Systematic Concurrent Object-Oriented Programming[1]

Bertrand Meyer

Interactive Software Engineering Inc.
270 Storke Road Suite 7
Goleta, CA 93117 USA
E-mail: <bertrand@eiffel.com>

ABSTRACT

How can the object-oriented model cover concurrent programming as effectively as it addresses sequential computation?

This article suggests that a modest adaptation to the standard mechanisms of object-oriented programming 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.

The mechanism is applicable to many different forms of concurrency, such as shared memory and distribution, as well as real-time applications. If there is only one physical thread of control, it yields a facility for coroutine programming.

The presentation does not just describe the mechanism but also discusses in detail the reasoning that led to it, based on a set of strict requirements and on an analysis of the full implications of the object-oriented method, in particular compatibility with the Design by Contract view of software construction. It illustrates the result through a number of examples borrowed from diverse application areas of concurrency.

1 INTRODUCTION

To judge by the looks of the two parties, the marriage between concurrent computation and object-oriented programming – a union much desired by practitioners in such fields as telecommunications, high-performance computing, banking and operating systems – appears an easy enough affair to arrange.

This appearance is deceptive: the problem is a hard one. This article points the way towards a possible solution. The precise problem it examines is a restricted one:

> *What is the simplest, smallest and most convincing extension to the method of systematic object-oriented software construction that can address the needs of concurrent and distributed computing as well as those of sequential computation?*

---

[1] Originated from an invited presentation at TOOLS PACIFIC 1992, Sydney. A shorter and slightly revised version with same title appears in *Communications of the ACM*, 36, 9, September 1993, pages 56-80.

In other words, the article does not claim to discuss concurrency and distribution in a general and unbiased way. Rather, it takes the object-oriented paradigm as a given (on the basis of its contributions to the production of quality software) and investigates how best to adapt it so that it covers concurrent applications as well as sequential ones.

The word "systematic" as it appears in the title of the article will provide strong guidance to our search for this minimal extension. We are interested in an approach which makes it possible to reason about software systems in a precise way, by extending to the concurrent case the systematic techniques, known as "design by contract", which can be applied to the systematic (although not necessarily fully formal) development of sequential object-oriented software.

A word of warning as to the actual ambition of the model discussed below. No representation is made that the work as reported is final, and a number of possible criticisms are discussed in the last section. I do think, however, that in its discussion of how the mutual attraction between object-orientation and concurrency can be turned into a reasonably happy marriage this article raises a number of questions which, although crucial for both theoretical understanding and practical implementation of concurrent object-oriented computation, have not been addressed (or in some cases even mentioned) by previous work on the subject, and which will somehow have to be answered before a solution can be widely accepted and applied.

2 SIMILARITIES AND CONTRADICTIONS

The property which at first glance suggests an easy match between the ideas of concurrency and object-orientation is the remarkable similarity between the basic constructs of both object-orientation and concurrency. It is hard to miss the analogies between objects and processes, or more accurately between the underlying abstractions: 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 their value between successive activations.
- Encapsulated behavior (a single cycle for a process; any number of routines for a class).
- Heavy restrictions on how modules can exchange information.
- A communication mechanism usually based on some form of message passing.

It is not surprising, then, that researchers have tried to unite the two areas. But although existing designs (for surveys see [18] and [1] as well as a recent thesis [16]) existing designs have introduced many fruitful ideas, it is fair to state that so far none has succeeded in providing a widely accepted mechanism for concurrent object-oriented programming. The primary reason is probably the undue complexity of most of the proposed solutions, which tend to add the full power of an independent concurrency facility to an object-oriented language, or conversely. Another reason is that little of the existing literature devotes much attention to correctness issues, and more generally to the possibility of systematic reasoning about concurrent object-oriented programs.

The mechanism described in this article attempts to remedy these limitations at least in part. The syntactical extension which it brings to an object-oriented language (Eiffel) is the smallest feasible – one new keyword. A new library class with four procedures is also provided to adjust the behavior in special cases. The mechanism makes the most possible use of existing object-oriented facilities – classes, inheritance, assertions, argument passing, deferred classes – to cover such concepts of concurrent computation as exclusive access, processes and synchronization.

Section 3 gives the criteria that have governed the search for the mechanism. Section 3 explains in detail how the mechanism was derived from the criteria. Section 4 presents a number of example applications, in various areas of concurrent programming. Section 6 is a precise The aim in preparing this article, and particularly the detailed justification of sections 3 and 4, was to involve the reader in the derivation of the mechanism, hoping that the result would in the end appear almost inevitable. For this reason, the discussion of sections 3 and 4 may appear somewhat slow. The reader who prefers to get to the result right away may choose to go over the sections in a different order: first 6 to discover the mechanism; then 5 to see its applications; then 3, 4 and 7 to understand the motivation and the conclusions.

3 CRITERIA

The design of a proper concurrent object-oriented mechanism must satisfy a number of criteria. The combination of these criteria, as defined below, places rather strict constraints on the possible concurrency mechanism – to the extent that one might wonder at first whether any solution is possible at all.

(***SIDEBAR 1***)

Basic terminology.

Object-oriented development is based on the notion of class. A class describes a set of potential run-time objects, defining them entirely by the applicable operations, or features. Any object created at run time from the pattern defined by the class is called an instance of the class.

The features of a class are of two kinds: routines, which describe computations applicable to instances of the class; and attributes, which describe data fields attached to instances of the class. For example, a class *CAR* may have attributes *weight* and *speed*, and routines *start, stop, accelerate, average_speed*.

A routine is either a procedure, which may change the object to which it is applied but does not directly return a result, or a function, which computes some information about the object and returns that information as a result. In both cases, the routine may have one or more arguments. In the example *start, stop, accelerate* will be procedures, and *average_speed* will be a function, computing the average speed since a certain starting time; the function will have one argument, representing that time.

The basic computational mechanism is feature call, which applies a feature to a certain object, known through a name in the software text, or entity. All entities are declared. For example, with the declarations

c: *CAR*; *x*: *REAL*

a call to feature *average_speed*, used here in an assignment, could have the form

x := *c*.*average_speed* (*0*)

which assigns to *x* the value of the average speed of the object attached to *c* (an instance of *CAR*) since time 0. The notation used here for calls is **dot notation**, which includes the following components: an entity representing the target object of the call; a dot (.); a feature name; a list of actual arguments, if any, in parentheses.

Other examples of calls are

c.*accelerate* (*20*);
x := *c*.*speed*

A call such as any of the above is executed as part of the text of some routine – itself executed on a certain object C_OBJ, the client object of the call. The object to which a call applies (the object attached to *c* in the examples) is called the supplier object.
A feature such as *speed* may be implemented as either an attribute or a function. Principles of information hiding and uniform access implies that this choice of implementation should make no difference to clients. The notation for feature calls, and the standard form for class documentation, known as the short form of a class, are indeed the same in both cases.

