Sequential and Concurrent Object-Oriented Programming

Bertrand Meyer

Interactive Software Engineering

USA

Abstract

How can the object-oriented model cover concurrent programming as effectively as it addresses sequential computation? The answer proposed in this article suggests that a modest adaptation to the semantics of object-oriented programs may suffice. This includes introducing an explicit notion of processor (but not of process, a concept which object-oriented techniques already cover), explicit declarations for "separate" entities, a new semantics for preconditions on routines handled by different processors, and "lazy wait" for implicit resynchronization.

1 INTRODUCTION

Object-Oriented design and programming techniques appear superior to traditional approaches in sequential programming. The next step, essential for such application domains as real-time processing, operating systems and distributed computation, is to support concurrent programming as well.

References [1] and [8] survey the state of the art in this field, reviewing ongoing efforts to extend various object-oriented languages with concurrent mechanisms. Reference [2] discusses an Eiffel-based concurrency model.

This presentation introduces a method for handling both sequential and concurrent object-oriented programming in a single framework, starting from two basic observations: the retained mechanism should be as close as possible to sequential object-oriented programming, narrowing down the semantic differences between concurrent and sequential computation to the strict essentials; and it should retain compatibility with assertion-based techniques needed to establish, at least informally, the correctness of object-oriented software.

2 PROCESSES AND PROCESSORS

In object-oriented programming, the basic concept is the class, describing a set of objects (the class's instances). Concurrent programming usually relies on the notion of process, or task; a process may be an instance of a process (task) type.

It is hard to miss the analogies between objects and processes (or between classes and process types). Both categories of constructs support:

- Local variables (attributes of a class, variables of a process or process type).
- Persistent data, keeping its value between successive activations.
- Encapsulated behavior (a single cycle for a process; any number of routines for a class).

Such strong similarities, with the last observation pointing to classes and objects as the more general concepts, suggest that concurrent object-oriented programming does not need a specific "process" construct.

One may point, of course, to an apparent difference: objects are "passive", waiting for external solicitations (calls to routines of their class), whereas processes are "active", having a script of their own to execute.

Closer examination, however, reveals two reasons why we should not attach too much significance to this distinction:

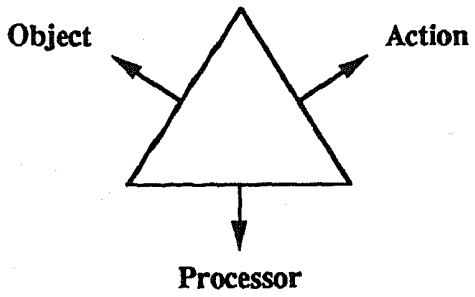
- First, we may view the routines applicable to an object as scripts; the only difference, then, is that objects have more than one script. The extra generality (not being limited to just one script) may then be viewed as a benefit, not a liability.
- Limiting an object's available scripts to just one raises the problem of how objects (processes) request services from each other. The standard object-oriented mechanism of feature call[1] (message passing) would not work directly any more; special synchronization mechanisms would be required.

[1] "Features" are the operations (commands or queries) applicable to the instance of a class. They include "routines" (computations) and attributes (data field accesses).

If we do not need processes and avoid introducing an explicit distinction between active and passive objects, what remains to cover the difference between sequential and concurrent computation is the notion of *processor*.

A processor is a separate thread of control capable of supporting the sequential execution of operations on one or more objects. It may be associated with a physical processor, for example a computer on a network, but may just as well be time-shared with other processors on a computer. As a result, the mechanism described below will potentially support distributed processing (where the processors are physically distinct computers) as well as multiprogramming (where the processors are supported by operating system processes) and coroutines (where the processors are threads in a common process).

The view of object-oriented computation which emerges is that of a triangle: computing consists of applying *operations* to *objects*; to do so requires the appropriate mechanisms – *processors*.



Object-oriented programming has been quite effective at capturing the first two aspects, by attaching the description of actions (routines) to the description of objects (classes). In ordinary sequential computation, there is only one processor, and so it remains implicit. With concurrent computation we need to make processors explicit.

A processor will be assigned to every object; the processor is in charge of executing any feature call on the object (of the form *a.f*, where *a* is an entity attached to the object, and *f* an exported feature of the corresponding class). The processor is said to **handle** the given object. Particular calls are said to be executed **on behalf of** the processors handling the calling objects.

3 A NOTE ON CSP

It is useful to compare the approach taken here with the well-known CSP model [3, 4].

One major difference is the role of processes in CSP; here, as noted, we do not need this concept, not because it is irrelevant but because it is covered by the notion of class.

