounded by the lesser of the number of domains and the number of activities. One cannot have more running activities than there are domains. But idle domains do not count, and so there also cannot be more domains running than there are activities.

The activity model we have described thus far is basically a remote procedure-call model with blocking sends for *call* messages, and non-blocking sends for *starts* and *returns*. The basic model can be easily mapped to a world with structured objects that support a visible interface and a hidden realization. Objects cannot be seen in an inconsistent state while they are assigned to an activity.

We shall now argue for two additional mechanisms that enable more flexible interaction between activities, while maintaining consistency with the remote procedure-call message-passing model and the principle of encapsulation adhered to by objects.

3. Delay Queues.

Objects may not always be ready to provide a service encapsulated in one of their operations. This can be handled either by having the object return an error, in which case it is up to the caller to try again or "poll" the object, or the caller can be *delayed* until its request can be serviced, thus obviating the need for polling.

A mechanism for delaying calls allows an object to schedule requests based on its own internal state. It also provides a way for activities to be *triggered* on the basis of other events that take place. Event-driven behaviour can thus be captured concisely.

In Hybrid, objects may delay calls by associating an operation with a *delay queue*. In the realization of the operation, it is declared as *using* the queue. This has the effect that calls to that operation are only accepted if the queue is *open*. If the queue is *closed*, the calls are delayed. Delay queues can therefore be viewed as a restricted form of *guard* [Dijk75], the only allowable boolean being: “is my delay queue open?” Objects service delayed requests on open queues before accepting new requests. (There is a permanently open queue for *return* messages and for calls to all other operations that do not use a delay queue.)

A delay queue is an instance variable of an object, and may only be opened or closed by the operations of that object. Typically, one operation will detect a condition that causes it to close a queue, and another may detect the condition that causes it to be opened again. The delay queue variable typically represents the availability of a resource or the precondition for performing an operation. Delay queues may be shared by several operations, but an operation can use at most one queue. (Some care must be taken in designing delay queues, or deadlock situations may arise: for example, if all operations use the same queue and that queue is closed, the object will be deadlocked since a call to an operation that opens the queue can never be accepted.)

As a first example, consider a *pipeline* as implemented by a circular bounded buffer in Hybrid (see example 1).

“Programs” in Hybrid are definitions of object types, in this case the type pipeline. The pipeline object type has two type parameters, itemType, the type of the objects the pipeline manages, and bound, which is used to index the array containing the buffered items. The bound parameter is constrained to be an enumerated type (enumType). Comments are preceded by a number sign (“#”).

A pipeline is defined with the abstract constructor in terms of the operations get and put. (Other type constructors exist in Hybrid for defining arrays, enumerated types, subtypes, and so on). Each operation includes a specification of its argument types (before the arrow) and return value types (after the arrow). Here, put and get accept and return, respectively, an object of the parameter type itemType.

A pipeline object has delay queues notFull and notEmpty as instance variables, initialized respectively to the values *open* and *closed*. As a consequence, calls to get are initially delayed since it uses the notEmpty queue. The getIndex and putIndex variables index the circular buffer, and are initialized to the first element of the enumerated type object bound.

The trigger conditions notFull and notEmpty are managed by the operations get and put. Whenever an event occurs that causes a trigger condition to become true or false, the appropriate delay queue is respectively opened or closed. This principle can be used in far more elaborate triggering examples, some of which we shall describe in this paper.