Minimality of mechanism

Object-oriented software construction is a rich and powerful paradigm, which, as noted above, would intuitively seem to be ready for supporting concurrency.

It is essential, then, to aim for the smallest possible extension. Minimalism here is not just a question of good language design. If the concurrent extension is not minimal, some concurrency constructs will be redundant with the object-oriented constructs, or conflict with them, making the programmer's task hard or impossible. To avoid such a situation, we must find the smallest syntactic and semantic epsilon that will give concurrent execution capabilities to our object-oriented programs.

Full use of inheritance and other object-oriented techniques

It would be unacceptable to have a concurrent object-oriented mechanism which does not take advantage of all object-oriented techniques, in particular inheritance.

(***SIDEBAR 2***)
Deferred classes and features:
A class is deferred (or abstract) if it has one or more features declared as deferred, that is to say, specified but not implemented. A non-deferred class or feature is said to be effective. A descendant of a deferred class (that is to say, a class which inherits from it directly or indirectly) is effective if it implements (or *effects*) all deferred features by providing effective forms.
A deferred class describes a general abstraction which may have many different realizations. It may also serve to capture a common set of behaviors by using an effective (non-deferred) routine, such as *live* below, which calls deferred ones (*setup, over, step*). Descendant classes will retain the common behavior and provide the specifics through effective versions of the originally deferred routines.

One of the most interesting contributions of object-oriented technology is its ability to support many different patterns of computation. Once you have identified a useful type of behavior, you can encapsulate it in a deferred class (see note 2) from which any class that uses that behavior will inherit. Such a class is similar to a process type in concurrent programming (a property which will play an important role below) and its general form may be expressed as:

```
deferred class PROCESS feature
    live is
            -- General structure with variants.
        do
            from setup until over loop
                step
            end;
            finalize
        end;
feature {NONE}
    setup is deferred end;
    over: BOOLEAN is deferred end;
    step is deferred end;
    finalize is deferred end
end
```

(The clause ***feature*** {*NONE*} introduces features which are not available to clients, being meant for internal use only. Features declared in a clause beginning with just ***feature*** without qualification are available for calls by any client and are said to be exported.)

Since routines *initialize, over, step* and *finalize* are deferred, descendants of *PROCESS* may provide effective implementations of these routines, corresponding to individual variants. There may be as many such classes as variants are needed. The overall behavior, however, is the same for all variants; it is determined by the effective routine *live*. In the example the structure of *live* involves a loop, but the same ideas are applicable to any other structure. Also, there will often be more than one routine such as *live*, covering several patterns of behavior that are known at the level of the deferred class.

This technique and many others (such as the use of polymorphism, static typing and dynamic binding to obtain flexible and safe software architectures) are essential to the object-oriented approach, and are potentially beneficial to concurrent programming as well. This has been recognized for example by recent changes to the parallel object-oriented language POOL [2], whose early versions did not support inheritance. The model presented here makes full use of all object-oriented techniques.

Compatibility with Design by Contract

Another central idea of what may be called the Eiffel methodology of object-oriented software construction is the notion of Design by Contract [11, 15]. According to this view, the design of a reliable software system should use a number of components (classes) which communicate with each other on the basis of precise definitions of obligations and benefits – contracts. The obligations and benefits are documented in the text of the software itself, where they appear in the form of **assertions**.

A contract governs the relation between client objects requesting services (by calling a feature) and supplier objects providing these services. The services are expressed by the routines of the supplier's class. For each routine, an assertion known as the precondition expresses the input requirements; it binds the clients and protects the supplier. Another assertion, the postcondition, expresses the properties ensured by any call to the routine; it binds the supplier and guarantees a certain result to the clients. Beyond the precondition and postcondition of its individual routines, a class is also characterized by its invariant, which applies to all exported routines, being in effect added to both their preconditions and postconditions.

A convenient example, which will help us repeatedly in our search for the proper concurrent extension, is a class describing queues (first-in-first-out container structures) of bounded size. The corresponding concurrent notion, developed below, will describe bounded buffers. Here is a sketch of the class, with the implementation details omitted. Preconditions are introduced by ***require***, postconditions by ***ensure***.

```
class BOUNDED_QUEUE [G] feature
    empty: BOOLEAN is
            -- Is there no accessible element?
        do
            Result := ...
        end;
    full: BOOLEAN is
            -- Is there no room left for more elements?
        do
            Result := ...
        end;
    put (x: G) is
                -- Add x as newest element.
        require
            not full
        do
            ...
        ensure
            not empty
        end;
    remove is
                -- Remove oldest element.
        require
            not empty
        do
            ...
        ensure
            not full
        end;
    item is
                -- Oldest element not yet consumed
        require
            not empty
        do Result := ...  end
```

```
feature {NONE}
        ... Secret features (used for the implementation) ...
invariant
        ... See below ...
end
```

Through its assertions, class *BOUNDED_QUEUE* describes a number of contracts governing the use of its features by clients. For example 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)

An important property of preconditions, which we will have to examine again in the concurrent context, is that they express the sole obligations that the client has to meet. This may be called the **no hidden clauses** rule of Design by Contract. For example, a call to *put* is guaranteed to succeed if the client has ensured the precondition, perhaps by writing it (with *q* of type *BOUNDED_QUEUE* [*X*] and *a* of type *X* for some type *X*) as:

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

or by prefacing it with a call to *remove*, whose postcondition implies the precondition of *put* (assuming that the call to *remove* is itself correct, that is to say, the precondition ***not*** *q.empty* initially holds):

```
-- /2/
        q.remove; q.put (x)
```

Another important use of assertions is the class invariant, which characterizes the consistency of a class. Assume for example an implementation of *BOUNDED_QUEUE* relies on the well-known technique of using an array of *array_count* elements, managed in a circular way as illustrated below. The details of the implementation are left to the reader; the following invariant will express its fundamental properties, in particular the need to keep one position unused so as to be able to distinguish between the empty and full cases:

```
array_count = capacity – 1; abs (next – oldest) < capacity
0 <= oldest; oldest <= capacity; 0 <= next; next <= capacity;
```

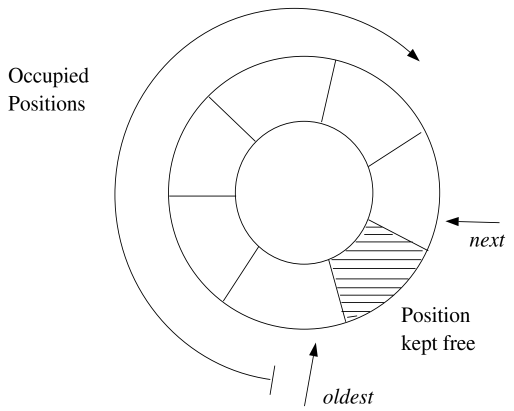

Figure 1: Queue implemented by circular array

Any call to an exported routine may assume that the invariant is initially satisfied (that is to say, it may expect to find the object in a consistent state); but it must also restore the invariant, in addition to ensuring its postcondition, on exit (that is to say, it must leave the object in a consistent state). We may thus picture the life of the object as a sequence of transitions between consistent states, as shown on the picture. The shaded squares represent states satisfying the invariant; the transitions represent calls to exported features, executed by clients. These states are the only ones in which the object is accessible to a new client.

