TAKING ADVANTAGE OF INHERITANCE 
TO SPECIFY PARALLEL OBJECT BEHAVIOR

Antonio Corradi, Letizia Leonardi

Dipartimento di Elettronica, Informatica e Sistemistica - University of Bologna
2, Viale Risorgimento - Bologna - ITALY - Ph: (+39)51 6443001 - E-Mail: BOARI@BODEIS.cineca.it

This work has been supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR, under Grant 89.00012.69, and by ESPRIT "Parallel Computing Action", under Grant 4028.

ABSTRACT

The paper describes how PO takes advantage of inheritance to specify parallel object behavior at two different levels of abstraction: by programming intraobject scheduling and by expressing constraints. The paper shows low level mechanisms that implement intraobject concurrent strategies in terms of variables and methods of scheduling. Scheduling methods are subject to classification and inheritance: their combination forms the scheduler of objects. The paper describes the pre-defined constraints available in PO. The application of constraints to classes of objects related by inheritance produces constraint composition that is automatically translated into low-level specifications.

1. INTRODUCTION

Reusability and extensibility are key requirements in Software Engineering [16]. Object-oriented environments, such as Smalltalk-80 [10] and Lisp-based systems [14], meet these requirements by introducing important innovations by means of new concepts, such as classification and inheritance. Classification promotes reusability and inheritance stresses extensibility. By applying classification, object descriptions are collected together. This leads to a reuse of software specifications. By applying inheritance, a class of objects can derive part of the behavior of its instances from existing specifications. Therefore, the design of a class does not start from scratches (reusability), but evolves from existing specifications (extensibility).

At the same time, availability of parallel architectures (even massively parallel) poses the problem of their effective utilization. New paradigms and programming models seem to propose a solution. Besides the actor model [12], parallelism emerges in several object-based proposals (e.g. Orient84/K, POOL-T, ABCL/I and Concurrent Smalltalk [18]).

The introduction of parallelism into object environments brings out the problem of how to apply classification and inheritance to the concurrent parts of object specifications [18]. Most object-oriented proposals do not consider this feature at all. We, instead, consider classification and inheritance in the specification of concurrent behavior of objects an important direction of research [14]. In fact, we think this topic will strongly influence future research and development of object-oriented parallel programming languages.

This paper describes a parallel object model, the Parallel Objects model [6] (PO for short) to address the issues of classification and inheritance with regard to object concurrency. PO deals with concurrency following the pioneer experience of Simula-67 [13]. Differently from Simula-67, PO introduces true parallel activities within objects. In PO, classification and inheritance are used in the expression of the scheduling of intra-object concurrent activities.

The paper aims to describe the PO features that augment the expressive capacity with regard to parallelism in the direction of reusability and extensibility. Throughout the paper, PO is compared and evaluated with other proposals.

2. THE PO MODEL

The PO model combines the typical properties of object-based systems [17], such as uniformity, reusability and dynamicity, with the concept of parallelism. PO is an object system based on classes [10], [7]. A PO object is always created by a class: it is an instance of that class. A class describes all the components of its instances: basically, variables (that constitute the instance state) and methods (or operations). The operations are the only ones that can act upon and modify the state. Other objects can only request the execution of one of the operations defined in the interface of an object. Therefore, objects represent data abstractions. In PO, classes are objects too. This leads to uniformity and dynamicity (i.e., the possibility of changing instance specifications) of the PO environment. Classes relate to one another by the inheritance relationship to allow reusability and extensibility of instance specifications.

PO employs two levels of parallelism integrated with the concept of object. The first called INTER-OBJECT parallelism derives from the independent capacity of execution attributed to each PO object. A PO object has at least one internal thread of execution. We call it basic thread of objects.

Computation in PO environments derives from objects mutual communication, by requesting each other operations [7]. PO furnishes several modes of communications: one is synchronous and two are asynchronous ones.