An instance of pipeline can be used to pass objects from one activity to another. Objects taking part in a producer activity *a* can use the put operation of the pipeline object to buffer items for a consumer activity *b*. The pipeline object alternatively services requests for activities *a* and *b*. If either activity gets “ahead of” the other (i.e., if the buffer becomes empty or full), the pipeline will delay one activity or the other by closing one of its delay queues. Notice that the pipe-

```
# definition header with parameters:
type pipeline of ( itemType , bound :< enumType ) :
        # interface specification:
        abstract {
                put : itemType -> ;
                get : -> itemType ;
        } ;

# realization of operations and instance variables:
private
{
        # instance variables and initialization
        var notFull <- open , notEmpty <- closed : delay ; # delay queues
        var bufferSpace : array [ bound ] of itemType ;
        var getIndex <- bound.first , putIndex <- bound.first : bound ;

        # calls are delayed if notFull is closed
        put : ( item : itemType ) -> ;
                uses notFull ;
        {
                bufferSpace [ putIndex ] := item ;
                switch ( putIndex ) {
                        case bound.last { putIndex := bound.first ; }
                        default { putIndex.inc ; } # increment
                }

                if ( putIndex =? getIndex ) {
                        notFull.close ; # full, so delay put()
                }
                notEmpty.open ; # trigger delayed calls to get()
                return ;
        }

        # calls are delayed if notEmpty is closed
        get : -> itemType ;
                uses notEmpty ;
        {
                var item : itemType ;
                item := bufferSpace [ getIndex ] ;
                switch ( getIndex ) {
                        case bound.last { getIndex := bound.first ; }
                        default { getIndex.inc ; } # increment
                }

                if ( getIndex =? putIndex ) {
                        notEmpty.close ; # empty, so delay get()
                }
                notFull.open ; # trigger delayed calls to put()
                return ( item ) ;
        }
}
```

Example 1: A pipeline

line does not have an activity of its own, but plays a passive role switching between activities *a* and *b*.

It is instructive to compare our pipeline object to implementations of bounded buffers using monitors [Hoar74, Wirt83] and CSP [Hoar78]. The monitor approach takes a shared memory view of objects, whereas the CSP approach uses message passing and guards to synchronize objects. Both of these solutions superficially resemble our pipeline object, however there are some important differences.

In the case of monitors, since objects are viewed as shared memory, *wait* and *signal* must be used to implement mutual exclusion as well as scheduling; in a message-passing world, mutual exclusion is not an issue, and one only has to worry about which message to accept next. Next, one is free to use *wait* and *signal* in a non-structured fashion, whereas delay queues can only be used at the point where one accepts a *call* message. Finally, *wait* and *signal* may entail a transfer of control, whereas *open* and *close* never do.

If we compare the use of delay queues in our pipeline object to the CSP solution of a bounded buffer, as proposed by Hoare, the differences are less profound. In fact, guarded commands appear to be strictly more general: guards may be arbitrary boolean expressions, whereas delay queues are only tested for being *open* or *closed*; guarded commands may be used anywhere, whereas delay queues only affect acceptance of messages; guards may be placed on output expressions as well as input expressions (since message-passing is synchronous in CSP); and one may explicitly name both input and output processes in CSP, whereas objects in *Hybrid* cannot do so.[1] Despite the restrictions we introduce, it is possible to solve many useful classes of concurrent problems using the mechanisms we propose.

In addition to triggering, delay queues can be used to implement *postactions*. By “postaction,” we mean an action that may be performed in response to a call to an operation, but which may proceed after the operation returns. Examples are consistency checking, garbage collection, reorganization of data structures, redisplaying of images and reformatting of text..

```
var ready : delay ;

delete : ( key : keyType ) -> itemType ;
        uses ready ;
{
        # code to mark item as deleted ...
        ready.close ; # delay future calls
        self ! cleanup ; # start new activity
        return ( item ) ;
}

# postaction
reflex cleanup : ;
{
        # code to clean up data structures
        ready.open ; # trigger delayed calls
        end ; # doesn't return
}
```

Example 2: Postactions

1. On the other hand, the collection of available processes and the input and output channels are statically defined in Hoare’s original description of CSP.

