Concurrency Issues in Object-Oriented Programming Languages

Papathomas, Michael

Concurrency Issues in Object-Oriented Programming Languages

M. Papathomas

Abstract

The integration of concurrent and object-oriented programming, although promising, presents problems that have not yet been fully explored. In this paper we attempt to identify issues in the design of concurrent object-oriented languages that must be addressed to achieve a satisfactory integration of concurrency in the object-oriented framework. We consider the approaches followed by object-oriented languages for supporting concurrency and identify six categories of concurrent object-oriented languages. Then, we review several concurrent object-oriented languages and examine the interaction of their concurrency features with their object-oriented features and with object-oriented software construction.

1 Introduction

Object-oriented programming and object-oriented programming languages (OOPLs) are becoming increasingly popular for the construction of computer software. OOPLs integrate a host of techniques that have proven useful for the development and maintenance of software and that promote software reusability [GR83,Cox86,Mey88]. It is also expected that software development support systems will be more successful at developing applications by configuring pre-packaged software, if they use object-oriented techniques and are based on an object-oriented programming language [TN88].

Advances in computer hardware and the more widespread and demanding uses of computers in various areas has increased the number of applications involving concurrent programming. Applications that have to manage the interaction of multiple cooperating users, distributed systems, exploit parallel hardware for increased performance or that make use of multiple windows supported by modern workstations, are more easily written in programming languages especially designed to support concurrency.

Although the integration of object-oriented and concurrent programming is promising for the development of software for such applications, the design of programming languages that keep up with this promise is a difficult task. The concurrent features of a language may interfere with its object-oriented features making them hard to integrate in a single language or cause many of their benefits to be lost. For instance, encapsulation in sequential object-oriented programming languages protects the internal state of objects from arbitrary manipulation and ensures its consistency. If concurrent execution is introduced in a language independently of objects it will compromise encapsulation,

since concurrent execution of the operations of objects may violate the consistency of their internal state.

The approach taken for concurrency should also be carefully considered so that it is compatible with the principles underlying object-oriented software development. A bad choice concerning the concurrency features could cause objects to be designed in a way that only fits the concurrency requirements of a particular application Therefore it would be difficult to reuse these objects in different applications.

In this paper we will examine the interaction of the concurrency features and more general the approaches for concurrency taken by Concurrent Object-Oriented Programming Languages (COOPLs) with object-oriented features such as encapsulation, data abstraction and inheritance, and object-oriented software construction.

The interaction of several object-oriented language features has been examined in [Weg87]. Consistency of a set of features was defined as the possibility for the features in the set to coexist in a language. A set of features were defined as orthogonal if for every subset of the set, there is a language possessing the features in the subset and no features in the complementary subset. Based on this definition concurrency was found to be orthogonal with other object-oriented features. As for consistency, it was argued that the existence of actor languages is an indication that object-oriented programming is not inconsistent with concurrency, but there is a potential conflict between the sharing required by inheritance and the autonomy of objects in COOPLs.

For the purposes of our discussion we will take a different approach for examining the relationship between concurrency and object-oriented features. This will allow us to obtain more insight concerning the nature of their interference and get a finer classification of COOPLs.

Our approach for determining consistency of object-oriented features with concurrency is not based only on the possibility for the features to coexist in a language; we also examine whether this coexistence diminishes the power of a feature with respect to its applicability and its intended purposes. We believe that clearly identifying the purpose of features results in better designed languages and makes the interference of different features more apparent.

As an example for clarifying our approach we may consider the interference that occurs between class inheritance and encapsulation when subclasses are allowed to access freely the instance variables of the parent class [Sny86]. In this case we may say that support for inheritance diminishes the degree of encapsulation that was achieved without inheritance. In a language that supports inheritance in this way, it is more difficult to modify the internal representation of a parent class without affecting its subclasses, since their internal representation may depend on the one of their parent class.

In the next section we discuss the general approaches followed by COOPLs for integrating concurrency in the object-oriented framework. We compare the role that objects have with respect to concurrency. Based on this we obtain a first classification for COOPLs and briefly discuss their advantages and disadvantages.

In Section 3 we survey several COOPLs belonging in each one of the categories

identified in section 2. The examination of COOPLs in section 3 reveals that a large variety of concurrency features is used by languages in each of the categories of COOPLs identified in section 2.

In section 4, we compare the concurrency features of the languages examined in section 3. This comparison concentrates on whether the concurrency features promote or hinder the development of object-oriented applications.

Section 5 discusses the need for data abstraction mechanisms in COOPLs, the limitations of data abstraction mechanisms supported in sequential object-oriented languages for COOPLs, and the possibility to extend these mechanisms so that they may capture more information about the time-dependent behavior of objects.

In Section 6, we examine the interference of the concurrency features of COOPLs with class inheritance. This reveals that the concurrency features of some languages are incompatible with class inheritance. In other languages class inheritance is supported but, depending on the concurrency features, it may be hard to use.

Finally, we present our conclusions and briefly discuss other important issues concerning the integration of concurrency and object-oriented programming which are not otherwise addressed in this paper.

2 Approaches in COOPLs

The approaches followed by COOPLs vary considerably with respect to what objects stand for. Objects may be considered as processes, shared passive abstract data types, or as encapsulations of multiple processes and data.

In some languages objects do not have any predefined properties concerning concurrency. Objects are in general similar to those in sequential OOPLs. The concurrent features of the language may be used to implement objects that have some properties concerning concurrent execution. For instance, it is possible to implement objects that protect their internal state by synchronizing concurrent invocations of their operations, as it is also possible, to have objects whose internal state is not protected from concurrent invocations of their operations. The language makes no distinction between these different kinds of objects. It is the responsibility of the programmers to design applications in such a way that operations of "unprotected" objects, will not be invoked concurrently.

On the other hand, other languages take the approach that all objects have some predefined properties concerning concurrency. For instance, in some languages objects are processes that communicate by exchanging messages, while in others objects have the property that execution of their operations is serialized.

Based on these two approaches we identify two categories of COOPLs. We will call *orthogonal* the category of languages where objects are unrelated to concurrency, and *non-orthogonal* the category of languages where objects have some predefined properties concerning concurrency.

The concept of object in languages belonging to the orthogonal category, is indepen-

dent of concurrency. Programmers should take care to implement objects in such a way that they may be used in concurrent applications, it is not possible to know whether an object's operations may be invoked concurrently without knowing how it is implemented.

In languages belonging to the non-orthogonal category, objects are associated some properties concerning concurrency. The property of objects that is common to all languages in this category, is that the internal state of objects is protected from concurrent execution of their operations. This property alleviates the problems that may occur with languages belonging to the orthogonal category if "unprotected" objects are used in a concurrent environment. This is accomplished in two ways: either the internal state of all objects is "protected" by default or there are objects of "protected" and "unprotected" kinds. In the latter case, the language distinguishes between these kinds of objects and disallows the use of "unprotected" objects in a context where their operations could be invoked concurrently.

According to whether or not a language supports objects of different kinds, we will further subdivide the category of non-orthogonal languages into the *non-uniform* and *uniform* categories. Languages in the uniform category support only one kind of object whereas languages in the non-uniform category split the object world into two kinds of objects: those that serialize the execution of their operations and those that do not.

Concurrent execution may be expressed by explicitly creating new threads of control, independently of objects, that communicate and synchronize by invoking the operations of shared objects. Another approach is to consider objects as active entities and express concurrent execution and synchronization by the creation of objects and their interaction. We use these two approaches to subdivide the uniform category into the categories *integrated* and *non-integrated*. For languages in the integrated category concurrent execution is expressed by interaction of objects whereas in the non-integrated category another concept like a process or activity is used for expressing concurrent execution.

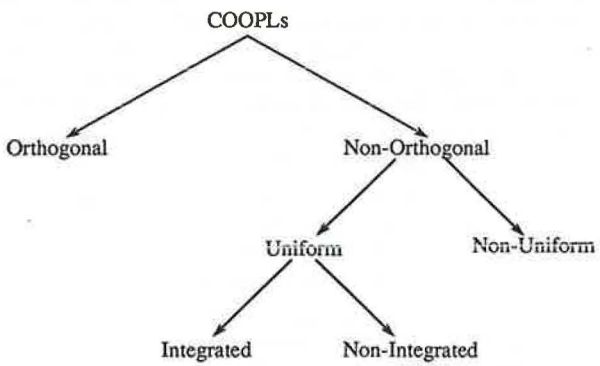

Figure 1:

The categories we have defined so far by examining general approaches taken by COOPLs is shown in figure 1.

From a reusability point of view languages in the orthogonal category have the disadvantage that in order to reuse objects one has to be aware whether their operations may be invoked concurrently. In order to function correctly objects depend on the environment in which they are used. When designing new objects special attention has to be paid for making them reusable for concurrent applications even if they were originally needed for a sequential application and are inherently sequential.

The non-orthogonal class has the advantage of preventing the problems that could occur by using "unprotected" objects in concurrent environment. Objects of sequential nature may be implemented in much the same way as in sequential languages. The mutually exclusive execution of the object's operations is handled automatically by the language.

The distinction between different kinds of objects characterizing the non-uniform category presents some disadvantages compared to the uniform one. The programmer has to decide in advance if a certain object should be of the "protected" or "unprotected" kind. Type hierarchies of "protected" and "unprotected" objects are typically kept disjoint, therefore objects having similar behavior may have to be defined twice introducing a certain redundancy in the class hierarchy. This approach may, on the other hand, have some performance gains. Unprotected objects may be implemented more efficiently, since invocations of their operations are not synchronized

The integrated approach has the advantage that concurrent applications are structured in terms of objects which are the units of concurrent execution. The communication and synchronization of objects is expressed at the object interface which is clearly defined. In contrast to this approach, the non-integrated approach tends to organize applications in terms of processes that call shared passive objects for their communication and synchronization. This obscures the points of interaction of processes with their environment and makes them harder to reuse.