Another difference is the role of communication as the basic mechanism in CSP, synchronization being viewed as a special case (transmission of an empty message). This provides an elegant structuring method, unifying various aspects of concurrency. In object-

oriented programming, however, the situation is different since communication is already present as a part of fundamental operation – feature call. This operation is written (using Simula or Eiffel notation) as

target.feature (*arg1*,....)

and denotes the application of *feature* to the object to which *target* is attached. This is a communication action; but if we required this communication to be blocking (forcing the caller not to proceed until *feature* had completed its execution), we would defeat the purpose of concurrent computation, which is to let various sub-computations to proceed in parallel.

Instead of using CSP-like mechanisms, then, it is appropriate to continue relying on feature call as the basic object-oriented operation, and view the introduction of parallelism simply as the introduction of separate processors capable of handling concurrently the features applied to various objects.

4 THE CONTRACT MODEL

An important idea of the theory of object-oriented design is the notion of programming by contract [5, 6]. The relation between an object requesting a service (by calling a feature) and the object providing that service may be viewed as a client-supplier relation, governed by mutual obligations and benefits. Assertions serve to express these obligations and benefits: a routine's precondition binds clients and protects the supplier (the routine); a postcondition binds the supplier and guarantees a certain result to the clients; an invariant clause expresses general consistency conditions.

As an example, here is the outline of a generic *BOUNDED_QUEUE* class similar to what may be found in the Eiffel Data Structure Library[2]:

[2] The use of a side-effect-producing function *get* does not conform to standard conventions in the Library.

```
class BOUNDED_QUEUE [T] export
  put, get, full, empty, ...
feature

  put (x: T) is
      -- Add x to queue.
    require
      not full
    do
      ...
    ensure
      not empty
    end; -- put

  get: T is
      -- Oldest element, removed from queue
    require
      not empty
    do
      ...
    ensure
        not full
    end; -- get
  ...
end -- class BOUNDED_QUEUE
```

The contract for *put* (*x*) is expressed by the following table:

	Obligations	**Benefits**
Client	Queue not full	Queue not empty; *x* inserted
Supplier	Must insert *x*	Queue not full (some space left)

5 CONTRACTS AND CONCURRENCY

A fundamental consequence of the use of the contract model as a guide to build and understand software architectures is the "**no hidden clauses**" rule: The precondition, postcondition and invariant are (within the limitations of the assertion sublanguage) the only conditions that govern the relationship between client and supplier.

This property means that an "obligation" entry in a table such as the above is not just "bad news" for the party to which it applies, but also partly good news since it means that the party is not expected to meet any other unexpressed condition; in other words, the obligation is not just necessary but also sufficient. It expresses the full correctness requirement.

In the above example, then, a client of *put* may guarantee the correctness of a call by making it of the form

```
/1/:  if not q.full then
          q.put (a)
      end
```

or (given the postcondition of *get*, and assuming the call to *get* is itself guaranteed to be correct):

```
/2/:  t := q.get;
      q.put (a)
```

This property is what makes the contract model practically useful.

Unfortunately, it cannot apply to concurrent situations. Assume that the processor handling /1/ or /2/ is different from the processor handling *q*. Then between the test in /1/ (or the call to *get* in /2/) and the call to *put*, any number of processors can call features such as *put* on *q* on behalf of other clients, making the precautions in /1/ and /2/ totally useless.

In other words, merely ensuring the precondition before a call does not guarantee correctness any more if the client and the supplier are handled by different processors.

This suggests the inescapable consequence: the sequential contract model does not hold as such for concurrent computation.

The requirements expressed by a precondition continue, however, to condition the correct execution of a routine. For example, we cannot write a correct version of *put* unless we can be sure that the queue is non-full on entry. In other words, we still need the precondition, but not (if the call is handled by a different processor) under its usual semantics of correctness condition on the caller.

What then should the semantics be in such a case? The practice of concurrent programming suggests that, for a client, the condition

```
not q.full
```

means that the request *q.put* (*x*) will not be served **until** the condition is satisfied. We may choose, then, to interpret preconditions as wait conditions, rather than correctness conditions, if the client and the caller are on different processors. (If, in light of the discussion at the beginning of the paper, we want to draw analogies with established approaches to concurrent computation, objects then appear closer to monitors than to processes).

6 SEPARATE ENTITIES

The semantics of a correctness condition is, of course, quite different from that of a wait condition. If a client class contains the call

q.put (x)

the precondition of *put* may now mean either of two things:

- If client and supplier are handled by the same processor, the call is only correct if the client guarantees not *q.full.*
- If the processors are different, the precondition simply means that the client may not be served until the suppliers satisfies *q.full.*

With such a considerable difference in semantics, it should be immediately clear from the text of the client class which interpretation is the correct one. Otherwise the uncertainty would prevent human readers from understanding the class, and compilers from generating the correct code.