In the example we have a delete operation which deletes an item associated with a key value and returns the value of the item deleted. The delete operation starts a cleanup postaction which cleans up data structures and possibly does compression. The postaction is encapsulated as a reflex, thus causing a new activity to be created when it is invoked. Note that reflexes in Hybrid are invoked with an exclamation mark (self!cleanup) instead of a period to clearly denote the creation of a new activity

In order to prevent further calls to delete intervening before the cleanup reflex can start, the delete operation is associated with a delay queue (called ready) which is closed before cleanup is invoked. Only when the postaction has terminated is the delay queue opened. Notice the use of the end statement to terminate an activity, since a reflex does not “return” anywhere.

Delay queues provide the programmer with control over the scheduling of invocation requests to operations and reflexes. As a consequence, event-driven behaviour or “triggering” can be easily expressed by delaying calls to operations which are later triggered by the awaited events. Delay queues are not adequate for expressing more flexible interleaving of several activities within a single domain. We introduce the notion of “delegation” in the following section in order to address this problem.

4. Delegation.

Mutual exclusion is provided at the level of domains by allowing no more than one activity to be active in a domain at once. (By definition a process cannot be doing two things at once.) A further level of mutual exclusion is enforced by the semantics of remote procedure-calls since an object that calls an independent object causes its domain to become blocked rather than idle. Only requests related to the activity can unblock the domain in order to prevent other activities from seeing objects in the domain in an inconsistent state.

Sometimes it is convenient to relax this constraint. For example, an object that functions as an “administrator” [Gent81] may delegate a task to an independent “worker” object. The administrator, if it is not in an inconsistent state, may then wish to accept further requests while it is waiting for the worker to finish its job.

An expression in Hybrid may be bracketed with the delegate construct, which effectively causes the expression to be evaluated asynchronously. First the *target* and *argument* subexpressions are evaluated, and then the target operation is invoked *without* causing the domain to block. If the target is an independent object then the domain will be free to accept requests from other activities. When the target eventually returns, execution may resume from the point of the delegated call as soon as the domain is idle (i.e., the target’s *return* message must now queue up for the domain as would any other request).

If the target is a dependent object (i.e., in the same domain), then the effect is to permit the domain to switch activities, depending, of course, on whether or not any activities are waiting to get in, and whether the target operation has a delay queue or not.

For example, in:

```
delegate ( target.operation(argument) )
```

The *target* and *argument* expressions are evaluated, then a *call* message is sent to the *target*. The caller may proceed with *another* activity, if there is one waiting. (If there are no suspended activities to switch to, delegation has no observable effect.) When the *target* returns *and* the caller is idle, the calling activity may continue executing from the point where the call was made. The value of a delegated expression is exactly the same as an ordinary (non-delegated) expression.

In an implementation, a domain would have to keep track of several execution stacks, one for each activity with a pending delegated call. These activities continue executing in other domains, in parallel, if possible, and resume in the calling domain when the target operation completes.

In the following example we see how an administrator might delegate jobs to “worker” objects. The target and argument expressions here are both trivial, being respectively worker and newJob. We assume that the object ids (*oid*s) of the workers identify independent objects that may execute concurrently with the administrator. The call to the doJob operation is achieved by passing a *call* message to the worker. The administrator’s domain, instead of blocking, is now free to switch to another activity, if there is one waiting to get in.

```
var available : pipeline of ( oid of jobDoer, workerRange ) ;

assign : ( newJob : job ) -> job ;
{
        var doneJob : job ;
        var worker : oid of jobDoer ;

        # get a worker from the pipeline,
        # but don’t block if none available
        worker := delegate ( available.get ) ;

        # delegate the work, and switch to waiting calls
        doneJob := delegate ( worker.doJob ( newJob ) ) ;

        # release the worker
        available.put ( worker ) ;
        return ( doneJob ) ;
}
```

Example 3: Delegation

Delegation is used twice in this example. In the first case, the administrator must not block if there are no workers available in the pipeline. The call to available.get is therefore delegated, allowing the administrator to switch to another activity, possibly one that puts a worker back into the pipeline. When the call returns, the value returned by get is assigned to worker, just as if the call had not been delegated.