3 A Closer Look at some Languages

3.1 Languages in the Orthogonal Category

Smalltalk-80

Smalltalk-80 [GR83] supports concurrent programming by processes that communicate through shared objects. The creation of processes is accomplished by sending a fork message to a block context and semaphores are provided for their synchronization and mutual exclusion.

Semaphores can be used in two ways. Either the calling process uses semaphores before invoking the operations of shared objects or the execution of an object's operations is synchronized by using semaphores in its implementation.

The solution of using semaphores before calling a shared object has the known disad-

vantages of unstructured use of semaphores [AS83]. Furthermore, for the development of concurrent applications this approach gives more importance to processes than objects. Finally, the implementation of objects has to be known in order to make sure that processes are properly synchronized and that no problems will occur because of concurrent accesses to shared objects.

The second use of semaphores presents other disadvantages. Not all objects will use semaphores to protect their internal state from concurrent execution of their operations. Therefore, when reusing an object class one must find out if its implementation is such that its instances may be shared by concurrent processes. Another problem with this approach is that it may interfere with inheritance and with the use of the pseudo variables self and super. If a semaphore is used in methods for ensuring mutual exclusion, a call to any of these methods from within the object by using self would lead to a deadlock. When inheritance is used for defining a new class, whose instances could be shared by concurrent processes, one should make sure that concurrent invocation of inherited methods will not cause problems. This may be solved either by examining and modifying all inherited classes or by overriding all inherited methods. The overridden methods would just invoke the inherited method (using super) within a critical section.

Emerald

Emerald [BHJ*87] is an object-oriented language for programming distributed applications. An object may be associated with a process that starts executing after the object has been created and initialized. The operations of objects may be invoked concurrently by their own process, if they have one, and by those of other objects. Synchronization of processes and mutual exclusion is accomplished by using monitors [Hoa74]. A number of monitors may be used in the implementation of objects and their operations may be implemented as monitor procedures.. This approach is more flexible than viewing the whole object as a single monitor since it allows operations that are not implemented as monitor procedures or that are in different monitors to execute in parallel. For instance a single object could be used for encapsulating a database that may be accessed concurrently by readers but needs exclusive access by writers. To solve the same problem if objects were monitors would require a monitor for gaining the right to access the database in read or write mode, an object encapsulating the database and finally a third object encapsulating the monitor and the database for ensuring that users of the database do not bypass the locking protocol by calling directly the database. Other problems concerning the use of monitors are nested monitors calls [AS83,Lis77,Par78]. Nested monitor calls may be avoided by using monitors only when mutual exclusion is necessary for protecting the object's state, but this means that the use of monitors within objects should be done in a way that fits the design of a particular application and not the mutual exclusion requirements of individual objects. Such an approach has the disadvantage that objects may not be easily reused across applications.

Trellis/Owl

Trellis/Owl [MK87] supports concurrent execution by explicit creation of concurrent threads called *activities* for the execution of an operation. Activities communicate by invoking the operations of shared objects. Objects of the type lock provide support for

mutual exclusion and objects of the type wait queue may be used for synchronization by waiting and signaling. Objects of these types may be defined as instance variables of an object and may be used in the implementation of the object's operations. For mutual exclusion code blocks may be associated with locks. Combinations of lock blocks and wait queues may be used for implementing objects similar to monitors [Hoa74], yet they allow more flexibility than monitors, since some operations may be allowed to execute concurrently.

There is also support for direct synchronization of activities. An activity may wait until some other activity or set of activities terminate.

```
type_module Shared_char_Buffer

component me.mutex      : Lock;
component me.nonempty   : Wait_Queue;
component me.buffer     : Char_Queue;

operation create (Mytype) returns (Mytype)
      is allocate
      begin
            me.mutex    := create(Lock);
            me.nonemoty := create(Wait_Queue);
            me.buffer   := create(Char_queue);
      end;

operation insert(me, x: Char)
      is begin
            lock me.mutex do
                  insert(me.buffer,x);
                  wakeup(me.nonempty);
            end lock;
      end;

operation remove (me) returns (Char)
      is begin
            lock me.mutex do
                  if( empty(me.buffer)) then
                        wait(me.nonempty)
                  end if;
                  return remove (me.buffer);
            end lock;
      end;
end type_module;
```

Figure 2:

The use of locks and wait queues is illustrated in figure 2 that shows the implementation of a character buffer of unbounded size that may be shared by concurrently executing activities.

Guide

In Guide[DKM*89] threads, called activities, communicate by calling the operations of shared passive objects. The synchronization mechanism consists of associating an *ac-*

tivation condition with the object's operations which must be true before the execution of an operation may take place. Activation conditions are boolean expressions that may refer to arguments of the invoked operation, instance variables and a number of *synchronization counters* and operation names. Synchronization counters are associated with an operation and record automatically the number of started, completed, pending and ongoing computations of an operation. For instance *started(op)* is the number of initiated execution of operation *op*, *completed(op)* is the number of terminated executions of *op* and *current(op)* is equivalent to *started(op) - completed(op)*. Some keywords are used as syntactic sugar for commonly used activation conditions. For instance the keyword EXCLUSIVE may be used in the activation condition of an operation for expressing mutual exclusion. The same thing could also be done by having an activation condition stating that the sum of *current(op)* over all operations *op* of the object must be equal to zero. The keyword NOT followed by the name of an operation may be used for specifying that there are not any ongoing executions of the specified operation.

```
TYPE boundedBuffer IS
      METHOD Put(IN i: Item);
      METHOD Get(OUT i: Item);
END boundedBuffer

CLASS FixedSizeBuffer IS
IMPLEMENTS boundedBuiffer;

CONST size = some constant
buffer:     ARRAY[0..size-1] of Item;
nbr , first,  last: Integer = 0, 0, 0;

METHOD Put( IN i: ITEM );
BEGIN
      ...code for put...
END Put;

METHOD Get( OUT i: Item );
BEGIN
      ...code for get...
END Get;

CONTROL
      Put: NOT Get AND NOT Put AND nbr < size;
      Get: NOT Get AND NOT Put AND nbr > 0
end FixedSizeBuffer
```

Figure 3:

The use of activation conditions is illustrated in figure 3 that shows the implementation of a bounded buffer in Guide.

The CONTROL clause specifies the activation conditions for the operations `Get` and `Put`. The term " `NOT Get AND Put`" specifies mutual exclusion of both operations. This term could actually be replaced by the keyword EXCLUSIVE for obtaining the same effect. The term "`nbr < size` " in the activation condition for `Put` prevents the exe-

cution of Put when the buffer is full. Similarly `nbr > 0` prevents the execution of `Get` when the buffer is empty. The code of the operations `Put` and `Get` does not include any synchronization primitives.

As activation conditions are specified separately from the code of the methods and the methods do not contain any synchronization primitives, this mechanism seems promising for inheritance. The code of the methods could be inherited separately from the activation conditions and in some cases it could be possible to inherit the activation conditions for methods. In fact, as it will be shown in section 6, this mechanism interferes with inheritance mainly because of the way it is implemented.

ConcurrentSmalltalk

ConcurrentSmalltalk [YT87a] is an extension of Smalltalk-80. The aims in the design of this language were to provide a better integration of concurrent programming with objects than it is the case with processes and semaphores in Smalltalk-80.

The concurrent features introduced are: asynchronous method invocation, allowing methods execution to proceed after returning a result, Cbox objects for synchronization and an *atomic* object class.

Asynchronous method invocation is specified by terminating the method invocation expression by the symbol "&". An asynchronous method invocation expression returns a newly created CBox object. The caller may bind the CBox object to a variable and use it later for receiving the result of an asynchronous method invocation. The result associated with the CBox object is obtained by invoking a receive method defined on CBox objects. If the result is not available when receive is invoked the invoker is suspended. CBox objects are used for synchronizing the caller of an asynchronous method invocation with the called object and as a private communication channel for getting the reply. Without CBox objects, it would be difficult for the sender of an asynchronous message to tell apart the reply from the other messages sent to it.

The atomic object class is used to define subclasses whose instances have the property that execution of their methods is mutually exclusive. This is needed for preventing problems that may be caused by concurrent execution of an object's methods.

A problem with the concurrency features of ConcurrentSmalltalk is that they provide limited support for condition synchronization. There are no means for selective acceptance of messages and no way to suspend the execution of method without resorting to Smalltalk-80 semaphores. We will further discuss this in section 4.3.

SR

SR (Synchronizing Resources) [AOC88] is a programming language for distributed applications. Although it is not advertised as an OOPL it supports several object-oriented features. Programs are structured in terms of resources. A resource encapsulates variables and processes. Resource specifications declare operations that are used for interacting with a resource. A resource may be implemented as an abstract data type, by specifying a procedure for each operation, or by one or more processes. A resource implementing a queue of integers as an abstract data type is shown in figure 4.

```
resource Queue
      op insert(item: int)
      op remove() returns item: int

body Queue(size: int)
      var store[0..size-1] : int
      var front := 0, rear := 0, count :=0

      proc insert(item)
            if count < size -> store[rear]:=item; rear:=(rear+1)%size;
                               count++;
            [] else ->  #take action appropriate for overflow
            fi
      end

      proc remove() returns item

            ... implementation of remove...
      end
end Queue
```

Figure 4:

Resources implemented as processes may selectively service operation invocations by using *in* statements. This is similar to Ada's select/accept [ANSI83] but more powerful since it allows guards to depend on the arguments of the invocation.