The suggests that entities which may become attached to object handled by different processors should be specially identified. In an approach promoting the static description of software properties (such as types), such identification should be in the form of a static declaration. The declaration is of the form

/3/: *x:* **separate** *A*

and means: "Any object to which *x* may become attached at run-time will be of type *A*, or of a descendant type; any such object may be handled by a processor other than the processor handling the current object; as a consequence, the precondition of any routine applied to *A* will have the semantics of a wait condition".

In contrast, the standard form of declaration:

/4/: *x: A*

guarantees that objects attached to *x* will be handled by the same processor as the current object. A declaration of the form */3/* does not mean that the processor of *x* must be different, but only that it may be. Because of this possibility, however, preconditions on routines applied to an entity declared as "separate" will always have the semantics of wait conditions.

Consistency requires an obvious rule: in an assignment of the form

$x := y$

if the source *y* is separate, the target *x* must be separate too. The same rule applies to argument passing in routine calls, where *x* is the formal argument corresponding to the actual argument *y*. A composite expression *y* will be said to be separate if it involves one or more entities declared as separate.

Separate entities yield a special behavior for object creation. The Eiffel instruction

x!!

or

x!!f (...)

creates a new object, initializes its fields to default values, applies *f* to it with the arguments given (in the second form), and attaches it to *x*[3]. If *x* has been declared as

[3] This syntax for object creation is that of Eiffel version3. The earlier form was *x.Create (...)*.

separate, this now has the further effect of "grabbing" a new processor, physical or virtual, and assigning it to handle the object.

7 COMMENTS

Three important observations are in order.

First, we have narrowed down the semantic difference between sequential and concurrent computation to a very simple notion: the difference in precondition semantics for routine calls. This seems as good as any other characterization of parallelism.

This leads to the second comment. Designers or would–be designers of concurrent systems often comment that the originator of a request should not have to know which processor (for example in a network) will handle the request. The "**separate**" declaration may seem to conflict with this laudable goal. But in fact it does not. The identity of the processor that handles the request is indeed irrelevant to the client; but whether this processor is the same as the client's processor or another is very relevant. The semantics of the call cannot be the same in both cases, if only because in the second case the client may have to wait. It would be wrong, then, to take away from the client designer the responsibility of indicating whether he wants the call to be handled "separately" or not. Any further indication (such as *which* separate processor to use for the call) is an implementation decision, and may be left implicit; but that particular decision – handle by the same processor or another – is the one which must remain explicit.

The last comment addresses a possible objection to the use of preconditions as the central tool to characterize the semantic difference between sequential and concurrent computation. As users of Eiffel knows, assertion checking may be turned on or off at run time as a result of a compilation switch. Is it not dangerous, then, to attach that much semantic importance to preconditions in concurrent object-oriented programming?

Such an objection misses, however, the true nature of assertions. Assertions are not primarily a debugging or run-time checking tool. Instead, one should view assertions as full-fledged components of classes. In the form for *put* given in section 4, the precondition and postcondition belong to the routine as much as the *do* clause. They express the routine's specification.

Although this may appear paradoxical, the compilation option that switches run-time assertion checking on or off does *not* affect the semantics of the language. This is because the semantics of any language is defined for correct programs only; but a program whose execution may violate an assertion is incorrect! (The definition of a correct class in Eiffel is precisely that its **do** clauses are compatible with its assertions).

To a practicing programmer, the argument may appear specious, since checking assertions at run-time may be the best way to determine that a class is incorrect. But in principle it should be possible to prove class correctness statically; run-time monitoring is only an imperfect solution.

In any case, assertions are part of the software, whether or not monitored at run-time for debugging purposes. For "separate" entities, preconditions will always be checked, although for a different purpose.

8 PREDEFINED CONSTRUCTS AND LIBRARIES

The ability to declare an entity as "separate", with the associated change in precondition semantics, seems conceptually sufficient to describe general parallelism.

In practice, programmers will at some point need to give the practical indications governing execution: whether it will be truly parallel or quasi-parallel (coroutines); in the former case, how many physical processors (or operating system tasks) are available, and where they are to be found; etc.

Rather than through language constructs, programmers will express these indications by calls to routines of a library class (or several) designed specifically for that purpose. Classes which need to use these facilities should normally inherit from the corresponding library class.

Other features of this class serve to tune the details of the mechanism, for example to assign priorities to the processors waiting to be served by a certain other processor. This does not invalidate the basic semantics of the mechanism, which leaves unspecified the order in which these requests will be handled.

It is also through library features that we can establish the connection between the virtual processors used in this presentation and their actual hardware or operating system basis.