The call to worker.doJob is also delegated, since there is no reason for the administrator to wait while the worker handles the request. When the result is ready and the administrator is idle, the worker is made available again, and the result is returned to the original caller. (If there are more workers than the pipeline can hold, then the call to put should be delegated as well.)

One may imagine other kinds of administrators that use some other criterion than load-balancing to distribute jobs to workers. In a case where the delegated expression is directly returned, as in:

```
return(delegate(worker.doJob(newJob))) ;
```

we could optimize by substituting:

```
forward(worker.doJob(newJob)) ;
```

In this case the worker would not return to the administrator but return directly to the original caller, thus eliminating a message-passing operation. This optimization is relatively unimportant in the example we have given here, but the savings can be substantial in a setting where a large number of workers cooperate to produce a result. Consider an object-oriented implementation of a search tree in which nodes forward a call to one of their children to search for a particular entry. The node which determines success or failure may return directly to the caller rather than being forced to pop all the way back up the tree [Hogg87]. We provide this option in Hybrid, although it is not formally necessary.

5. Constraints.

Delegation used in conjunction with delay queues allows one to express *constraints*. By a constraint we mean a predicate that is expected to hold over a collection of objects. Examples are graphical constraints such as the requirement that the endpoint of one line be coincident with the midpoint of another line. Graphical constraints of this variety are managed in *ThingLab*, a "simulation laboratory" written in Smalltalk [Born81]. Another example is that of a database constraint which is checked periodically, or after every update.

Especially appealing in an environment like ThingLab is the possibility of being able to dynamically create and destroy constraints. This suggests that one might like to encapsulate constraints as objects themselves. (Of course, objects with static built-in constraints can also be of interest.) Such a constraint object would have to wait for a set of events, namely the modification of any of the constrained objects. The approach we suggest is to have the constraint object create one suspended activity for each awaited event. In order to be able to wait for multiple events, each of these activities *delegates* a call to an operation that returns when the constraint must be checked.

In the following example we see part of such a constraint object with an identify operation that constrains two points to coincide. (Each point might in turn be the endpoint of a line, or part of some other more complex graphic object.) The identify operation spawns two activities by invoking the connect reflex twice. These activities may start, one after the other, when identify returns. Each activity waits for one of the two points to be modified, and then re-establishes the constraint that the points coincide. The call to awaitUpdate must be delegated in order to allow the constraint object to switch to the second connect after the first has been set up.

The awaitUpdate operation requires some care. In order to be well-defined, one must specify the previous state with respect to which updates must be considered. Since constraints may themselves trigger updates, it is not possible to guarantee that all constraints will see all of the

```
# identify points p1 and p2
identify : ( p1 , p2 : oid of updateablePoint ) -> ;
{
        # set up activities
        # (the state is a “timestamp”)
        self ! connect ( p1 , p2 , p2.state ) ;
        self ! connect ( p2 , p1 , p1.state ) ;

        # make sure the points are in sync
        p1.moveTo ( p2.pos ) ;
        return ;
}

# connect point p1 to point p2
reflex connect :
        ( p1, p2 : oid of updateablePoint ; state : updateCount ) ;
{
        var newPos : point ;

        # wait for point p2 to change state
        ( newPos , state ) := delegate ( p2.awaitUpdate ( state ) ) ;

        # moveTo() has no effect if the new position
        # already coincides with the old
        p1.moveTo ( newPos ) ;
        # re-establish the constraint
        self ! connect ( p1 , p2 , state ) ;
        end ;
}
```

Example 4: Constraints

states of the objects they are monitoring (at least not in the setting we describe here). A state counter is therefore explicitly managed for each updateablePoint object (as opposed to a point object which maintains no such counter). The updateablePoint object must examine the previous state argument before determining whether the calling activity must be delayed or not. If so, it then forwards the call to its awaitNext operation, which is delayed until a state change takes place.