Resources are typed and may be created dynamically. Inheritance is supported by an *extend* construct which is used to refine a resource specification and provide different implementations of the resource's operations. It is possible to define *abstract* resources consisting only of a specification. Figure 5 shows the implementation of a resource that extends the queue given in figure 4 and implements it by a single process so that it can be used as a bounded buffer.

Capability variables to abstract resources may be used to refer to any concrete resource that extends an abstract resource by providing a *body*. It should be noted that this inheritance mechanism may not be used for inhering methods in the way it is done in languages like Smalltalk-80, but operates solely at the specification level.

A problem with resources is that no distinction is made between resources that are implemented as sequential data types whose operations should not be called concurrently, and resources that control activation of their operations which can may therefore be invoked concurrently. This may cause problems if a resource of the former kind is used in a concurrent environment.

3.2 The Non-Uniform Category

PAL

The programming language PAL, used in AVANCE system [BB88] supports two kinds

```
resource BoundedBuffer
      extend Queue

body BoundedBuffer(size: int)
      var store[0..size-1] : int
      var front := 0, rear := 0, count :=0

      process bb

          in insert(item) and count < size ->

                store[rear]:=item; rear:=(rear+1)%size; count++;

          [] remove() returns item and count>0 ->

                item := store[front]; front:= (front+1)%size; count--
          ni
      end
end BoundedBuffer
```

Figure 5:

of objects: *packets* and *datatype values*. Packets are associated with persistence, resiliency, synchronized access and independent existence. Datatype values are instances of abstract data types. Datatype values may only be used within a packet so that they are protected from concurrent access by the serialized execution of the operations of the enclosing packet. A new process in PAL may be created for the execution of any PAL expression. Execution of an expression in a new process creates an instance of a process data type that may be used in way similar to ConcurrentSmalltalk's Cbox objects for obtaining the computed result. Process communication takes place by invoking the operations of shared packets.

The distinction between packets and datatypes in PAL was made mainly because of the implementation overhead associated with the execution of packets. The implementation of datatype values has less overhead since execution of their operations does not have to synchronized. By restricting the use of datatype values within packets, datatype values are protected from concurrent access.

The disadvantage of this approach is that datatypes and packettypes form two separate type hierarchies that could contain objects with similar functionality: code that was written to operate on datatypes may not be used for packettypes and vice versa. The decision of implementing an object as a packettype or a datatype is left to the programmer, and it may depend on the intended use of an object in a particular application.

An Extension to Eiffel

Caromel [Car89] has proposed a concurrent extension to the programming language Eiffel [Mey88] that also takes the approach of separating the object world into two kinds of objects: *process* and *data objects*. Process objects are associated to a single thread of control that executes the code of their *live routine*. A process object accepts operation invocations explicitly by using a `Serve` within its live routine. The `Serve` primitive is

non-blocking. For accepting a request this primitive is called by specifying the name of the operation to be accepted. If there is a pending request for the specified operation, it is accepted and the specified operation is executed. Otherwise the `Serve` primitive has no effect and the execution of the live routine proceeds at the next statement. The operations of process objects are invoked asynchronously. An approach somewhat similar to ConcurrentSmalltalk's *CBoxes* and ABCL/1's *future type messages* is taken for cases where operations return a result. When such an operation is invoked an object representing the result is returned immediately. The caller is suspended when it attempts to use a result that is not yet available.

Data objects do not have a live routine and their operations are invoked as ordinary procedure calls. They may only be used within a process object so that their operations are never invoked concurrently.

Operations of objects, other than the live routine for process objects, do not contain any concurrent constructs. This is interesting for class inheritance. A class may inherit all the methods of another class and override just the live routine for changing the concurrent behavior of the object.

As both the `Serve` primitive and operation invocations are non blocking, it is rather difficult to synchronize process objects. The only ways are to use busy-waiting or to use the result returned by operations. For instance, if a process object wants to accept a message before proceeding with the execution of its live routine, the serve primitive specifying the operation has to be included in a loop.

This weak form of synchronization may lead to complex solutions to synchronization problems, where unneeded return values are used merely for synchronization, and complicate the protocols for object interaction.

ACT++

ACT++ [KL88] is a object oriented language that extends C++ [Str86] with concurrency features derived from the actor model [Agh86]. Although we do not have a detailed description of this language it is worth presenting some of its features.

ACT++ supports two kind of objects: actors and passive objects. Actors are active objects that serialize concurrent invocation of their operations. Passive objects are ordinary C++ objects. They are constrained to be used only within actors so that their operations may not be invoked concurrently. Any C++ object may be an actor if its class is defined as a subclass of a predefined actor class.

Two primitives `reply` and `become` are described in [KL88] for concurrent execution. The reply primitive is used within an operation to return a reply to the caller. Statements following the reply statement in an operation are executed in parallel with the caller. The `become` primitive is inspired from the replacement behavior in the actor model. An actor object may use the `become` primitive to specify a behavior name that is used to indicate what message it is willing to accept next. Once the replacement behavior has been specified, processing of messages may start according to the new behavior and in parallel with the statements following the `become` primitive in the old behavior. The old and new behaviors may not share the actor's internal state so no interference may take place

because of the parallel execution of behaviors.

Behaviors are specified, in the definition of an actor class, in terms of the operations that they may process.

Figure 6 sketches the definition of an actor class that implements a bounded buffer of integers.

The behavior part defines three behavior names for bounded buffers. The `empty_buffer` behavior accepts and processes the operation `put`. The behavior `full_buffer` accepts only `get` operations and the behavior `partial` accepts both `get` and `put` operation invocations.

Operation `get` illustrates the use of the primitives `reply` and `become`. The next item in the buffer is immediately returned to the caller by the `reply` statement, then the caller and the buffer proceed in parallel. The buffer examines its internal state and chooses its replacement behavior. The replacement behavior is specified by using the `become` primitive with the associated behavior name. The idea of behavior abstraction is interesting because it extends the data abstraction mechanism of sequential languages by including some information on the temporal aspects of an objects behavior.

Another interesting aspect of behavior abstraction is that it may be combined with class inheritance. We will further discuss this point in section 6.

3.3 The Integrated Category

ABCL/1

In ABCL/1 [YSTH87] objects are active entities that encapsulate a single thread of control and communicate by message passing. An object may be in one of the modes: *dormant*, *active* or *waiting*. Objects are created and remain in the dormant mode until they receive and start processing a message. While they are processing a message objects are in the active mode. From the active mode they may go back to the dormant or enter the waiting mode. An object goes from the active mode to dormant mode, when it has finished processing a message and no further messages have arrived. An object goes from the active mode to the waiting mode when it has to wait for a message satisfying a particular constraint to be received before it may do anything else.

Object interactions may have the form of remote procedure calls called *now type message passing*, asynchronous message passing called *past type message passing*, and *future type message passing*. Future type message passing is similar to the CBox objects feature of ConcurrentSmalltalk [YT87a]. For future type messages a special future variable is specified for storing the result to be returned by the receiver. The sender may continue executing in parallel with the receiver and obtain the result when needed by using this special variable. If the result is not available yet, the sender is suspended. It is also possible for the sender to test if the result is available.

Objects are defined in terms of their state and their *script*. The state of an object contains the definitions of its permanent variables which can only be accessed from

```
class bounded_buffer: Actor
{
      in_array buff[MAX];
      in    in, out;

      behavior:
            empty_buffer = { put()}
            full_buffer = { get() }
            partial_buffer = { get(), put() }

      public:

      buffer()
      {
            initialisation of the buffer ...

            become empty_buffer;
      }

      void put(in item)
      {
            insert an item
            if the buffer is now full

                  become full_buffer;
            else
                  become partial_buffer;
      }

      in get()
      {
            return the next item ...

            reply buff[out++];

            if the buffer is now empty

                  become empty_buffer;
            else
                  become partial_buffer;
      }
}
```

Figure 6:

within the script. The script specifies the messages that are accepted by the object and the actions to be executed upon receipt of a message. The specification of an acceptable message consists of the specification of the mode in which the message is acceptable, a message pattern, an optional reply destination, and an optional constraint.

The reply destination may be saved and a reply may be sent latter using a past type message. This allows an object to decide when to process requests even if they were issued by now type messages. Even if a now type message has been accepted by an object further messages may be accepted and processed before replying to the first message.

```
[ object buffer
      ( state [ s = ( create-storage 3)] )
      ( script
            ( => [:put aProduct]
                  (if (full? s) then
                        (select
                              ( => [:get]
                                    !(fetch s)
                              )
                        )
                  )
                  ( store aProduct s )
                  !"done"      ; return "done" as an acknowledgment
            )

            ( => [:get]@R
                  (case (fetch s)
                        (is :empty
                              (select
                                    (=> [:put aProduct]
                                          ; sent to the consumer
                                        [R <= aProduct]
                                          ; confirmation to producer
                                        !"done"
                                    )
                              )
                        )
                        ( is aStoredProduct
                              [ R <= aStoredProduct ]
                        )
                  )
            )
      )
]
```

Figure 7:

Figure 7 shows a bounded buffer object that illustrates some of these features. In the dormant mode the buffer accepts messages that match the message patterns `:put aProduct` and `:get`. Accepting a `:put` message binds the variable `aProduct` to what follows `:put` in the message. When a put message is accepted if the buffer is full it goes into waiting mode by executing the *select* form. This allows the buffer object to wait

until a get message is received. When a get message arrives an item is removed from the store of the buffer and is sent to the object that sent the get message, then the item to be put in the buffer is inserted in the storage and an acknowledgment is sent to the sender of put.