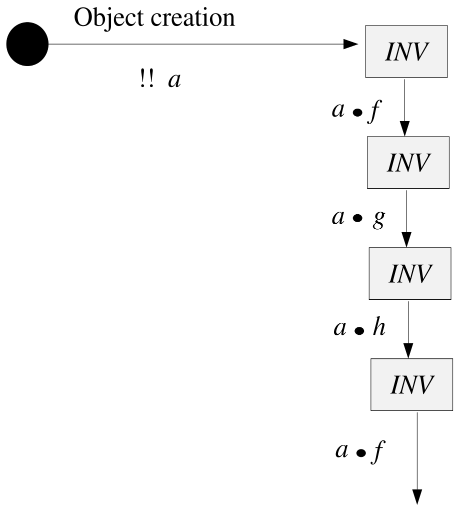

Figure 2: Object lifecycle

The notion of invariant is essential to the design of a proper exception mechanism. In the Design by Contract approach, an exception occurs when a routine is unable to fulfil its contract through the initially planned strategy. Then the routine may try again through a Retry instruction, usually after attempting to correct the source of trouble. Otherwise the routine will execute to the end its (implicit or explicit) Rescue clause, fail, and cause an exception in the client, which will then be faced with the same choice – Retry or failure. In case of failure the Rescue clause is not required to fulfil the contract as expressed by the postcondition (this would be success, not failure!), but it must restore the invariant, so as to leave the object in a consistent state for any later attempt at Retry.

Another reason which makes the notion of invariant important for concurrency is that it enables us to put the notion of "express message" [18] into a proper perspective. Express messages, as proposed, allow interrupting the execution of a routine on a certain supplier object, on behalf of some client, when a call comes in from another client which is deemed more important. An incoming express message will then get served right away; only then will the original client's execution resume. Defined in this way, however, such a mechanism conflicts 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 well-known source of such a situation is static binding.)

As a consequence of this analysis, the mechanism described below does not support express messages in the sense of [18]. It will, however, allow a VIP client to interrupt an earlier client, causing an exception in that client. Such a facility is compatible with Design by Contract, since as a result of the exception handling the invariant will be restored.

The relevance of invariants to concurrent programming was expressed by Hailpern [8], who uses a concept of "monitor invariant, [which] must be true when no process owns the monitor". Starting from an object-oriented basis, we do not need a special notion; class invariants will provide a way to characterize the invariant properties associated with monitors, processes and other constructs of non-object-oriented concurrent programming.

Provability

Design by Contract provides the starting point for a potential formal approach to object-oriented computation. Assuming proof rules were available for the inner details of an object-oriented language, we could use a general proof rule for calls to prove entire object-oriented systems. The proof rule for calls is fundamental because at the heart of object-oriented computation lie operations of the form

t.*f* (..., *a*, ...)

which call a feature f, possibly with arguments such as a, on a target t representing an object. The basic proof rule may be informally stated as follows:

> If we can prove that the body of of f, started in a state satisfying the precondition of f, terminates in a state satisfying the postcondition, then we can deduce the same property for the above call, with actual arguments such as a substituted for the corresponding formal arguments, and every clause of the form *some_boolean_property* in the assertions replaced by the corresponding property on t, of the form *t*.*some_boolean_property*.

For example, if we are able to prove that the actual implementation of *put* in class *BOUNDED_QUEUE*, assuming ***not*** *full* initially, produces a state satisfying ***not*** *empty*, then for any queue q and element a the rule allows us to deduce

$\{\textbf{\textit{not}}\ q.full\}\ q.put\ (x)\ \{\textbf{\textit{not}}\ q.empty\}$

where the assertions in braces describe the input and output assumptions respectively.

The proof rule, an adaptation to the object-oriented form of computation of Hoare's inference rule for procedures [9] (see also the presentation in [13]), may be expressed as:

$$\frac{\left\{ INV \wedge \bigwedge_{p \in Pre\ (f)} p \right\} \quad body\ (f) \quad \left\{ INV \wedge \bigwedge_{q \in Post\ (f)} q \right\}}{\left\{ \bigwedge_{p \in Pre\ (f)} t.p' \right\} \quad t.f \quad \left\{ \bigwedge_{q \in Post\ (f)} t.q' \right\}}$$

Here *INV* is the class invariant, *Pre* (f) is the set of precondition clauses of f and *Post* (f) the set of its postcondition clauses. Recall that an assertion is the conjunction of a set of clauses, of the form

$clause_1; \ldots\ clause_n$

The large $\wedge$ signs indicate conjunction of all the given clauses. The actual arguments of f have not been explicitly included in the call, but the primed expressions such as $t.q'$ indicate substitution of the actual arguments of the call for the formal arguments of f.

In the interest of conciseness, the rule is stated above in the form which does not support proofs of recursive routines. Adding such support, however, does not affect the present discussion. For details of how to handle recursion, see [9] or [13].

The reason for considering the assertion clauses separately and then "anding" them is that this form prepares the rule's adaptation, described below, to "separate" calls in the concurrent extension. Another property of the non-concurrent rule which is of interest as preparation for that extension is that the invariant *INV* must be taken into account in the proof of the routine body (above the line), with no directly visible benefit for the proof of the call (below the line). More assertions with that property will appear in the concurrent rule.

The provability requirement and the criterion of compatibility with Design by Contract have a major immediate consequence for concurrent computation. The bad news about proving the correctness of a class is that for every exported routine f you must prove a property of the form

$\{INV \wedge allpre\ (f)\} \quad body\ (f) \quad \{INV \wedge allpost\ (f)\}$

The good news, however, is that you only have as many properties of this kind to prove as the class has exported routines. (Attributes do not require any specific proof, but they may be involved in the invariant.) This provides a very strong guideline as to the granularity of exclusive object access in concurrent computation. If we allowed a supplier object to accept a new call from a client while a call from another client is in progress, the "good news" would not hold any more; the property depicted on figure 2 above, where the shaded states (before and after execution of exported features) are the only ones in which an object is accessible from the outside, would also fail to be satisfied. As a consequence, a proof of correctness would have to take into account all possible interleavings, producing a combinatorial explosion of properties to prove. This would in fact remove any hope of producing realistic proofs, even informal ones, or just of being able to reason about software texts in a systematic fashion.