The wasUpdated reflex is used by update operations to close the wasUpdated delay queue after all the activities suspended on that queue have been triggered. Note that since this reflex itself uses the wasUpdated delay queue, the effect of the request is to “flush” the queue. Old calls to awaitNext will precede the wasUpdated request, whereas new calls will follow it.

6. Concurrent Subactivities.

It is possible to structure activities hierarchically to some extent using delegation and delay queues. A parent activity must create several child activities, each of which updates a counter when it terminates. The parent then suspends itself on an operation that is triggered when all the children are done. The approach is workable but results in rather clumsy code. Furthermore, one would like to be able to *explicitly* express the idea that a domain may be blocked on a *set* of subactivities rather than just a single activity. This enables more flexible switching between the subactivities, without allowing other, unrelated activities into the domain.

```
var pos : point ;
var updates := 0 : updateCount ;
var uponUpdate : delay ;

awaitUpdate : ( previous : updateCount ) -> (point, updateCount) ;
{
        # check if an update has already taken place
        if ( previous <? updates ) {
                return ( pos, updates ) ;
        }
        else {
                forward ( self.awaitNext ) ;
        }
}

# in sync, so "next" state is well-defined
awaitNext : -> ( point, updateCount ) ;
        uses uponUpdate ;
{
        # triggered by any update operation
        return ( pos, updates ) ;
}

# invoked by update operations to "flush" calls to awaitNext
reflex wasUpdated : ;
        uses uponUpdate ;
{
        uponUpdate.close ;
        end ;
}
```

Example 5: Awaiting updates

In Hybrid one expresses this through the coloop and coblock constructs and the activity construct. A coloop (executed repeatedly) or a coblock (executed just once) creates a scope within an operation that permits new concurrent subactivities to be created. The activity construct creates a new child activity which takes over from the parent until it either terminates or executes a delegated call. When the parent exits the coloop or coblock, it waits until all its children have terminated. Then the parent activity may continue.

The scheme is hierarchical, and supports the notion of nested subactivities. Note that at no time may a parent and a child be simultaneously active within the same domain. Each activity is a unique thread of control and cannot "share" a domain with another activity. Concurrency (and hence potential parallelism) enters the picture when a subactivity delegates a call to *another* domain, thus allowing its parent or any of its peers to continue in the domain it has just left.

In example 6 we see how a coloop can be used to delegate jobs to an array of worker objects. Each time the parent enters the activity statement, control is switched to a new child. When the child delegates its call to a worker, control is returned to the parent (or to a waiting peer). When the parent breaks out of the coloop, it is required to wait at the end of the coloop scope till all the children have terminated.

```
var n : workerRange ;

n := workerRange.first ;
coloop {
        activity {
                doneJob[n] := delegate ( worker[n].doJob ( newJob[n] ) ) ;
        }
        if ( n <? workerRange.last ) { n += 1 ; }
        else { break ; }
} # wait here ...
```

Example 6: Concurrent subactivities.

In this example we created a number of similar, parameterized subactivities. The coblock construct is sufficient if a sequence of dissimilar subactivities is required. In this case one would make repeated use of the activity construct within the coblock rather than the single instance in the example. The coblock used in this fashion is comparable to the cobegin statement in CSP [Hoar78] and Argus [Lisk83].

7. Atomicity.

Atomic actions can be very useful in a distributed environment [Lisk83]. They provide the logical counterpart to delegation, since with atomic actions one wishes to further restrict interleaving of activities by ensuring mutual exclusion for a series of statements rather than for a single *call/return* sequence. Atomic actions are useful for ensuring that an object not be “touched” by another activity while a series of requests are made to it. Atomic actions, when combined with checkpointing, also provide a facility for reliably “aborting” transactions that go wrong because of a detected deadlock or some unsatisfied requirement discovered during execution.