If a get message is accepted when the object is in the dormant mode the reply destination for this message is bound to the variable R. If the buffer is empty the object goes in the waiting mode waiting for a put message. When a put message arrives the item contained in the put message is immediately sent to the reply destination of the get message, saved in variable R, and an acknowledgment is returned to the object that sent the put message.

The reply destination of a message is determined implicitly for now type messages and future type messages. In general for past type message there is no reply destination since no reply is expected, but it possible to explicitly specify a reply destination for past type messages. This is very useful when an object wants to forward a request sent to it, with a now type message, to another object.

Another interesting feature supported is *express mode* messages. A message sent in express mode has the effect of interrupting processing of a message that was sent in *ordinary mode*. After the express message has been processed the object may choose to resume its previous activity or aborting it. This feature is very useful when there is a need to interrupt an object while it is executing.

This feature could be simulated in languages that allow objects to peek at messages sent to them. Still, it would be painful to program it explicitly. Every now and then the code of ordinary messages should examine the message queue and take the appropriate action.

The usefulness of this feature is illustrated in [YSTH87] by an example of a team of problem solvers working on a problem in parallel. When the first of them finds a solution the others are interrupted by sending them a message in express mode.

Since arbitrary interleaving of processes that share variables may cause problems, an object explicitly specifies what messages may be received in express mode. Also, atomic blocks may be used for making sure that certain sensitive blocks of code are executed atomically i.e. they may not be interrupted by express mode messages. ABCL/1 does not support classes or types of objects, but it is possible to define generator objects that are used to create several objects from the same object description. This is illustrated in figure 8 which shows an object that is used to create bounded buffer objects.

```
[ object create-buffers
      ( script
            ( => :new
                  ![ object

                        ...state and script as in figure 7
                  ]
            )
      )
]
```

Figure 8:

POOL-T and POOL2

In POOL-T [Ame87b] objects are considered as processes that communicate and synchronize by extended rendez-vous like in Ada [ANSI83]. Each object encapsulates a single thread of control that is created and activated at object creation. The sequence of statements executed by this thread is specified in a body. Acceptance of requests is done explicitly by use of an `answer` statement within the body. The answer statement is used to accept messages and execute the requested method; it is the only means for synchronization. Methods may have a post-processing section that is executed after the method has returned. The synchronization mechanisms are similar to those used in Ada and present the same limitations as other languages combining synchronous communications with static process structure. These are discussed and illustrated with examples in Ada in [LHG86]. Asynchronous message passing was later introduced in POOL2, another language of the POOL family.

The approach that an object encapsulates exactly one process causes some difficulties for providing concurrent implementations of objects. The functionality of these objects has to be implemented by several objects. Objects that are using this functionality may have to be aware of several objects that are used in the implementation and often the protocols become more complex. These problems may be solved by using *units*. Units are similar to modules or Ada *packages*, they may be used for hiding complex interactions between objects that are needed for achieving the desired parallelism. An example in [Ame87a] of a parallel search in a symbol table illustrates this use of units.

Hybrid

Hybrid [Nie87] is based on a model that combines features from message passing between objects with threads that communicate by invoking the operations of shared objects.

The units of concurrency are called *domains*. A domain consists of a process that encapsulates a collection of related objects. The collection of objects corresponds to an independent "top-level" object and its sub-objects. Domains communicate by exchanging messages. Message passing operations are structured as remote procedure calls. A message may be a request for executing an operation of an object of the domain or a return message containing the results of a previous call. The calling domain is blocked

until it receives the corresponding return message. Call messages transfer control from one domain to another. The thread of control identified by a sequence of calls is called an *activity*. Every message is associated with exactly one activity. New activities are created by calling operations called reflexes that start a new thread of control.

Domains may be active, idle or blocked. A domain is active when it processes a call message associated with an activity. A domain is blocked when some object in the domain has sent a call message to an object in another domain and waits for the reply. A domain that is neither blocked nor active is idle. A blocked domain may accept messages associated with the activity that blocked it. An idle domain may accept messages associated with any activity. Messages that are not accepted by a domain are stored in a message queue associated with the domain.

An activity is active if a message associated with it is being processed by some domain. It is suspended if it is not active.

An activity is an abstraction that results from the structure imposed on domains and on message passing operations between domains. Hybrid constructs may be understood both in terms of activities or in terms of message passing operations between domains.

Delay queues allow objects to selectively respond to incoming messages. A delay queue may be associated with an operation by declaring the operation as using the delay queue. A delay queue may be closed or open. When it is closed messages concerning the associated operation are delayed until the queue is again open. Delay queues may be closed and opened by invoking the close and open operations defined on delay queues within the operations of the object. Figure 9 illustrates the use of delay queues for implementing a bounded buffer in Hybrid. The operations put and get are associated with the delay queues `putDelay` and `getDelay`. The two delay queues are opened and closed by the operations `get` and `put` in such a way that no invocations of `put` or `get` are accepted when the buffer is respectively full or empty.

This mechanism is not very powerful for handling condition synchronization. The decision of delaying a call can not be based on the arguments of the invocation. The condition that must be satisfied for accepting a call is only tested when the delay queue is opened and there is no guarantee that the call waiting for the condition will be accepted next. The programmer must explicitly close the delay queue when because of the acceptance of other calls the condition becomes false. It is difficult to express preference of certain invocations over others. For instance when programming a disk head scheduler we would like to accept messages concerning the cylinder that is closest to current position of the disk head. Solutions to these problems may actually be expressed in Hybrid by combining delay queues with the *delegated call* mechanism.

The delegated call mechanism allows a domain to switch its attention to another activity, while a call issued by the current activity is processed by another domain. The execution of the activity is resumed, in the domain that issued the delegated call, sometime after the call has returned depending on whether the domain is busy with other activities.

```
type boundedBuffer :
abstract{
      init : ->;
      put : string ->; uses delay;
      get : -> string; uses delay;
};

private {

      var putDelay, getDelay : delay;
      var buffer : strarray;
      ...

      init : ->;
      {
            ...
            putDelay.open();
            getDelay.close();
      }

      get : -> string; uses getDelay;
      {
            ...
            get an item.
            if the buffer is now empty

                  getDelay.close();

            putDelay.open();
            ...
      }

      put: string -> ; uses putDelay;
      {
            ...
            insert an item
            if the buffer is now full

                  putDelay.close();

            getDelay.open();
      }
}
```

Figure 9:

SINA

Object interaction in SINA [TA88] takes place by remote procedure calls. Messages sent to objects are stored in an interface queue of unspecified length. Each object has a system defined object manager that manages the interface queue and selects messages to be processed by the object. The object manager of an object O is denoted ↑O and it supports the operations `hold()` and `accept()`. These operations have the effect of setting the object's interface in the *hold* and *accept* state respectively. While the interface is in the hold state messages are not processed by the object, they are instead delayed in the interface queue until the object's interface returns to the accept state. In the accept state the object's interface alternates between the sub-states *blocked* and *free*. In the free state a message is removed from the interface queue, a process is created for processing the message and the interface moves to the blocked state. No further messages are processed while the interface is in the blocked state. The interface goes back to the free state when the process that caused the interface to go in the blocked state terminates. With these scheme messages are processed serially and only one process is executing in an object at a time. A `detach()` primitive offers the possibility to process messages concurrently. When a process invokes this primitive the interface of the object goes to the free state. So that another process may be created for processing the next message concurrently.

Objects in SINA may be methods or data objects. Data objects may be primitive objects such as integers or instances of user defined types. Methods are associated with a process description (i.e. the code of the method). Each time a message is sent to a method object, a process is created for executing the process description with the formal parameters bound to the objects contained in the message.

New object types are defined by grouping other data objects and methods together in a type definition. A type definition specifies which objects (usually methods) are visible to other objects which determines the type's interface. Objects contained within a type definition may invoke the `hold()` and `accept()` operations on object managers of other objects contained within the type. This allows to realize various forms of synchronization.

Although this model at first may appear extremely complex the resulting language is rather simple. Type definitions resemble to types or classes in other languages and method execution may be understood as in other languages. This is illustrated in figure 10 by an example of a bounded buffer type in SINA.

The `initial` statement is a process description for a process that is executed at the creation of an instance of this type. After this process has terminated the interface of the object is in the free state and messages may be processed. The execution of `↑get.hold()` by this process sets the interface of the method get to the hold state by invoking the operation `hold()` on its object manager. The interface of `get` is now in the hold state which means that messages sent to `get` will be queued until its interface goes in the accept state. The interface of `put` is in the accept state so messages sent to `put` may be processed.

When a message is sent to `put`, a process is created that executes the code of the method `put`. The code of this method appends an item to the buffer and if the buffer

```
type buffer interface is
begin
      method put( integer as item ) returns nil;
      method get() returns integer;
end;
type buffer local is
begin
      objects integer as itemcount, head, tail, buff[N];

      initial
      begin head := 0; tail := 0; itemcount := 0; ^get.hold();
      end

      methods
      put:
      begin
            itemcount := itemcount +1;
            if itemcount = N then  ^put.hold();
            buff[tail]:= item; tail := (tail+1) mod N;
            ^get.accept();
            return val;
      end;

      get: objects integer as val ;
      begin ... code for get  ... end;
end;
```

Figure 10:

becomes full it sets the interface of put at the hold state so that no more messages will be processed by put, then it sets the interface of get to the accept state by invoking the accept operation on its object manager. After the execution of the method terminates messages to get may be processed.