Due to object asynchronous behavior, each object owns a private queue called Request Queue (see figure 1). The Request Queue buffers incoming requests. The object basic thread extracts from this queue requests and, according to local policy, decides its service. A PO object may also host several internal concurrent threads of execution. These threads (called activities) are tied to execute only within the belonging object. This is the second level of parallelism in PO: INTRA-OBJECT parallelism. In this case, the basic thread of an object assumes a more complex role. Because of its role, we call it scheduler or simply scheduler. The scheduler extracts requests from the Request Queue and, on the basis of the object state (i.e., of the scheduling state and the execution status of activities), creates new activities to serve requests (see figure 1). A new activity executes the method that corresponds to the selected request and terminates by returning a result. The scheduler controls all object activities: it owns a further queue, the Activation Queue, to keep all activity descriptors.

The intra-object parallelism of PO is an important innovation over object systems where objects can serve only one request at a time. Those objects behave as serializers of incoming requests and the object scheduling policy is generally embedded and not flexible [8]. These object systems achieve further parallelization either by generating new objects or by decomposing existing objects in smaller size ones. This perspective produces small computational entities, but it is greatly subject to dynamic reorganization if concurrent requirements tend to change. Let us note that decomposing an existing object into parts...
While in execution is extremely difficult. By accommodating parallelism within objects, PO allows a user to exploit parallel semantics intrinsic to applications. For example, a container of information can be mapped in a PO object that offers two operations, READ and WRITE. Because of intra-object parallelism, requests of READ an item from the container by readers are served in parallel.

As a second example, a buffer of finite size can be implemented by one PO object with two operations INSERT and EXTRACT. Even the activities that embody the service of INSERT and EXTRACT can execute concurrently.

A PO scenario tends to produce objects larger in size than serialized ones. A PO user can take advantage of the logical decomposition of applications and directly map logical entities in PO objects. Any addition of parallelism does not produce new objects, but only change the schedulers of existing objects.

In PO, internal object scheduling is expressed in a flexible and uniform way. A PO user can specify the concurrent object behavior by using the same constructs to specify the non-concurrent behavior of objects. Concurrency is expressed by means of scheduling methods and variables, the same as it is the object non-parallel part.

Methods and variables of scheduling constitute a part of a PO object again described in its class. This part is isolated as much as possible from the non-concurrent one. The separation between non-concurrent and concurrent specifications complies with a criterium of modularity and independence [3]. One may change the concurrent behavior of an object without altering the "normal" non-concurrent part.

PO relies not only on the notion of classification, but also on inheritance to reuse and extend the internal concurrency specifications: new intra-object scheduling strategies can derive from existing ones.

A PO user may also adopt a higher level perspective than scheduling programming. Instead of specifying scheduling variable and methods, a user can specify intra-object concurrent behavior by imposing constraints on the activity creation. Constraints are constructs of high level of abstraction similar to path expressions [2], but without their limitation [8]. The PO system automatically translates constraints into scheduling methods and variables, i.e. into low-level concurrency specifications. The relationships of classification and inheritance applies to constraints too. In fact, the constraints applied on an object derive from the composition of all constraints both from the ones specified in its class and from the inherited ones.

3. INHERITANCE OF INTRA-OBJECT CONCURRENT BEHAVIOR

Inheritance works not only for normal behavior, but also for intra-object concurrency. On the one hand, a user may rely on the PO environment that furnishes a default scheduling policy inherited by all objects. On the other hand, a user is free of explicitly specifying the internal scheduling of objects by structuring them in classes related by inheritance.

A user may install in his/her PO environment one of two different default policies:
- The MAXIMALLY SEQUENTIAL strategy. It implies that each object acts as a serializer of requests.
- The MAXIMALLY PARALLEL policy; in this case, each object scheduler creates an activity as soon as a request arrives.

Several object-oriented systems adopt the first strategy as default object scheduling strategy [18]. In this case, an object can be considered as a sort of monitor [11]. This policy may become too strict to describe all possible object semantics with regard to intra-object parallelism.