Atomic actions can be specified in Hybrid using the atomic statement. This defines a scope within which all completed operations cause their objects (and hence the domains containing them) to enter a *ready* state. This is like the *blocked* state, except that the domain waits for a commit or abort message (or any further calls related to the transaction).

```
var n : partRange ;
var part : array [partRange] of oid of graphicObject ;

n := partRange.first ;
atomic {
        coloop {
                activity {
                        delegate ( part[n].displaySelf ) ;
                }
                if ( n <? partRange.last ) { n += 1 ; }
                else { break ; }
        }
}
```

Example 7: Atomic actions

In the example, a complex graphic object displays its parts within a single atomic action to guarantee that the parts will all be synchronized. Assuming that the parts are independent objects, it would otherwise be possible for some parts to be moved by other activities before all of them have been displayed. The atomic action prevents this from happening.

Notice that it is possible to use concurrent subactivities within atomic actions. Delegation within atomic actions only makes sense within this context. To otherwise delegate a call enabling switching to other activities would conflict with the atomicity requirement.

8. Implementation.

In our prototype implementation of Hybrid we are mapping each environment of objects with its own object manager to a single Unix[1] process. Each object manager maintains a workspace containing the persistent data of each of its objects. The workspace may be atomically written to disk.

Hybrid type definitions are translated into intermediate code which is stored in the workspace and interpreted at run-time. The object managers mimic some of the functions of an operating system by time-sharing between active domains. Scheduling of domains is done on the basis of message-passing operations executed. Message-passing between objects in different environments is accomplished by object managers communicating through Unix “sockets”. Parallelism is thus achieved by having object managers execute on different machines.

One may speculate as to the requirements for an architecture better suited to supporting active objects. Such a machine would probably be a multiprocessor, with each processor running an object manager and a medium-sized collection of domains. Message-passing and process-switching would need to be fairly cheap since heavy use is expected to be made of both. It must be possible for processes to have multiple stacks to easily support delegation. (Recall that a domain requires a stack for each pending activity with a delegated call.)

Active domains should either reside in main memory only, with a method for “swapping” them out to disk, or object managers should manage their workspaces as huge persistent virtual memories with a paging algorithm that atomically updates *versions* of the stable workspace. In either case, it is important that “dormant” domains, that is, domains that are idle for a long time, be swapped out, and thus impose no run-time overhead.

9. Concluding Remarks.

We have presented a model for active objects in which concurrently executing objects communicate using an extended remote-procedure call protocol, and we have shown how the model can be used consistently in an object-oriented programming language to express solutions to a number of important concurrent problems. Specifically, we have demonstrated how to capture mutual exclusion, pipelines, triggering, postactions, “administration,” constraints, concurrent subactivities and atomic actions.

1. Unix is a trademark of AT&T Bell Laboratories

A more detailed description of the Hybrid language is given in [Nier87a]. A formal specification for our activity model is provided in [Nier87b]. In the same paper we examine the issue of deadlock and we suggest ways in which it can be handled.

We have not addressed the issue of signaling or "express messages" as they are called in ABCL/1 [Yone86]. We are still investigating ways in which the "interruption" of an active object can be defined in a way that is consistent with data-hiding and object-independence. The idea of "high-priority" operations is promising, though it is not clear how that may be used to affect the control of the interrupted activity.

We are experimenting with techniques for animating objects [Fium87]. We expect that the triggering and constraint mechanisms provided by Hybrid will be useful in binding animated objects to the active objects they represent. The separation of specification and realization given by the object-oriented approach means that we shall be able to respecify our animated objects in Hybrid while retaining parts of their existing implementation in C.

We are also carrying out research into useful objects for encapsulating and managing knowledge [Tsic87]. The objects, or "knos," are highly independent, active entities that may move between object environments. Knos may be complex, consisting of many concurrent parts (i.e., domains). Knos may react to their environment based on the rules they contain. Our previous experiments with knos have been in terms of object-oriented Lisp, using the flavors package [Wein81, Moon86]. We expect that the activities model that Hybrid provides will be useful for structuring knos, and that the triggering and delegation mechanism will be invaluable for managing the relationships between kno parts.