4 Concurrent Programming

The examination of COOPLs in the previous section reveals that, independently of the classification of section 2, a variety of approaches and notations are taken for concurrency.

In [AS83] a number of notations for concurrent programming have been surveyed and concurrent programming languages have been classified in the three categories: *procedure-oriented, message-oriented* and *operation-oriented.* The characteristic of *procedure-oriented* languages is that processes communicate through shared variables, synchronization constructs, as for instance monitors, are used for synchronization and mutual exclusion. The last two categories are based on message passing for communication and synchronization. The difference between these two categories is identified by the explicit use of primitives for sending and receiving messages in *message-oriented* languages, and structured implicit message passing, like remote procedure calls or rendez-vous, in *operation-oriented* languages.

Languages in each of these categories are roughly equivalent in expressive power. Languages in the procedure-oriented category are not suitable for distributed systems because of the cost of simulating shared memory when there is none. Operation-oriented and message-oriented languages may be implemented both on shared memory systems and distributed systems. Operation-oriented languages are better suited for programming client/server forms of process interaction while message-oriented are best suited for pipelined computations.

Andrews and Schneider conclude that although the basic problems of concurrent programming are understood and mechanisms for solving these problems have been developed, the appropriate combination of language constructs deserves further examination. The examination of the concurrency constructs that we undertake in this section goes along this direction, although we are more specifically interested in the combination of constructs that provide better support for object-oriented programming.

The approach taken in their survey concentrates on how concurrent execution is expressed and how concurrent threads communicate and synchronize. In order to compare the concurrency features of the languages that we examined we will take a rather different view. We will center our attention on the concurrent properties of objects since object-oriented systems should be structured in terms of objects rather than concurrent threads of control. Instead of considering the synchronization of threads that call the operations of objects, we will consider the means by which objects may handle concurrent requests made by their environment. To make this more precise, instead of taking the view that processes in say Trellis/Owl or Smalltalk-80 communicate by using shared memory (i.e. the instance variables shared by concurrent invocations of an objects operations) which corresponds to the procedure-oriented model, we will consider that objects receive requests from their environment and for handling such requests a new process is created automatically when a request is received, which corresponds to the operation-oriented category.

The reason for taking this view is that it allows us to concentrate on the concurrent properties of objects and provides a common basis for comparing the approaches taken for concurrency by the languages examined in section 3.

A somewhat similar view of concurrency has been taken in [Weg87] and [LHG86]. They both have examined and compared concurrent programming languages with respect to the process structure of their modules. Liskov et al. have also considered the combination of the process structure of modules with primitives for module interaction. They have also formulated concurrency requirements for modules of distributed programs and examined in detail the program structures needed to satisfy these requirements in languages with static process structure and synchronous communication primitives.

4.1 Process Structure of Objects

The process structure of objects determines the number of processes that may be active within an object at the same time, and how they are created and synchronized.

Liskov et al. distinguish between *static* and *dynamic* process structure. A module

has a *static process structure* if the number of threads that may be executing within a module is fixed. The programmer is responsible for multiplexing requests among this fixed number of threads. A module has a *dynamic process structure* if the number of processes that may be executing within a module is variable. Processes are scheduled by the system and the programmer synchronizes access to shared variables. With this definition we could say that an Ada task [ANSI83] is a module with static process structure whereas an Ada package is a module having dynamic process structure.

Wegner uses the term process for a concurrently executing object in a concurrent object-based language. A thread is defined as what, more traditionally, would be called a sequential process from an operating systems point of view. A thread consists of a thread control block, a locus of control and an execution stack. Processes are classified with respect to the properties of their threads into the categories: *sequential, quasi-concurrent* and *concurrent*. Sequential processes have a single thread of control, quasi-concurrent processes have several threads of control but only one thread may be executing at a time. Execution of threads in quasi-concurrent processes is multiplexed in a way similar to coroutines. Concurrent processes have several threads that may be executing concurrently.

We will say that an object has a *dynamic process structure* when a variable number of processes may be executing concurrently within an object and they are created automatically by an operation invocation. With this definition all languages in the orthogonal category support objects with dynamic process structure. For languages like SINA and ACT++ which support objects with a variable number of processes, but where the processes are not created automatically by operation invocations, we will say that they support objects with concurrent processes or more simply *concurrent objects*.

We will use the terms *single-process object* and *quasi-concurrent object* in the place of *sequential process* and *quasi-concurrent process*, while we are going to use the term *concurrent object* for objects with concurrently executing threads that are not created automatically by operation invocations.

4.2 Concurrency Requirements for Objects

The main requirement expressed by Liskov et al. [LHG86] is that modules should be able to turn their attention to another activity if the currently executing activity has to be delayed.

They identify two different situations in which this is needed: *local delay* and *remote delay*. The first situation arises when a server may not process immediately a client's request, for instance, because of the temporary unavailability of some local resource. In this situation the server should be able to put aside the current request and turn its attention to other requests that could be processed immediately. Remote delay is encountered when a server, in order to process a client's request, invokes another module. The module called by the server may not be able to process the request immediately or it could take a lot of time. In the meanwhile the server could accept requests from other clients.

Another reason for dealing with remote delays is for avoiding unnecessary deadlocks. In a system where inter-module communication paths are organized hierarchically a local delay in a lower level module produces problems similar to the problem of nested monitor calls [Lis77]. Liskov et al. closely examine the solutions that may be expressed to these problems by languages that combine modules with static process structure with synchronous remote procedure calls and conclude that this combination does not provide sufficient expressive power for coping with these problems.

In the rest of this section we will compare the approaches followed in COOPLs with respect to the process structure of their objects and their primitives for communication and synchronization. We will examine the support provided for coping with local and remote delays, scheduling of requests, dealing with mutual exclusion problems and providing concurrent implementations of objects.

Languages in the orthogonal category do not impose any a priori restrictions on the process structure of objects. This has the advantage that it is flexible for expressing solutions to synchronization problems and it makes it easy to implement objects concurrently. The main disadvantage is that the programmer has to deal explicitly with mutual exclusion problems because of the dynamic process structure.

Languages in the non-orthogonal category are more restrictive with respect to the process structure of objects. The main objective of this approach is to make it easier to handle mutual exclusion problems. In some languages mutual exclusion problems are eliminated, in others one has to be concerned with mutual exclusion only when there is explicit interleaving of threads or when the creation of concurrent processes for executing the operations of objects is explicitly requested. On the other hand, depending on the restrictions imposed and the choice of communication and synchronization primitives it may become more difficult to deal with some concurrency problems.

4.3 Languages in the Orthogonal Category

In Emerald monitors may be used for coping with local delays by suspending threads in condition variables. Scheduling of requests may be done by using a combination of monitors and the process associated to an object. For example, the threads executing an object's operations could be suspended in a monitor and the process associated with the object could wake them up according to some scheduling algorithm. Concurrent implementations of objects are possible, since operations that are not implemented as monitor procedures may be executed concurrently. It may however be more difficult to deal with remote delays and there is a possibility of deadlocks because of nested monitor calls. It is possible to design objects so that nested monitor calls do not occur but this is contrary to encapsulation since objects have to be designed in a way that depends on the implementation of other objects in an application.

CBox objects and asynchronous operation invocations in ConcurrentSmalltalk provide a satisfactory solution for coping with remote delays. If an object does not want to be suspended when invoking the operation of another object it may invoke it asynchronously and use a CBox object for obtaining the result later. No support is provided

for coping with local delays, scheduling and selective acceptance of requests. To some extent this could be accomplished by using CBox objects and asynchronous calls but this would often lead to complex solutions. It could also be possible to use the Smalltalk-80 semaphores but the aims in the design of ConcurrentSmalltalk was to provide higher level synchronization mechanisms.

The limitations of the synchronization mechanisms of ConcurrentSmalltalk for coping with local delays is illustrated by an example of a bounded buffer presented in [YT87a]. In this example a bounded buffer has methods `deposit` and `remove` for depositing and removing items. With the synchronization mechanisms of ConcurrentSmalltalk there is no easy way for the buffer to suspend or avoid the execution of the `deposit` or `remove` method invoked by a producer or consumer when it is respectively full or empty. The solution that is given in [YT87a] solves that problem in the following way: when the buffer is full/empty invocation of the methods `deposit/remove` return a value that indicates this fact. The producer/consumer has to check the return value and suspend its activity if the buffer is full/empty. When the buffer is at a state that the activity of the consumer/producer may be resumed it invokes one of their methods for waking them up.

This solution has the disadvantages that the buffer requires that the producer and the consumer have a certain method fixed in advance for waking them up and that they are well behaved. Protocols for object interaction become more complex and objects are more difficult to understand and reuse.

The limitations of the synchronization mechanisms were identified and corrected in a latter version of ConcurrentSmalltalk [YT87b] by the introduction of the concept of a *secretary*. A secretary is an object that may be associated with any other object and handles receipt and execution of its methods. An object may request its secretary to suspend the execution of method until some condition is met. The resulting synchronization mechanism is similar to monitors [Hoa74].

The synchronization mechanisms of SR provide support for all of these problems. Explicit acceptance of messages with guards, that may depend on the state of a resource and arguments of the call, can be used to cope with local delays and scheduling. Remote delays may be handled by using asynchronous operation invocations, by using multiple processes for handling requests, or by creating new resources that handle requests to remote servers. Mutual exclusion may be handled by implementing a resource by a single process, or by synchronizing the execution of processes that share the state of resource through calls to other resources that are especially designed for this purpose.