PO does not force a specific intra-object concurrency policy. In fact, when the default policy does not match user needs, a user explicitly specifies other strategies such as reader/writer or priority policies. When the user needs to intervene on object concurrent strategies, he/she normally chooses the maximally parallel policy as default.
This does not limit parallelism and is more suitable for producing a highly parallel object scenario. In the following, we assume that the PO environment adopts as default the maximally parallel policy. The presence of a programmable scheduler in each PO object favors synchronization schemes based on an a priori avoidance of interference between activities. In this case, the scheduler creates a new activity only when it does not interfere with other concurrent object activities. Therefore, the user does not specify synchronization when designing the object normal part. The separation between synchronization (scheduling) part and normal behavior is neat and leads to another dimension of modularity [3].

PO also supports synchronization tools more traditional such as semaphores [9] and monitors [11]. By using these tools, a user deals with synchronization only a posteriori, i.e. when the activities are in execution. This traditional case is beyond the scope of the paper. A PO user has two different ways of expressing object synchronization: he/she can define either scheduling methods and variables (i.e., programming at low-level) or apply pre-defined constraints (i.e., programming at a high-level). In any case, scheduling policies defined in other classes (super-classes) can be inherited and even additional strategies can be specified.

With the scheduler programming approach, the object scheduler is a combination of scheduling methods locally specified and inherited from other classes. The sub-section 3.1 describes the details of this low-level approach. Constraints impose restrictions on the creation of activities in a PO object. For example, a constraint may force mutual exclusion between activities that execute different methods within the same object. Constraints apply on the object interface, but they can also specify policies depending on the scheduling state. The sub-section 3.2 shows examples of constraints combined with inheritance. This produces reusability and extensibility.

### 3.1 Inheritance in Scheduling Programming

When a user programs scheduling at this low-level, he/she has complete visibility of the internal of an object. In particular, the user can access the Request and Activation Queues and the scheduling variables. With the scheduling programming, the user defines dedicated methods (scheduling methods) to his/her classes. Those form the code of the scheduler for their instances. In general, a class specifies one scheduling method for each operation defined in its object interface. The syntax used in the definition of methods and variables of scheduling is the same used to define the normal part. (Note that we use a simplified Smalltalk-like syntax [10], [11].)

We use some examples among the classical concurrent ones to describe the scheduling programming. Let us note that all the examples of this sub-section and of the next one do not specify the normal non-concurrent behavior of the defined classes. In fact, we assume that the chosen scheduling policies are always a priori. Therefore, the scheduling part is independent of the specific implementation of the normal part.

As the first example, we consider one class of objects (RWInfo, see figure 2) that obey to a reader/writer policy. The defined interface offers two operations: READ and WRITE. Readers request the execution of READ operations, while writers of WRITE operations. An instance of the class RWInfo can accommodate intra-object parallelism: more requests of READ can be served in parallel, each one by one activity. Of course, one request of WRITE can be served if and only if there is no service of the same request. Until the WRITE is terminated, the service of any other request must be postponed.

Each instance of the class RWInfo has a scheduler that executes the combination of all scheduling methods found in the inheritance graph. In particular, this is a simple example with a simple scheduler: the scheduler of each RWInfo instance executes in an endless loop the two scheduling methods called READ and WRITE.

As a second (more complex) example, we consider another class of objects (InfiniteBuffer, see figure 3) that implements a producer/consumer policy with no limit in the bufferable items. This class defines in its interface two operations: INSERT and EXTRACT. Producers request execution of INSERT operations, while consumers of EXTRACT operations. Instances of the class InfiniteBuffer can allow intra-object parallelism: one request of INSERT can be served in parallel with one request of EXTRACT. Of course, one request of INSERT is served if and only if there is no service of the same operation. For one EXTRACT request the above condition is only sufficient: in fact, it also necessary that there is at least a buffered item.