Finally, we have been carrying out work on the modeling of cooperating active objects to better understand complex activities or *tasks*, and to formalize the properties of concurrent object-oriented systems [Hogg87].

10. 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

[Andr83] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," ACM Computing Surveys, vol. 15, no. 1, pp. 3-43, March 1983.

[Blac86] A. Black, N. Hutchinson, E. Jul and H. Levy, "Object Structure in the Emerald System," ACM SIGPLAN Notices, vol. 21, no. 11, pp. 78-86, Nov 1986.

[Born81] A. Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," ACM TOPLAS, vol. 3, no. 4, pp. 353-387, Oct 1981.

[Byrd82] R.J. Byrd, S.E. Smith and P. de Jong, "An Actor-Based Programming System," Proceedings ACM SIGOA, SIGOA Newsletter, vol. 3, no. 12, pp. 67-78, Philadelphia, June 1982.

[Cox86] B.J. Cox, *Object Oriented Programming – An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

[Dijk75] E.W. Dijkstra, “Guarded commands, nondeterminacy, and formal derivation of programs,” CACM, vol. 18, no. 8, pp. 453-457, Aug 1975.

[Fium87] E. Fiume, D.C. Tsichritzis and L. Dami, “A Temporal Scripting Language for Object-Oriented Animation,” to appear, Eurographics ’87, Amsterdam, The Netherlands, 1987.

[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.

[Gold83] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[Hewi77] C. Hewitt, “Viewing Control Structures as Patterns of Passing Messages,” Artificial Intelligence, vol. 8, no. 3, pp. 323-364, June 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.

[Hogg87] J. Hogg, “Modelling Coordination Among Objects,” in *Objects and Things*, ed. D.C. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, March 1987.

[John86] R.E. Johnson, “Type-Checking Smalltalk,” ACM SIGPLAN Notices, vol. 21, no. 11, pp. 315-321, Nov 1986.

[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.

[Moon86] D.A. Moon, “Object-Oriented Programming with Flavors,” ACM SIGPLAN Notices, vol. 21, no. 11, pp. 1-8, Nov 1986.

[Nier87a] O.M. Nierstrasz, “Hybrid – A Language for Programming with Active Objects,” in *Objects and Things*, ed. D.C. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, March 1987.

[Nier87b] O.M. Nierstrasz, “Triggering Active Objects,” in *Objects and Things*, ed. D.C. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, March 1987.

[Scha86] C. Schaffert, T. Cooper, B. Bullis, M. Killian and C. Wilpolt, “An Introduction to Trellis/Owl,” ACM SIGPLAN Notices, vol. 21, no. 11, pp. 9-16, Nov 1986.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[Ther83] D.G. Therault, “Issues in the Design and Implementation of Act2,” M.Sc. thesis, TR \#728, MIT AI Lab, June 1983.

[Toko86] M. Tokoro and Y. Ishikawa, “Concurrent Programming in Orient84/K: An Object-Oriented Knowledge Representation Language,” ACM SIGPLAN Notices, vol. 21, no. 10, pp. 39-48, Oct 1986.

[Tsic87] D.C. Tsichritzis, E. Fiume, S. Gibbs and O.M. Nierstrasz, “KNOs: KNowledge Acquisition, Dissemination and Manipulation Objects,” ACM TOOIS, vol. 5, no. 1, pp. 96-112, Jan 1987.

[Wein81] D. Weinreb and D. Moon, *The Lisp Machine Manual*, Symbolics Inc., 1981.

[Wirt83] N. Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin, 1983.

[Yoko86] Y. Yokote and Mario Tokoro, “The Design and Implementation of ConcurrentSmalltalk,” ACM SIGPLAN Notices, vol. 21, no. 11, pp. 331-340, Nov 1986.

[Yone86] A. Yonezawa, J-P Briot and E. Shibayama, “Object-Oriented Concurrent Programming in ABCL/1,” ACM SIGPLAN Notices, vol. 21, no. 11, pp. 258-268, Nov 1986.