Guide's activation conditions provide a satisfactory mechanism for dealing with local delays. Remote delays may also be handled since multiple processes are allowed to be active within an object. However it is difficult to control the mutual exclusion of these processes after they have completed their remote call. This synchronization mechanism provides limited support for scheduling requests.

4.4 The Non-Orthogonal Category

There are mainly three approaches with respect to the restrictions imposed on the process structure of objects in this category.

The most restrictive one views objects as sequential processes. In this case no problems may occur because of access to shared variables since only one process has access to an object's instance variables. It is also easier to prove the correctness of operations since they are executed sequentially. We will name this approach *single-process objects.*

A second approach is to permit multiple threads of control to execute within an object in a quasi-concurrent fashion with only one thread that is active at a time. The interleaving of threads occurs in well defined points and it is controlled explicitly. We will name this approach *quasi-concurrent objects.*

The most permissive approach permits multiple threads to be active at the same time within an object. Taking this approach may require that additional synchronization primitives have to be introduced for controlling the interference of threads that share the instance variables of an object. We will say that this approach supports *concurrent objects.* It should be noted that the difference between this approach and the dynamic process structure of objects in the orthogonal category is that in the orthogonal category threads are created automatically at receipt of a message whereas in this case the object controls explicitly when and for which messages new threads are created.

4.4.1 Single-Process Objects

From the languages that we have presented in section 3, POOL-T, POOL2, ABCL/1 and the extension to Eiffel proposed by Caromel follow this approach.

The synchronization mechanisms in the extension to Eiffel provide limited support for coping with local delays and scheduling. The `Serve` primitive may be used for coping with local delays by not accepting a request that would be delayed, but this decision may not be based on the arguments of the request. In some cases the arguments of the request are necessary for determining if a request should be delayed. Asynchronous operation invocations combined with data driven synchronization may be used for handling remote delays. This would be even better if the result returned by an asynchronous call could be passed to other process objects and if it would be possible to test if the result is ready. A problem with this proposal is that it is difficult to synchronize objects.

POOL-T provides support for local delays by explicit acceptance of messages but the decision for accepting a message can not be based on the values of the arguments supplied to a call. Remote delays are not easy to handle because once a request has been accepted by an object a reply must be returned to the caller before accepting another request. Solution to these problems may be expressed by creating a new objects with predetermined behavior for handling the requests.

ABCL/1 provides more flexible solutions to all these problems by supporting powerful guards and by allowing to explicitly program scheduling and multiplexing of messages by manipulating messages and reply destination as data. It may be argued that the resulting

programs lack structure and are hard to understand. It may also be argued that because of the existence of express mode messages ABCL/1 its objects should be considered quasi-concurrent. We have classified it under this category because we considered that express mode messages are more a mechanism for handling exceptional situations than for controlling the multiplexing of quasi-concurrent processes.

4.4.2 Quasi-Concurrent Objects

Hybrid is the only language from the languages examined in section 3 that belongs to the non-orthogonal category and supports quasi-concurrent objects. The delegated call mechanism of Hybrid provides a powerful means for coping with remote delays [LHG86]. This use of delegated calls is illustrated in [Nie87] and [KNP88] by programming an administrator [Gen81]. When the administrator accepts a call from a client object it selects a worker object, implemented as a separate domain, and issues a delegated call to it supplying information about the work to be done. The delegated call does not block the administrator object. It allows it to accept more calls from other clients while the job of the first client is processed by the worker. The two requests are executed in the administrator by quasi- concurrent processes which results to a clean program structure.

The delegated call mechanism may also be combined with delay queues for providing a better way for coping with local delays than it is possible with the sole use of delay queues. Yet the solution is complex and we believe that some more direct means should be provided for dealing with local delays.

The interaction of activities, domains and delegated calls causes some problems for controlling the interleaved execution of threads. With Hybrid's execution model interleaved execution of an object's operations may occur because of cyclic invocations and delegated calls. Cyclic invocations occur when an object calls another object's operation which calls back an operation of its caller. This is allowed for providing support for a form of recursive calls. In other languages with synchronous operation invocation, POOL for instance, this situation would cause a local deadlock.

4.4.3 Concurrent Objects

ACT++ and SINA are languages in the non-orthogonal category that support concurrent objects.

The support provided by ACT++ for dealing with local delays is limited because a decision for accepting a request may not be based on the actual parameters of an invocations. The multiple threads created by the `become` primitive may in some cases be used for coping with remote delays but not in all cases since after the `become` primitive has been executed the thread may not access the object's state.

SINA also provides limited support for local delays. In order to delay a request after it has been accepted the process must call an object whose interface is in the hold state. This may be accomplished easily since it is possible to access the object manager of objects defined within a type and set their interface in the hold state. This feature may

also be used for scheduling. The possibility to have multiple concurrent processes within an objects is satisfactory for dealing with remote delays and for implementing objects concurrently.

4.5 Discussion

The controlled execution of concurrent threads within an object is a net advantage of the non-orthogonal approach. The question of whether a language should support single-process, quasi-concurrent or concurrent objects is debatable. The advantage of the single-process approach as supported by POOL-T, POOL2, ABCL/1 and Caromel's concurrent extension to Eiffel, comes from the fact that objects are internally sequential, but this also has some drawbacks. For dealing with synchronization problems with the concurrent extension to Eiffel, the interfaces of objects may become more complex and objects should be used in a disciplined way. In POOL-T and POOL2 units may be used for providing simple interfaces and enforcing the protocols but the solution may require multiple objects and may be rather complex.

The sequential and quasi-concurrent approach also limit the potential for parallelism by not allowing concurrent execution to take place within an object. This problem may be solved by implementing the functionality of an object by a collection of objects that execute concurrently, but this has some disadvantages. There are cases where it may not be natural to decompose objects, by decomposing objects we may get a number of new classes that may not be useful for reusing on their own. If the number of such classes becomes important it may be more difficult to locate useful classes. The objects obtained by such a decomposition may need to cooperate more closely with each other than with other independent objects. In this case we should be able to provide different levels of interfaces. One level for independent objects and another for objects that are more intimately related. If an object is decomposed to a set of concurrently executing objects we may need other ways for controlling non interference and for coordinating them. This may be done either by using more complex protocols or by atomic actions that may involve sets of objects. These solutions are more complex than if objects could use multiple threads for the execution of requests from other objects.

We believe that using multiple processes for implementing objects may in some cases provide cleaner and more structured solutions. What seems more important than the problems of interference within a single object is the choice of concurrent programming features that provide satisfactory solutions for implementing objects in various ways, without affecting the way in which they are used and without having to depend on the implementation of other objects in a particular application.

5 Data Abstraction in COOPLs

Most of the languages that we have examined support data abstraction. The benefits of data abstraction have been recognized long ago [Par72,LZ74] and support for programming with abstract data types (ADTs) has been provided by several languages starting

with Clu [LSAS77] and Alphard [SWL77].

One of the benefits of data abstraction is the development of abstractions that more closely model physical or conceptual entities occurring in an application domain and promotes higher level programming. The effective realization of the abstract data type is hidden behind an interface. Operations defined at the interface capture the behavior of the entity modeled by the ADT and protect the internal state used to implement the ADT from inconsistent use. This separates the concerns of realizing the abstractions and programming by specifying interactions between the abstractions which model the entities of an application domain. The implementation of an ADT may be replaced, extended, without affecting applications making use of the ADT.

The modular decomposition that occurs by programs developing in terms of ADTs provides modular units that are more likely to be reused than modules resulting from an arbitrary decomposition of a systems functionality for managerial purposes.

5.1 Support in Programming Languages

Many of the benefits of data abstraction seem to be overstated when one examines what actually happens in programming languages. The information that is available at the interface consists of a set of operation names and arguments together with the types of arguments and return values in the case of typed languages. This information may in many cases be insufficient for reflecting the semantics of the data type. Several authors have shown this problem by taking as example an ADT representing a queue and a stack. Both of these ADT support `get` and `put` operations, yet their semantics are different.

This problem may be addressed by an appropriate choice of names for the types and their operations, by comments, by some formal specification of the behavior, or by examination of the implementation.

Relying on comments or other informal descriptions of the behavior of an ADT has the disadvantages that such informal description may be imprecise and incomplete. Some things that may seem trivial or irrelevant to the implementor may be essential for certain uses of the data type that where not anticipated by its implementor.

Formal techniques for specifying the behavior of ADTs, such as algebraic specifications solve some of the problems of informal descriptions. The disadvantages of these approaches is that they may be hard to understand for programmers and are event harder to write correctly. Also, it is hard to guarantee that implementations correctly implement the specifications and that changes are propagated to specifications the implementation is changed. Assertions included in the code like in Eiffel [Mey88] attempt to solve these problems.

If all of the previous approaches fail to provide enough information for reusing an ADT the remaining solution, assuming that the code is available, is to examine its implementation. This may be hard and time consuming. The implementations may be based on a number of other ADT. Then the problem of understanding the behavior of an ADT is replaced with a greater number of instances of the same problem. For under-

standing the implementation one may need to understand the behavior of the ADTs used in it. The properties of the ADT that are discovered by examining the implementation may not be properties of the ADT but of the particular implementation.

5.2 Data Abstraction for COOPLs

The presence of concurrency in COOPLs makes the problems examined above even harder. The behavior of objects includes a time dimension. This can not be captured by the names of operations and the types of their arguments. For example figure 11 shows what could be the interface specification of an object modeling a vending machine in the language Hybrid [Nie87].

```
Type VendingMachine:
abstract{
      selectItem: itemType -> ;
      insertAmount: amountType -> ;
      getItem: -> itemType;
      cancel: ->;
      getChange: -> amountType;
}
```

Figure 11:

The problem with such an interface specification is that it does not contain information as in what order the operations should be invoked. Should one first make a selection then insert the amount, or the other way around? Also what should happen if the operations where called concurrently?