As a result, any client accessing an object through a feature will have to be guaranteed exclusive access to the object throughout the duration of the call. The smallest permissible level of granularity for exclusive access to an object is the execution of a call to an exported feature.

This observation generalizes the comments made above about express messages. As noted then, nothing prevents us from interrupting an ongoing call as long as this is done properly: the previous client must receive an exception, so that it will be forced either to fail or to take corrective action. A facility supporting such a scheme, which we may call a *duel*, will be part of the mechanism described below.

Support for command-query distinction

An important part of the object-oriented method, which we should try to preserve in concurrent programming, is the necessity to maintain, whenever possible, a strict distinction between two kinds of features: commands and queries. A query, implemented as an attribute or function, is a feature that returns some information about an object. A command, implemented as a procedure, is a feature which may modify the state of an object.

The command-query distinction directs programmers to refrain from using a programming style that has become popular in recent years, especially in connection with the spread of C: functions that produce visible side effects. This practice endangers referential transparency (substitutivity of equal for equals).

The attempt to maintain a strict command-query distinction explains why class *BOUNDED_QUEUE* above has two separate features *remove* and *item*. Function *item*, a query, accesses the oldest element not yet removed, but does not remove it; successive calls to this feature will return the same value. Procedure *remove*, a command, removes the oldest element. A designer who does not apply the command-query distinction might be tempted to replace these routines by a side-effect-producing function *get*, which both removes an element and returns its value.

The presence of several concurrent clients accessing the same supplier will make it at first harder to ensure the command-query distinction; but this rule will in fact suggest some of the important properties of the concurrent mechanism.

Applicability to many forms of concurrency

A general criterion for the design of a concurrent mechanism is that it should make it possible to support many different forms of concurrency: shared memory, multitasking, network programming, distributed processing, real-time applications.

With such a broad set of application areas, a language mechanism cannot be expected to provide all the answers. But it should lend itself to adaptation to all the intended forms of concurrency. Mechanisms external to the language itself will make it possible to describe a mapping between the physically available processing units and the threads of control (called “processors” below) needed by the software text.

Support for coroutine programming

An interesting form of computation is the use of coroutines. Present in the first object-oriented language, Simula 67 [5, 17], coroutines are sequential program units which communicate on an equal basis (rather than the master-subordinate relationship of a calling unit and a routine). Between its successive reactivations, a coroutine retains the value of its data, and the location of its active program counter − as opposed to a routine, which is always restarted at the beginning. Coroutines restart each other explicitly through operations called ***resume*** in Simula.

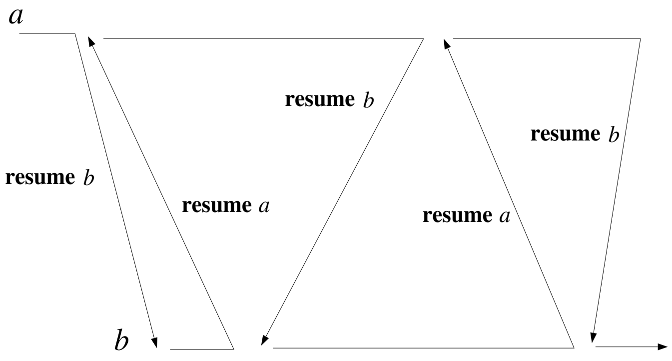

Figure 3: Coroutine computation

Coroutine computation may be viewed as an extreme form of concurrency in which only one processing unit is available. Any general-purpose concurrency mechanism should reduce to a coroutine mechanism in this case.

A particularly important consequence of this property will be to ease the transition from a simulation to the actual system. To write a computer program simulating a real-world system that involves concurrent activities (as most do), it is often convenient to use a coroutine-based scheme, such as Simula's discrete event simulation facilities. If the same language mechanism is used for coroutines and for actual concurrency, distinguished only by the number of available processing units, writing the simulation helps write the actual system, and conversely if the system has already been written it is easier to write a simulation.

Adaptability through libraries

Many concurrency mechanisms have been proposed over the years, from semaphores and conditional critical regions to Petri nets, monitors and CSP. Each has its partisans, and each may provide the best approach for a certain problem area.

It is important, then, that the proposed mechanism should support at least some of these mechanisms. More precisely, the solution must be general enough to allow us to *program* various concurrency constructs in terms of the basic mechanism that we will have obtained.

Here the facilities of the object-oriented method should again be put to good use. One of the most important aspect of the method is that it supports the construction of libraries for widely used schemes. The library construction facilities (classes, assertions, constrained and unconstrained genericity, multiple inheritance, deferred classes and others) should allow us to express many concurrency mechanisms in the form of library components.

Support for reuse of non-concurrent software

A criterion of the desirable rather than essential category is the ability to reuse existing, non-concurrent software, especially libraries of reusable software components.

This may not always be achievable, since concurrent computation places new demands on structures; for example we must specify the scope of exclusive access segments. But even when existing software cannot be reused exactly as it stands the work involved in making it applicable to concurrent development should be reasonable, involving for example the writing of simple "wrapper" classes encapsulating existing sequential classes.

This criterion may be viewed as another form of the minimality requirement, applied here not to the job of the language designer but to that of software developers.

Support for deadlock avoidance

A specific but important question is how the sought mechanism will help solve a difficult problem of concurrent programming: avoiding deadlock.

A solution which guarantees deadlock avoidance in all cases would probably be too limited. For example, the requirement to support many different forms of concurrency, stated above, suggests that with our mechanism we should be able to write a class describing semaphores. But as soon as we have semaphores we have the possibility of deadlock. A more realistic requirement, then, is that the mechanism should make it possible to avoid deadlock by observing certain statically enforceable restrictions – which must remain reasonable.

Making this possible was an important concern in the design of the mechanism described here. So far, however, the question of deadlock avoidance has not been pursued very far, and no result appears on this topic below.

4 DESIGNING A SOLUTION

Let us now examine how the above requirements determine a concurrency mechanism that remains compatible with the letter and spirit of object-oriented software construction.