In particular, a "separate" entity need not be the target of a *Create* operation and (as described above) grab a new processor; instead, it may become associated, through a library call or some other mechanism, with a physical processor – for example a node in a network of workstations.

The approach followed here is the same as for exception handling: a simple language mechanism, an library mechanisms for fine-tuning specific details.

9 ATOMICITY AND DUELS

The contract model implies that the unit of granularity is the execution of routines.

The principal criterion for a class to be correct is that for every routine *r*:

$\{INV \ \& \ pre_r\} \ do_r \ \{INV \ \& \ post_r\}$

In other words: starting fro a state satisfying the class invariant and the routine precondition, the body will yield a state satisfying the invariant and the postcondition.

Consider a processor which is executing a routine on an object on behalf of some client. We might think of some mechanism allowing another "more important" client to interrupt this execution and get served right away; only then would the original client's execution resume. This concept is known as express messages [8].

Such a mechanism would, however, conflict with correctness requirements: if we allow executions to be interrupted, we cannot guarantee any more that they will preserve the invariant. Producing an object which does not satisfy the invariant of its own class is probably the worst disaster that may occur during the execution of an object-oriented program. (Another source of such a situation is static binding).

As a consequence, the mechanism described here does not directly support express messages.

It may be necessary in some cases, however, to *cancel* the execution of a routine (to free its processor). The library mechanism supporting this is used under the form

a.stop

and may only have an effect if some processor is executing a routine on *a*. The effect is then to raise an exception in that routine.

The resulting situation may be called a **duel**. Let *b* be the originator of the above *stop* request. If *a* has protected itself against the "stop" execution, then *a*'s execution is not impacted and an exception is raised for *b*. Otherwise it is *a* which gets the exception.

With proper settings this in fact makes it possible to obtain the equivalent of express messages. This solution avoids the danger mentioned above since it raises an exception if a routine is interrupted before normal termination; then part of the task of the exception handler (**rescue** clause) is to restore the invariant [5, 6].

10 STRICTNESS AND LAZY WAIT

Assume that a processor *P* executes the following routine call on behalf of a certain object:

x.rout

Assume *x* is separate. Then *P* may proceed with the subsequent operations without waiting: not having to wait is indeed the aim of making *x* separate, and the central benefit of parallel computation. In most cases, however, *P* will eventually need to use some of the results produced by *rout*; at that stage it should wait if the processor in charge of *x* has not finished executing *rout* or is busy with some other computation.

An important idea behind the mechanism described here is that in such a situation programmers should not have to write an explicit re-synchronization instruction to request waiting; instead, the wait, if needed, should occur automatically whenever *P* needs access to the value of *x*. More precisely, *P* will wait for *x* to be available when (but only when) it must perform a **strict** operation on *x*.

Strict operations on an object include the following cases (see [7] for a more general definition of strictness):

- Arithmetic operation such as addition.
- External operations such as *print*.
- Use as target of a routine or attribute application: *x.rout* or *x.attr* require *x* to be ready an so will make *P* wait if the server processor is not ready.

On the other hand, some operations are not strict in Eiffel and will not make *P* wait. These operations include in particular:

- Use of *t* as right-hand side of an assignment instruction *u* := *t*, at least for the most common case in which the values of *t* and *u* are references to objects, not the object themselves. (Then it does not matter that the object's processor is not available as long as we have a reference to that object).
- Use of *t* as argument to a routine, for arguments passed by reference.

The resulting mechanism, which yields a simple and general method for programming concurrent applications, may be called **lazy wait**. An earlier use may be found in [2].

As a simple example, consider the following extract from a binary tree class, where function *nodes* gives the number of nodes in the tree:

```
class BINARY_TREE [T] export
        left, right, nodes, ...
feature
    left, right: BINARY_TREE [T];
    nodes: INTEGER is
            -- Number of nodes in this tree
        local
            ln, rn: INTEGER
        do
            if not left.Void then
                ln:= left.nodes
            end;
            if not right.Void then
                rn:= right.nodes
            end;
            Result := ln + rn + 1
        end; -- nodes
end -- class BINARY_TREE
```

The routine *nodes* is a standard recursive computation.

If, however, some parallel hardware is available, we can go further: by declaring *left* and *right* as **separate**, we let the subcomputations *left.nodes* and *right.nodes* proceed in parallel, themselves sprouting many others (dispatched according to the number of physically available processors); the only strict operation is the addition, and only it may cause waiting.

11 CONCLUSION

This presentation has described an approach to concurrent object-oriented computation, and the rationale that led to it.

No implementation is available as yet, and some details clearly require further work. The design described seems, however, to ensure a minimal departure from the concepts of sequential object-oriented computation, and to retain compatibility with the assertion concepts which are so essential to the understanding of this field.