Some of these problems could be solved by an informal descriptions of the behavior, but in COOPLs such descriptions are more likely to be imprecise and incomplete than for the sequential case.

Concerning formal specifications, there is no consensus for formal methods for specifying the externally visible behavior of concurrent systems. Furthermore such methods may have the same problems as algebraic specifications for ADTs. Determining properties of objects by examining the implementations presents the disadvantages mentioned above for ADTs, and concurrent systems are harder to understand.

For evaluating which abstraction mechanisms are adequate for COOPLs it is essential to identify what information should be captured and what should be hidden. Examining how objects relate to concurrency is important in order to characterize the relevant information. We may identify three main ways:

- An object's representation may need to be protected from concurrent invocation of its operation.

- The object is used to model a real world process or an entity with time varying behavior.
- The object is implemented in a concurrent fashion although the modeled entity may not be inherently concurrent.

The first and third points address issues that are more related with an object's representation. The first issue was discussed in section 2 and we have seen that there languages that ensure that the representation of objects is protected from concurrent execution. In these languages the object abstraction is powerful enough to hide this representation issue.

The second point addresses the problem of objects with time varying behavior. Such objects are able to decide whether and when they will respond to incoming messages. These decisions may be based on information concerning the past object's history, its state and the request. This behavior is characteristic of an abstract object and not of a particular representation. Knowledge of this behavior is essential for the development of applications involving collections of cooperating objects. Object interactions have to be properly synchronized on the basis of this information.

In most COOPLs the behavior of such objects is expressed by using the concurrency constructs of the language. The design of concurrent constructs of languages vary in their ability to convey information about the abstract behavior without detailed examination of the representation, and with respect to their expressive power.

Support for mechanisms similar to data abstraction in COOPLs is even more badly needed than in sequential languages but also is hard to provide. Ideally the data abstraction mechanism provided by COOPLs should separate the aspects of concurrent execution relevant to an object's implementation from the concurrent behavior of the abstraction. The complexity of the analysis and design of systems of concurrent objects would be reduced if it would be possible to suppress details concerning particular realizations. Replacing and extending the representation of objects could be done without requiring reexamination of the whole system. It would in principle be enough to ensure that the "new objects" have compatible behavior. For reaching such goals we would need a way for describing the externally observable behavior of objects, be able to prove and automatically check that the realization of objects satisfies the specifications.

This goal seem unreachable in the near future. It would be more reasonable to provide compromises by designing abstraction mechanisms that capture more information about the behavior of active objects than abstract data types. A parallel to this approach may be drawn with the way ADTs are used in sequential programming languages and the use of assertions instead of supporting algebraic specifications and automatic verification of programs.

5.3 Support for Data Abstraction in COOPLs

The data abstraction mechanisms supported by most of the languages examined in section 3 are not any different than in sequential languages. We may however say that the

languages in the non-orthogonal category extend the data abstraction mechanism of sequential languages since objects are associated with concurrent behavior.

The approach taken in ACT++ extends signatures with a behavior specification that provides some limited information about temporal aspects of the behavior of objects. It provides the information that the object may be in states where only a subset of operations defined at its interface are acceptable. This is rather limited because it does not specify how and when the object may change its state.

An early proposal for an abstraction mechanism capturing synchronization information for abstract data types was path expressions [CH74]. The idea in path expressions is that an expression in a regular language specifies synchronization constraints on the invocations of operations of an abstract data type.

```
Type VendingMachine:
path
 selectItem ( cancel , (insertAmount; getItem; getchange) )
end
abstract{
      selectItem: itemType -> ;
      insertAmount: amountType -> ;
      getItem: -> itemType;
      cancel: ->;
      getChange: -> amountType;
}
```

Figure 12:

Using the notation introduced in [CH74] the signature of the vending machine type of figure 11 is extended to the one shown in figure 12. The path construct describes the sequences in which the operation invocations are accepted by objects of the type `VendingMachine`. The operator ";" is used to indicate sequencing of operations while the operator "," indicates alternation. The path in figure 12 specifies that the vending machine first accepts a selection then it either accepts a `cancel` operation or the sequence of the operations `insertAmount`, `getItem` and `getChange`. This provides a lot more information about the behavior of instances of `VendingMachine` than did the signature in figure 11.

It should be noted that the behavior specified by a path expression is effectively observed by an implementation of an abstract data type since no other synchronization mechanisms may be used in the implementation of its operations.

The main problem of this synchronization mechanism is its expressive power. It is not powerful enough for specifying condition synchronization [AS83]. Synchronization decisions can not be based on arguments of the invoked operation. In some cases the previous history of the object as captured by path expression may not be enough to convey information about the object's state.

6 Inheritance

Several authors have mentioned the interference of inheritance as a code sharing mechanism with concurrency. America [Ame87a] discusses the difficulties for integrating inheritance in the language POOL-T and concludes that inheritance would be of little use in this language. Wegner examined the consistency of object-oriented features and mentions a potential inconsistency between the independence of objects emphasized by concurrency and the sharing of code required by inheritance [Weg87]. Decouchant et al. [DKM*89] discuss the interference of the synchronization mechanisms of Guide with inheritance. Kafura and Lee [KL88] examine more generally the problem of integrating inheritance in COOPLs. They compare and classify the approaches taken by various languages and propose their solution for the language ACT++. Briot and Yonezawa [BY87] discuss inheritance for COOPLs assuming no shared memory. They take the approach to support inheritance by delegation [Lie86] and show that the messages used for delegation heavily interfere with normal processing causing difficult synchronization problems.

In the following we will discuss the interference of class inheritance with concurrency based on the languages that we have examined in section 3. Then we will attempt to draw more general conclusions.

Clearly the first requirement for a language to support class inheritance is that it supports some notion of class. Although not all languages that we examined have a notion of class we will abstract this fact and examine in what ways the synchronization mechanisms would interfere with inheritance.

ABCL/1 and SR do not support class inheritance and actually it would be difficult to do so because of the internal structure of their objects. ABCL/1 objects are not structured in terms of methods. The message acceptance forms may associate different actions with a message depending on where the message pattern appears within an object's script. In fact an object's script may consist of a single unit of code because of the use of select and nested acceptance of messages. This makes it difficult to use class inheritance, like say in Smalltalk-80, since in many cases it would be reduced to editing this single block of code. SR presents similar problems since often the implementation of objects is a single sequential block of code containing message acceptance statements.

Emerald does not support classes and class inheritance. The main difficulty for supporting class inheritance in Emerald would be whether and how to inherit the process that may be associated with an object and because of the use of monitors for synchronization. In order to synchronize new operations added by a subclass with operations that were defined in the superclass the monitor would have to be completely redefined in the subclass.

Trellis/Owl supports class inheritance but its effective use presents several difficulties when the classes involved use the synchronization primitives. A first problem comes because of mutual exclusion. When a new method is added in a subclass all inherited methods must be examined to check if the new method interferes with any inherited methods. If such methods are found they should be overridden in the subclass by methods

that are mutually exclusive with the method added in the subclass.

Other problems are caused by condition synchronization. If threads executing the methods added in a subclass are suspended until some condition becomes true about the state of the object, inherited methods that make this condition true should be located and modified so that they wake the suspended thread by signaling the corresponding wait queue. The same thing should also be done in the opposite sense, the threads executing methods added by the subclass should wake threads that were suspended by executing an inherited method.

In Hybrid the situation is better for mutual exclusion, since method execution is by default mutually exclusive, but in the case of condition synchronization the situation is worse. When a delay queue is open it means that the necessary conditions are met for executing a method associated with the delay queue. Threads that execute a method do not check if the necessary conditions are satisfied. This means that besides the situation described above for Trellis/Owl, in Hybrid we also have to make sure that, if the execution of a method added in the subclass invalidates the conditions necessary for the execution of inherited methods, the associated delay queues are closed. This problem may be illustrated with the example of the bounded buffer presented in section 3. Imagine a subclass of the buffer that supports an additional operation that checks if the buffer is empty and, if not, removes an item from the buffer. If this operation removes the last item in the buffer it invalidates the necessary condition for the execution of the inherited operation `get`, namely that the buffer is not empty, so it should close the delay queue associated with `get`.

The languages POOL2 and POOL-T do not support inheritance. Although classes are structured in terms of methods there is a problem concerning the body of the class. America [Ame87a] explains that they have considered several ways of inhering the body of a class but none of them provided a satisfactory solution.

Caromel in his proposal takes the approach that the live routine of an object class should be systematically overridden in subclasses. Inheritance for other methods works as in sequential languages because methods other than the live method are not allowed to use synchronization primitives.

The synchronization mechanism of Guide interferes with inheritance in several ways. The separation of the activation conditions from the code of methods and the fact that methods do not contain any synchronization primitives, suggests that methods and activation conditions could be inherited independently. This is not the case because of the way that this synchronization mechanism is implemented. The code generated for methods by the compiler includes code that checks the activation condition and suspends the thread by using semaphores. As a result even if just the activation conditions of a method are modified in a subclass a new method has to be generated for the subclass.

Another problem is that the activation conditions may refer to names of methods. This has the effect that when new methods are added in a subclass the activation conditions of inherited methods may have to be modified for referring to the new method names. For example, if we defined a subclass of the bounded buffer and added an additional method that should be executed in mutual exclusion with inherited operations

the activation conditions of these methods would have to be rewritten. By contrast, in Trellis/Owl the new method could use the lock used by the inherited methods, and mutual exclusion would be achieved without modifying the inherited methods.