No active-passive distinction

The first point to study is whether we need a special notion of "active object" and process. Most current proposals for object-oriented concurrency mechanisms include such notions. Two typical examples among many is a recent article by Caromel on a concurrent Eiffel extension [7], which begins: "*The first choice faced when designing a concurrent language is process genesis. What language construct and concept permit process definition?*", and a description of the POOL language by America[3] which, again in an introductory paragraph, states that "*each object also has a* body*, a local process that starts as soon as the object is created and runs in parallel with all the other objects in the system*". The present discussion differs from such approaches by refusing to include an explicit notion of process or of active object.

The reasoning behind proposals supporting active objects is as follows. In the usual, sequential form of object-oriented computation, objects are "sitting there" waiting for requests addressed to them. Such requests are feature calls, of the form ***t*.*f*** (...), meaning "apply the feature associated with *f* to the object attached to *t*, with the given arguments if any". (See note 1 for the definition of the basic model and terminology.)

Such a feature call is executed on behalf of a requesting object, the client, and is addressed to a target object, the supplier. In this scheme a supplier object is just a passive repository of features, ready to be triggered at the client's behest. The client is in the same position with respect to its own clients; in fact every object is used as supplier of other objects. Execution of a system is started by creating one object (the root) and applying a feature to it, firing off a whole chain of feature calls.

For concurrent computation, the argument goes, we need active objects with their own computational power – their own agenda. Such objects will correspond to the processes found in most non-object-oriented models of concurrent computation. So we will have objects of two kinds: passive objects, as in sequential object-oriented computation; and active objects. Such a distinction was in fact previewed in Simula [5, 17], where a class could, in addition to its features, include a body – a sequence of instructions describing a behavior associated with the instances of the class. This made instances of such classes the forerunners to active objects. This facility served in particular for the use of Simula classes as coroutines, and for discrete-event simulation.

On closer examination, however, the passive-active distinction appears unjustified and in fact harmful. It is useless to associate a special algorithm with an object when, through the routines of its class, it can have as many as the class author desires.

A typical example of an active object would be the process associated with a printer. The process would describe the algorithm that governs the printer's life: initialize; repeatedly process user jobs; shutdown. In object-oriented programming, however, this will just be one of the features associated with the printer. The corresponding class, using techniques of behavior encapsulation illustrated above with class *PRINTER*, could appear in sketched form as

```
class PRINTER inherit

        PROCESS
                rename over as off_line, finalize as stop end

feature

        off_line: BOOLEAN;
                -- over is effected as an attribute, under the name off_line.

        stop is
                        -- Go off-line.
                do off_line := true end;

feature {NONE}

        step is
                        -- Execute individual actions of an iteration step.
                do
                        start_job; process_job; finish_job
                end;

        setup is do ... end;

        start_job is do ... end;

        process_job is do ... end;

        finish_job is do ... end

end
```

The procedure *process_jobs* describes the process associated with the printer. Were a special notion of active object to be added, this procedure would become the body of class *PRINTER*. But why should we settle for one procedure when we can have as many as we want? By sticking to the above scheme, we retain the ability to have more features than just *process_jobs* available to clients, for example *stop*, *print_diagnostics*, *prepare_for_maintenance* and many others. This is the starting point of the object-oriented method: the realization that you can almost always do more than one thing with an object. This refusal to consider any one feature as "the main operation" on a class is one of the main tenets of the object-oriented method, and yields some of its key advantages in terms of software extendibility and reusability. By grafting onto object-oriented programming an independent concept of process, in the form of active objects, we would lose this essential property and gain nothing new.

The addition of processes as an independent concept would cause other problems as well. Limiting an active object's available scripts to just one raises the question of how active objects (processes) request services from each other. The fundamental object-oriented computation mechanism, feature call, would not work any more without some special synchronization mechanism: in the execution of $t.f$ (...), if the object T_OBJ attached to t is active, it will be busy with its own computation and not ready to handle the call unless special measures are taken. To solve the problem, we would have to add CSP- or Ada-like mechanisms, resulting in a full new language layer. The complexity of the result would be unacceptable.

Processors

If processes are not the appropriate new basic concept, we must find a better way of expressing the fundamental difference between sequential and concurrent object-oriented programming.

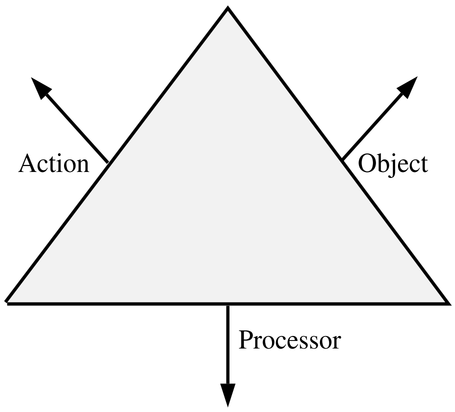

Figure 4: The basic triangle of computation

The following simple observation may serve as a basis for an answer. Computation, as described by the above figure, involves three elements: certain processors apply certain actions to certain objects.

Object-oriented programming has been quite effective at capturing the last two aspects, by attaching the description of actions (routines) to the description of objects (classes). In ordinary sequential computation, there is only one processor, which is why it tends to remain implicit.

With concurrent computation, however, we have two or more processors, and so we need to make processors explicit. This will be the major result of adding concurrency to the framework of sequential object-oriented computation.

For every object T_OBJ, there must be a processor which is responsible for executing all calls having T_OBJ as target. This processor will be said to handle T_OBJ; the handler of every object is determined when the object is created. Although it is possible to conceive of schemes in which an object may be reassigned to a different processor during the course of its life, this possibility will not be offered in the model discussed here.

A processor is a separate thread of control capable of supporting the sequential execution of operations on one or more objects. It is important to note that this notion of processor is virtual, not physical. A processor may represent a physical computational device (CPU), for example a computer on a network, but this is not necessarily the case: a processor may just as well be time-shared with other processors on a computer. For example a Unix task or a lightweight process may be used as processors. The difference between virtual processors and physical CPUs was clearly expressed by Lieberman [10]:

> *The number of* [*processors*] *need not be bounded in advance, and if there are too many* [*processors*] *for the number of real physical* [*CPUs*] *you have on your computer system, they are automatically* time-shared*. Thus the user can* pretend *that processor resources are practically infinite.*

(Lieberman's terminology and concurrency model are different from those of this article, hence the bracketed words.) To avoid any confusion, the present discussion will employ the term "processor" only in this sense of virtual thread of control; when it needs to refer to an actual computational device, it will rely on the term CPU.

Because processors are virtual, not physical, the mechanism described below may be used to support distributed processing, where the processors are physically distinct computers, as well as multiprogramming, where the processors are supported by operating system processes. An extreme case is the availability of just one CPU: then, as described in more detail below, the mechanism may be used for coroutine programming.