To take into account the current number of buffered items, we introduce a scheduling variable (#items). Note that only the scheduler handles this variable, thus avoiding any problem of mutual exclusion. Moreover, the scheduler acts only on the system state and not on the normal state, whose values can be subjected to continual actions of concurrent activities.

The introduction of a scheduling variable implies the definition of a second type of scheduling methods: the post type (*). Scheduling methods of type post contain actions that must be executed after the termination of an activity, i.e. a service. For example, a post-scheduling method that increments the variable #items will be executed by the scheduler as soon as an INSERT request.

Figure 2. Reader/writer policy: low-level scheduling approach.

Class RWInfo

<table>
<thead>
<tr>
<th>Inherits From</th>
<th>Object</th>
<th>&quot;the root of the inheritance graph&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance State</td>
<td>Normal</td>
<td>&quot;definition of the state for the normal part of each instance&quot;</td>
</tr>
<tr>
<td>System</td>
<td>&quot;no scheduling variables&quot;</td>
<td></td>
</tr>
<tr>
<td>Instance Operations</td>
<td>Normal</td>
<td>&quot;definition of the operations READ and WRITE for the normal part of each instance&quot;</td>
</tr>
<tr>
<td>System</td>
<td>&quot;The scheduler is expressed at low-level&quot;</td>
<td></td>
</tr>
<tr>
<td>Scheduling Method READ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg act</td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg &lt;- Request_Queue_IsIn (&quot;msg.op=READ&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>act &lt;- Activation_Queue_IsIn (&quot;act.op=WRITE&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(msg = NULL) and (act = NULL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling Method WRITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg act</td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg &lt;- Request_Queue_IsIn (&quot;msg.op=WRITE&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>act &lt;- Activation_Queue_IsIn (&quot;act.op=READ&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(msg = NULL) and (act = NULL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>End Class</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(*) The scheduling methods mentioned up to now can be therefore called pre-scheduling methods.
An analogous post-scheduling method is defined for decrementing after an read extraction (READ) operation service.

Let us note that defining items as scheduling variable (not as a normal variable) allows an instance of InfiniteBuffer to accommodate intra-object parallelism.

As an extension of the above class, a sub-class of the class InfiniteBuffer (FiniteBuffer, see figure 4) can be defined. This sub-class implements a producer/consumer policy with a limited buffer size. The buffer is handled in a circular way.

The class FiniteBuffer inherits the definition of its super-class: in particular, the methods and the variable of scheduling. This sub-class adds some definition: in particular, it defines one normal instance variable (Size) that represents the buffer dimension.

The limited dimension of the buffer modifies the scheduling policy: for a FiniteBuffer instance, one request of INSERT is served not only if there is no service of READ requests, but also if there is enough space in the buffer. For this reason, this class must override the scheduling method for serving INSERT requests to take into account this new condition.

This example introduces the combination of scheduling methods. Each instance of the class FiniteBuffer has a scheduler that forever executes the combination of all scheduling methods (non overridden) found in the inheritance graph.

### Figure 4. Producer/consumer policy: the low-level scheduling approach for finite buffers.

The combination is constituted by the following methods:
- the newly defined scheduling method INSERT;
- the inherited scheduling method EXTRACT;
- the inherited post-scheduling method INSERT;
- the inherited post-scheduling method EXTRACT.

#### 3.2 Inheritance in Constraint Programming

Constraints allow a high level perspective in the specification of concurrent object behavior. They define synchronization in terms of object interface, i.e. of operations names, similarly to path expressions [1]. However, differently from them, they allow more insight on object internals. There are policies where constraints are expressed in terms of dedicated scheduling variables and even parameters of activities.

To explain constraints, we use the same examples as in the above subsection. Constraints are again specified in the system part of classes as scheduling programming. For more details about constraints see [8].