A problem that should be noticed with the languages we have examined so far is that in order to synchronize the execution of methods of a subclass with those of its superclasses the subclass has to access instance variables of its superclasses. This may not be desirable since it weakens encapsulation [Sny86,Ame87a].

A rather different approach to inheritance that alleviates some these problems has been taken in ACT++. The conditions under which a method may be activated are expressed in terms of a behavior name of the behavior abstraction associated with a class. The operations that are added by a subclass may be associated with a behavior name, defined in the subclass, that redefines an inherited behavior name. When the inherited behavior name is used with the `become` primitive in an inherited method it will, enable all the operations associated with its redefinition in the subclass.

```
class extended_buffer : public bounded_buffer{

behavior:
      extended_full_buffer ={ get(), getrear()}

      redefines full_buffer,

      extended_partial_buffer = { get(), put(), getrear()}
      redefines partial_buffer,
public:

      int getrear()
      {
            implementation of getrear ...
      }
};
```

Figure 13:

An example for illustrating this idea is to add a method `getrear` in a subclass of the bounded buffer presented in section 3. that removes an item from the rear end of the buffer. Figure 13 illustrates the definition of such a subclass called `extended buffer`. The redefinition of the behavior names `full_buffer` and `partial_buffer` defined in the superclasses has the effect that the execution of the statements `become partial_buffer` and `become full_buffer` in inherited methods will also enable the execution of the method `getrear`.

6.1 Discussion

First there are languages that by their construction are not suitable for supporting class inheritance. These are languages like SR, Emerald and ABCL/1. For such languages

delegation seems a reasonable alternative for code sharing, but it is not without problems [BY87]. In languages like Trellis/Owl and Hybrid inheritance is supported but it may be difficult to use. In languages that separate the concurrency features from the implementation of methods, like Guide and Caromel's extension to Eiffel, it is easier to inherit the sequential part of objects. The non-uniform approach has an additional problem because there are separate inheritance hierarchies for the different kinds of objects.

7 Conclusion

Concurrency is not orthogonal to other aspects of object-oriented programming. Although several object-oriented languages that provide support for concurrent programming have been designed and implemented their concurrent features interfere with their object-oriented features. Furthermore, the approaches taken for concurrency may have a considerable impact on the structure of applications in a way that is contrary to the principles underlying object-oriented programming.

Concerning the approaches taken for integrating concurrency in the object-oriented framework we have identified six categories of languages with respect to the evolution that undertook the concept of object for coping with the demanding requirements of the concurrent world.

A great variety of concurrency features is used by COOPLs in each category for supporting concurrent programming. Although notations for concurrent programming have been extensively studied during the past two decades and several concurrent programming languages have been designed and implemented, the combination and design of concurrent programming features that are best suited for object-oriented programming deserves further examination. We have addressed these questions in two ways. First, by comparing the concurrency features of some COOPLs based on an approach that differs from the more traditional views of concurrency and that, we believe, is consistent and better adapted to object-oriented programming. Second, by examining the interference of the concurrency features of COOPLs with class inheritance.

Although abstraction mechanisms conveying more information than ADTs about the behavior of objects are badly needed in COOPLs, very little has been done in this direction by COOPLs. This may be explained by the fact that it does not seem to exist a consensus on formal models for the specification of the behavior of concurrent systems, that would provide a basis for development of abstraction mechanisms that would extend the data abstraction mechanisms to concurrent languages.

We have not addressed a number of other important issues which include: typing, exception handling, persistence and transactions. At the current stage of development of COOPLs it does not seems that typing could be used any differently than in sequential OOPLs. We believe that the development of type systems for COOPLs is intimately related to the development of formal models for the specification of the time dependent behavior of objects, and the development of abstraction mechanisms that will extend data abstraction to include more information about such behavior.

Exception handling mechanisms are even more important in concurrent systems than they are in sequential ones. The failure of a process should not entail the failure of the whole system since other processes may be able to proceed.

Persistence is useful for the development of a host of applications. Providing support for persistence in a programming language frees the application programmer from the burden of explicitly managing persistence by using files. The atomicity properties of transactions are especially attractive in a system that has to deal with long lived persistent data. Although persistence on its own does not seem to interfere with concurrency features of a language it is not the same concerning transactions. The noninterference property of transactions seems contrary to the close interaction and communication of processes that characterizes concurrent programming and the independence of objects promoted by object-oriented programming.

The full integration of concurrency with all the other aspects of object-oriented programming presents several problems that deserve more attention. In this paper we have identified some of them and examined by comparing several languages how language design choices may make them more acute or eliminate them. However further work is required for gaining more insight in the nature of these problems and for developing languages that provide satisfactory solutions.

References

[ANSI83] American National Standards Institute, Inc., *The Programming Language Ada Reference Manual.* Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[Agh86] G.A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986. 4.32 agh.

[Ame87a] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. *BIGRE*, (54):281–289, June 1987.

[Ame87b] P. America. POOL-T: A Parallel Object-Oriented Language. In M. Tokoro A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 199–220, The MIT Press, Cambridge, Massachusetts, 1987.

[AOC88] G.R. Andrews, R.A. Olsson, and M. Coffin. An Overview of the SR Language and Implementation. *TOPLAS*, 10(1):51–86, January 1988.

[AS83] G.R. Andrews and F.B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[BB88] A. Bjornerstedt and S. Britts. AVANCE: An Object Management System. *SIGPLAN Notices*, 23(11):206–221, November 1988.

[BHJ*87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Ditribution and Abstract Data Types in Emerald. *Transactions on Software Engineering*, SE-13(1):65–76, Jan 1987.

[BY87] J-P. Briot and A. Yonezawa. Inheritance and Synchronization in Concurrent OOP. *BIGRE*, (54):35–43, June 1987.

[Car89] D. Caromel. A General Model for Concurrent and Distributed Object-Oriented Programming. *SIGPLAN Notices*, 24(4), April 1989.

[CH74] R.H. Campbell and A.N. Habermann. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science*, 16:89–102, 1974.

[Cox86] B.J. Cox. *Object Oriented Proggramming: An Evolutionary Approach.* Addison-Wesley, 1986.

[DKM*89] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Rivelli, and Rousset de Pina. A Synchronization Mechanism for Typed Objects in a Distributed System. *SIGPLAN Notices*, 24(4), April 1989.

[Gen81] W.M. Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software-Practice and Experience*, 11:435–466, 1981.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hoa74] C.A.R. Hoare. Monitors : an Operating System Structuring Concept. *CACM*, 17(10):549–557, October 1974.

[KL88] D.G. Kafura and K.H. Lee. *Inheritance in Actor Based Concurrent Object-Oriented Languages.* Technical Report TR 88-53 Departement Of Computer Science Virginia Polytechnic Institute and State University, 1988.

[KNP88] D. Konstantas, O.M. Nierstrasz, and M. Papathomas. An implementation of hybrid In D. Tsichritzis, editor, *Active Object Environments Technical Report, Centre Universitaire d'Informatique, University of Geneva*, pages 61–105, 1988.

[LHG86] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. In *Proceedings of the 13th ACM symposium on Principles of Programming Languages*, St. Petersburg, Florida, 1986.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *ACM SIGPLAN Notices, Proceedings OOPSLA '86*, 21(11):214–223, Nov 1986.

[Lis77] A. Lister. The Problem of Nested Monitor Calls. *Operating Systems Review*, 5–7, Jul. 1977.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *CACM*, 20(8):564–576, Aug 1977.

[LZ74] B. Liskov and S. Zilles. Programming with Abstract Data Types. *Proceedings of the ACM Symposium on Very High Level Languages, SIGPLAN Notices*, 9(4):50–59, 1974.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[MK87] J.E.B. Moss and W.H. Kohler. Concurrency Features for the Trellis/Owl Language. *BIGRE*, (54):223–232, June 1987.

[Nie87] O. Nierstrasz. Active Objects in Hybrid. *Object–Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices*, 22(12):243–253, Dec. 1987.

[Par72] D.L. Parnas. A Technique for Software Module Specification with Examples. *CACM*, 15(5):330–336, May 1972.

[Par78] D.L. Parnas. The non-problem of Nested Monitor Calls. *Operating Systems Review*, 12(1):12–14, 1978.

[Sny86] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 21(11):38–45, Nov 1986.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[SWL77] M. Shaw, W.A. Wulf, and R.L. London. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. *Communucations of the ACM*, 20(8):553–564, August 1977.

[TA88] A. Tripathi and M. Aksit. Communication, Scheduling, and Resource Management in Sina. *JOOP*, 24–36, Nov/Dec 1988.

[TN88] D. Tsichritzis and O.M. Nierstrasz. Application Development Using Objects. *Proc EUROINFO'88*, 15–23, 1988.

[Weg87] P. Wegner. Dimensions of Object-Based Language Design. In *Proceedings OOPSLA '87*, pages 168–182, ACM, Orlando, Florida, December 1987.

[YSTH87] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Corcurrent Programming*, pages 55–89, The MIT Press, Cambridge, Massachusetts, 1987.

[YT87a] Y. Yokote and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Corcurrent Programming*, pages 129–158, The MIT press, Cambridge, Massachusetts, 1987.

[YT87b] Y. Yokote and M. Tokoro. Experience and Evolution of ConcurrentSmalltalk. In *Proceedings OOPSLA '87*, pages 406–415, ACM, Orlando, Florida, December 1987.