A process, in this scheme, becomes a trivial notion – an instance of a "process class". A process class has a distinguished procedure, the only one of interest for clients. For the common case in which the distinguished procedure is a loop describing a behavior to be repeated until termination, we can write any process class as an effective descendant of the class *PROCESS* as introduced above. There is no need for new language constructs.

Contracts and concurrency

Moving from sequential to concurrent object-oriented programming, then, will imply making the processors explicit in some way. To find the appropriate method for doing this, we must understand what having more than one processor implies for the basic scheme of the object-oriented software method: design by contract.

As pointed out above, a fully defined contract implies a no hidden clause property: clients that "play by the rules", observing the precondition of a call, are guaranteed to obtain the result, as expressed by the postcondition. Unfortunately, this crucial property will not hold in a concurrent context without a change in the semantics.

Consider again the bounded queue example, with the calls to *put* introduced above as /1/ and /2/. In either case, the condition ***not*** *q.empty* will hold prior to the call to *put*. This results from the test in the first case and from the postcondition of *remove* in the second. Another way to express this property is to state that in sequential programming assertions express correctness conditions. If, prior to the call to *put*, the condition ***not*** *q.full* is violated, this simply indicates a **bug** in the client containing this call. The only reasonable response is to correct the bug. The presence of a mechanism to check assertions at run time, as indeed exists in the Eiffel environment, does not change anything in this regard: run-time assertion monitoring is a precious aid to quality assurance, in particular testing and debugging; but it is not a technique for treating certain special but expected conditions in a particular way.

Unfortunately, these properties break down for concurrent situations. Assume that the processor handling the client is different from the processor handling the object attached to *q*. Then in /1/, as any student having taken an introductory course in concurrent computation knows, one or more processors can sneak in between the test and the call to *put* and call features such as *put* on *q* on behalf of other clients, making the test totally useless. In the same way, between the two instructions of /2/, other processors can invalidate the result (***not*** *q.full*) achieved by the call to *remove*.

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 means that the sequential contract model does not hold in its original form for concurrent computation.

Separate entities

A consequence of the previous discussion is that the sequential semantics of assertions will have to be adapted for cases in which the client and the supplier are handled by different processors.

Before we start exploring what the new semantics should be in such cases, and regardless of the eventual answer, we must address an absolute requirement: ensuring that the effect of an operation is clear from the software text.

Any concurrent semantics we choose will imply that *t.f* (...) may have a different effect depending on whether the object attached to *t* is handled by the same processor as the client, or a different one. It would be unacceptable to hide this important difference from the reader of the software text.

(***SIDEBAR 3***)

Entities, types and values

The term "entity", a generalization of the usual notion of variable, covers any name used to denote run-time values. Entities include the following: attributes of classes (see note 1); formal arguments of routines; local entities of routines (accessible only within the routine's body, and allocated anew for each execution of the routine, as with a local variable in Pascal); and *Result*, a predefined entity used in functions to denote the result to be returned to the client.

Types describe the run-time values which entities denote in the software text. There are two kinds of values: objects, and references to objects. Correspondingly, there are two kinds of types: expanded types, whose values are objects, and reference types, whose values are references to objects.

Expanded types include basic, predefined types such as *INTEGER* and *REAL*. So the value of an entity declared of type *INTEGER* is directly an integer value.

A reference type may be obtained from a class which is not declared as ***expanded***. For example with the class *BOUNDED_QUEUE* as above, the declaration

q: *BOUNDED_QUEUE* [*X*]

introduces *q* as an entity of reference type. The possible values of such an entity at run time are references: either void (not attached to any object), or attached to an object, here an instance of *BOUNDED_QUEUE*. "Attached" is the technical term which describes the association between a non-void reference and an object.

As a result, we should have a special notation to declare that a certain entity (see note 3 for the definition of this term) denotes objects that will be handled by a different processor.

The syntactic extension is immediate. Instead of the usual declaration

```
-- /4/
    x: SOME_TYPE
```

we will declare an entity as

```
-- /5/
    y: separate SOME_TYPE
```

to express that *y* may become attached to objects handled by a different processor.

With such a declaration, the creation instruction

```
!! y.make (...)
```

has an extra effect. In all cases (***separate*** declaration or not) the instruction creates a new object, initializes it to language-defined default values, and applies the creation procedure *make* with the given arguments as a way to override default initialization as needed. If *y* is declared as ***separate***, the instruction will, in addition, assign a new processor to handle the newly created object.

The semantics of the language should not specify how the new processor is determined, although it is possible to envision library mechanisms which will give programmers some control in this respect.

Another way to obtain a separate object may be through a call to some function of the form

```
new_object: separate T is ...
```

such a function may be declared as ***external***, allowing some control from outside the language proper. This makes it possible to assign objects to different threads of control (for example, to various computers on a network or multi-microprocessor system) through some external mechanism, without recompilation of the software. The details of such mechanisms fall beyond the present discussion, but it is important to make sure they are possible.

A possible objection should be examined here. One might be concerned that by declaring *t* as ***separate*** we are giving out too much implementation detail. Should we not just be able to write *t*.*f* (...) without worrying where the call is executed? But this objection is not justified. What should be hidden from the client in all but special cases is the precise knowledge of *which* processor handles a call. The mechanism described here achieves this objective. But whether the processor in question is the same one as the processor handling the client, or another, is highly relevant information, since the semantics is different. In addition to the change in the interpretation of assertions, there is an even more fundamental difference: execution by the same processor is blocking − the client cannot proceed until execution of *f* has been completed − whereas execution by a different processor should not prohibit the client from continuing its own execution until it actually needs the results of the call, if ever. (This property will give rise to the policy of lazy wait, explained below.) So it is indeed necessary to state clearly whether the processor is the same.

Certain classes are meant to be used only as types of separate entities. As a notational convenience, it will be permitted to declare such classes as

```
separate class CLASS_NAME ... the rest as usual ...
```

meaning that an entity declared of type *CLASS_NAME* will always be separate. As with the two other existing mechanisms to characterize a class, ***deferred*** and ***expanded***, the separate or non-separate status of a class is not transmitted through inheritance: a separate class may inherit from a non-separate one (the common case) and conversely. So for example we may describe bounded buffers through a class declaration consisting simply of

```
separate class BOUNDED_BUFFER [G] inherit
        BOUNDED_QUEUE [G]
end
```

The three specific properties of classes are exclusive: the syntax enables the keyword ***class*** to be preceded by at most one of ***deferred***, ***expanded*** and ***separate***.

The ***separate*** declaration for entities and classes will turn out to be the only syntactic extension needed by the mechanism described here, although we have yet to examine its full semantic consequences.