In the first example, the class RWInfo, instead of specifying the scheduling policy by low-level scheduling, can introduce the following constraints:

- **MaxPar (READ)** any number of activities that serve READ requests can be concurrently in execution at the same time;
- **MaxSeq (WRITE)** only one activity that executes a WRITE is possible at a time within a RWInfo instance;
- **MutEx (READ, WRITE)** if one activity executes a WRITE, any READ is impossible and vice versa.

These constraints are composed to form the intra-object scheduling policy for each instance of the class RWInfo. PO automatically translates the composition of constraints into variables and methods of scheduling at low level (see sub-section 3.3).

As second example, the producer/consumer problem can be solved...
again, by using inheritance as before but applied on constraints. In this case, we introduce also constraint on the scheduling state. The class InfiniteBuffer can provide the following constraints:

- **MutEx (INSERT)** a request for the INSERT can be served only if there are no other INSERT activities in execution;
- **MutEx (EXTRACT)** a request for the EXTRACT can be served only if there are no other EXTRACT activities in execution and
- **SchedCond (EXTRACT, "#items > 0")** there is at least one item in the buffer.

The sub-class of InfiniteBuffer, FiniteBuffer, only adds an additional constraint:

- **SchedCond (INSERT, "#items < Size")** a request for the method INSERT can be served only if there is enough room in the buffer.

The scheduling policy for a FiniteBuffer results from the composition of the constraints defined in this class and its superclass.

In particular, the constraint composition for INSERT results from:

- MutEx (INSERT), inherited from the super-class InfiniteBuffer and
- the directly specified SchedCond (INSERT, "#items < Size")

while the constraint composition for EXTRACT is made of:

- MutEx (EXTRACT) and
- SchedCond (EXTRACT, "#items >0")

both inherited from the super-class InfiniteBuffer.

There is a basic rule that must be followed in the constraint composition for a single operation: **only the strictest constraints among the applicable ones have effect.**

For example, let us suppose a class C inherits MaxPar (A) from its super-class C1 and MaxSeq (A) from another super-class C2. The only effective constraint for the operation A is MaxSeq (A) because it is stricter than MaxPar (A).

Moreover, there can be also problems due to composition of constraints: in fact, some constraints are **incompatible** [8] and some constraints can be in **conflict.** For instance, several SchedCond constraints may introduce conflicts.

To avoid such unpleasant situations, a class may selectively release any constraint by explicitly subtracting it. A class may disable any of its constraints for itself and for its sub-classes. The same feature is offered for the normal part of a PO class.

### 3.3 From Constraints to Scheduling Programming

The examples of constraints given in the above sub-section automatically produce the scheduling programming part presented in the subsection 3.1. The PO environment furnishes a tool that automatically translates the constraint composition in scheduling methods (of pre- and post-type).

Let us note that one constraint can produce more than one scheduling method and vice versa, more constraints can be composed to define one single scheduling method. In fact, on the one hand, a constraint can represent a mutual condition on two operations, as MutEx (INSERT, EXTRACT), and therefore it has an impact on the scheduling methods of INSERT and EXTRACT. On the other hand, the constraint composition for a class InfiniteBuffer is derived from the translation of two constraints, MutEx (EXTRACT) and SchedCond (EXTRACT, "#items >0")

### 4. CONCLUSIONS AND FUTURE WORK

The paper describes how PO employs inheritance to specify parallel object behavior. PO allows the user to take advantage of reusing mechanisms at two different levels of abstraction: at scheduling programming and at constraint level.

Moreover, PO allows a user to enlarge the set of pre-defined constraints further pushing towards reusability and extensibility.

The current and future work on PO is about its implementation on massively parallel architectures [15]. In particular, we are implementing the support for PO on a Trasputer-based system, a MEIKO Computing Surface [5]. We have chosen a massively parallel architecture because it allows us to evaluate the effectiveness of inter-and intra-object parallelism. In fact, using a such kind of target, we can also map one single PO object on more than one single physical node. This truly implements intra-object parallelism when it is necessary to have real parallelism. Currently, we are testing the performance of the PO support depending on the requirements of applications.

### REFERENCES