It will be convenient to use the term "separate" in various contexts. A class declared as ***separate class*** is a separate class. An entity declared as ***separate*** *SOME_TYPE*, or as *SOME_SEPARATE_TYPE* if *SOME_SEPARATE_TYPE* is based on a separate class, is a separate entity. A call $t.f$ (...) is separate if its target t is separate. An expression y is separate if it involves at least one separate entity. If t is a separate entity, and at some point of the execution of the system has a non-void value, the attached object, which is handled by a different processor, is said to be a separate object.

For consistency, a validity rule will require that in any assignment of the form $x := y$, if the source y is separate, the target x must also be separate. Otherwise we would be able to cheat by manipulating a separate object through a non-separate entity x, obtaining the wrong semantics. It will be permitted, however, to assign a non-separate value to a separate entity; calls on the corresponding object will then have the semantics of separate calls, which (as will be clear below) is harmless.

The same rule applies to actual-formal argument association, as in the call $t.f\ (y)$, where the corresponding formal argument declared for routine f is x. (The semantics of assignment is the same as that of argument passing; the term **attachment** covers both operations.)

Semantics of assertions

With the notion of separate entity in place to ensure that every call is clearly identified as intended for the same processor or for another, we need to come back to the important question of what assertions mean for separate calls.

We have seen that preconditions, or at least those clauses of a precondition that involve separate entities, cannot be taken as correctness conditions. But the requirements expressed by a precondition are still needed for the routine to do its job properly. For example, we cannot write a correct version of *put* without some guarantee that the queue is non-full on entry.

What then should the semantics of a precondition such as ***not*** *full* be if the client and supplier are handled by different processors? Only two answers seem to make sense:

- Failure to meet the precondition may mean failure of the call. As noted above, the contract model provides for such a case: a call may fail if the routine is unable to fulfil its contract; this causes an exception in the client. The client may recover from the exception through a Rescue clause which takes any needed corrective actions and, through a Retry instruction, tries another (or the same) strategy. If no Rescue clause is present, the client itself fails and passes the exception to its own client. This behavior may be called the exception semantics.

- More commonly, failing to meet the precondition may simply mean that the conditions are not ripe yet for the routine body to execute, without implying failure. The call in this case should just block the client from progressing, releasing the supplier's processor for handling requests from other client processors, which may be expected to produce effects that will make the precondition true. When the supplier's processor completes a call, it will examine the requests from blocked clients and select one for which the blocking precondition is now true. This behavior may be called the waiting semantics.

Both strategies are compatible with the contract model. Experience with the practice of concurrent programming suggests that we should retain the waiting semantics as the default. This is particularly clear in the bounded queue example: handled by a separate processor, the bounded queue becomes a bounded buffer, which various clients use to deposit and withdraw elements. Then if such a client calls *put* with the buffer full, or *item* with the buffer empty, it should normally be made to wait until another client has corrected the situation (through a *remove* in the first case and a *put* in the second).

So as a basic semantic rule a precondition clause involving a separate call should cause the client to wait until the clause becomes satisfied. (More on this below.)

There may remain a need for the exception semantics when the client expressly wants to treat precondition violation as an exceptional case. A typical example is access to a file, a case which already arises in sequential object-oriented programming, as the underlying concurrency is usually hidden. The following extract (using class *FILE* from the basic Eiffel library) would on the surface seem safe:

```
f: FILE;
...
if f /= Void and then f.readable then
        f.some_input_routine
                -- some_input_routine is any routine that reads
                -- data from the file; its precondition is readable.
end
```

Here the test in the conditional instruction is meant to guarantee that f is attached to a readable file, meeting the precondition of *some_input_routine*. But although this extract does not explicitly show the concurrency, a file is a separate persistent structure, so the client has no way to avoid the case in which an interactive user (or some other software system) will access the file and make it unreadable between the test of *f.readable* and the execution of *some_input_routine*. Here, however, waiting for *readable* to become true again is not the appropriate behavior; instead, the client should probably get an exception (from which it will be able to recover if it has a Rescue clause).

Because such cases are unlikely to be the most common, the waiting semantics will be the default. Through calls to routines of the kernel library, as described below, it will be possible for a client to request the exception semantics.

Reserving objects

Considering separate precondition clauses as wait conditions does not yet provide us with enough control, especially if a client needs to reserve an object for the duration of several operations.

In the bounded queue example, consider the client extract (again for q declared as a queue and a as a queue element):

```
-- /3/
        a := q.item;
        ... Other instructions (not calling remove) ...
        q.remove
```

The call to *item* accesses a queue element (after a wait if the queue is empty at the time of the call). The call to *remove* is intended to remove that element. But there is no guarantee that the two calls indeed manipulate the same queue element: with the mechanisms introduced so far, we have no way of preventing other separate clients from intruding between the two calls and performing one or more *remove*.

The problem arises even if the calls to *item* and *remove* are consecutive in the client text. Of course, in this case, we could solve the problem by renouncing the command-query distinction and adding a feature *get* to class *BOUNDED_QUEUE*:

```
get: X is
                -- Remove an element, and return it as result.
        require
                not empty
        do
                Result := item; remove
        end
```

but this would imply a drastic change in the recommended design style and in any case does not help us for the general case with one or more "Other instructions" between the two calls.

These observations highlight the need for a technique that will allow a client to reserve a separate supplier object for a certain period. Although routine calls similar to P and V operations on semaphores could be envisioned for that purpose, it is preferable, in the interest of deadlock prevention, to look for a linguistic construct: whereas it is practically impossible to ensure that every client that executes a P will later execute a V, a linguistic construct which reserves an object will have a fixed syntactic scope; execution of the construct will automatically release the object at the end of that scope.

Such a construct could have the following form:

```
        -- Note: uses a form not retained. For purposes of discussion only.
hold q then
        ... Here the client has exclusive access
           to the object attached to q ...
end
```

Let Q_OBJ be the object attached to *q*. The execution of such a construct would imply that if Q_OBJ is already reserved the client will wait until Q_OBJ is free again.

Unfortunately, a simple ***hold*** construct as defined above does not provide us with a flexible enough synchronization mechanism. Often a client will need to wait not just until a certain supplier becomes available, but also until a certain condition becomes true – for example ***not*** *q.empty* if the client needs to perform a *q.item* or *q.remove*. This suggests a variant of the ***hold*** construct, which includes waiting on one or more conditions, for example:

```
        -- Note: uses a form not retained. For purposes of discussion only.
hold q when not q.empty then
        a := q.item;
        ... Other instructions ...
        q.remove
end
```

Such a construct could be used as the basis of a workable solution. One advantage of such a solution would be to give assertions their original semantics back: within a ***hold***, a precondition is a correctness condition; the proof rule (left for the reader to express) would indicate that, right after the ***then***, the condition given by the ***when*** clause may be assumed.

On further examination, however, this solution is not fully satisfactory. A first problem that will have to be settled by the language designer is whether to allow any separate call (such as *q.item*) outside of a ***hold*** on the corresponding target (here *q*). The goal of consistency, always so important in language design, suggests prohibiting this. But then client code will be encumbered by many ***hold*** instructions.

Even if we decide against this strict policy, clients will have to include numerous ***hold*** to access such separate objects as bounded buffers. Any such situation – where a clearly identified scheme occurs repeatedly in a certain application area – triggers the object-oriented designer's basic instinct: encapsulate. Rather than individually wrapping every non-trivial buffer access in its own ***local*** .. ***end***, the competent object-oriented designer will write a class which encapsulates the corresponding behavior, for example:

```
        -- Note: uses a form not retained. For purposes of discussion only.
class BUFFER_ACCESS [G] feature
        put (q: separate BOUNDED_QUEUE [G]; x: G) is
                        -- Insert x into q, waiting if necessary until there is room.
                do
                        hold q when not q.full then q.put (x) end
                end;
        remove (q: separate BOUNDED_QUEUE [G]) is
                        -- Remove an element from q, waiting if necessary
                        -- until there is such an element.
                do
                        hold q when not q.empty then q.remove end
                end;
        item (q: separate BOUNDED_QUEUE [G]): G is
                ... Left to the reader ...
end
```

Any class that needs that behavior will inherit from *BUFFER_ACCESS*.

Such classes are likely to be needed for every separate data structure. But then we may ask whether the ***hold*** construct is useful at all. If every significant use of a buffer *q* occurs through a routine such as those above, with *q* as argument, could we not use argument passing as the mechanism for reserving objects? In other words, we might simply decide that whenever a routine call has a separate argument, any routine call will perform a "hold" operation on the corresponding separate object.

There remains the problem of waiting not just on object availability but also on conditions. Here the idea of preconditions as waiting conditions makes a comeback: since the routine cannot execute its body properly until its precondition becomes available, the precondition again presents itself as offering a natural wait condition. We may note here that the style that would likely have become the most common for using the ***hold*** construct, as illustrated by *BUFFER_ACCESS* above, would entail frequent duplications of precondition texts as hold conditions; this is the case above with the use of *put*, *remove* and *item*.

The solution finally retained, then, involves the following elements:

1 • No special "holding" construct is needed. ***separate*** declarations remain the only syntactical extension.

2 • No separate call, of the form $t.f$ (...) where the target t is separate, is permitted unless t is a formal argument of the routine where the call appears.

3 • If any non-void argument to a routine call is separate, the call will block until the corresponding object becomes available.

4 • If a precondition clause of a routine f involves a separate call (whose target, because of rule 2, must be a formal argument of f), a call to f will block until the precondition clause is satisfied.

5 • If neither rule 3 nor rule 4 causes a call to block, the call is said to be **satisfiable**. If there are one or more satisfiable calls on available objects handled by a given processor, one of them will proceed. (The semantics does not specify which satisfiable call will be selected if there are two or more; but it does require that one of them will proceed. In other words, a processor may not go on strike.)

6 • "Available", as used in rules 3 and 5, is defined as follows. An object is *busy* if some call using it as a target has been started but not yet completed. The object is available if it is not busy and its processor is either idle (not executing any call) or blocked (as per rule 3 or 4) on a separate call executed on behalf of another object.

Rule 2 may seem unduly restrictive. It does bring, however, a much desirable extra safety, avoiding situations such as /3/ above where the author of the client code mistakenly believes that two distinct calls using the same separate entity as targets, such as the calls $q.item$ and $q.remove$, actually apply to the same object. Without rule 2, such errors would be likely to occur. They would be hard to detect and debug since the corresponding run-time behavior, which depends on how many other calls creep in between the two calls, is non-deterministic. Such errors are typical of what makes it hard to construct and debug parallel programs, and we should not lightly forsake an opportunity to avoid them through a statically enforceable rule.

True, rule 2 will require some extra work on the part of the concurrent programmer: we cannot any more write $q.remove$ freely, but must enclose this call in a routine using q (assumed to be of type ***separate*** *BOUNDED_QUEUE* [*X*] for some *X*) as argument. But this extra requirement seems justified in light of the gain in reliability; in addition, we may expect that many uses of such structures will rely on a class encapsulating the appropriate behavior, such as *BUFFER_ACCESS* above, where the problem is taken care of once and for all. As before, classes needing these facilities will inherit from *BUFFER_ACCESS*.

The class *BUFFER_ACCESS* will now be written more simply as:

```
        -- Encapsulation of access to bounded buffers
class BUFFER_ACCESS [X] is

        put (q: separate BOUNDED_QUEUE [G]; x: G) is
                        -- Insert x into q, waiting if necessary
                        -- until there is room.
                require
                        not q.full
                do
                        q.put (x)
                ensure
                        not q.empty
                end;

        remove (q: separate BOUNDED_QUEUE [G]) is
                        -- Remove an element from q, waiting if necessary
                        -- until there is such an element.
                ... Left to the reader ...

        item (q: separate BOUNDED_QUEUE [G]): G is
                        -- Oldest element not yet consumed
                require
                        not q.empty
                do
                        Result := q.item
                ensure
                        not q.full
                end

end
```

The use of separate actual arguments as ways to reserve objects requires one further comment. If a routine call names two or more such arguments, the call will block until it has got hold of **all** of them, and satisfied the corresponding preconditions. This may prove difficult to implement, especially in a distributed system, which will require consensus between the various processors. It is possible to restrict the mechanism by allowing a routine call to involve at most one separate argument. Then it will be the individual programmer's job to reserve all necessary resources through nested calls, using one of the algorithms described in the literature; see for example [4]. It may be preferable, however, to keep the model without restrictions, following the argument that whenever possible tedious and error-prone tasks should be handled by the programming environment rather than by individual programmers. This reasoning is central in object-oriented programming, where it justifies such important facilities as garbage collection and compiler-applied static binding.

A proof rule

In the scheme retained here, waiting on a precondition clause occurs only for a precondition of the form *t.cond*, where *t* is a formal argument of the enclosing routine, and is separate. This is why the preconditions on the routines of *BUFFER_ACCESS* are needed: violation of a precondition of *BOUNDED_QUEUE* would be a correctness violation, not a wait condition, since these routines do not have any separate argument. The routines of *BUFFER_ACCESS*, however, have a separate argument *q*, so any precondition clause of the form $q.some\_condition$ in these routines is a waiting condition.

In general, then, in a routine of the form

f (..., *a*: *T*, ...) ***is***
  require
    clause1;
    clause2;
    ...
  do ... ***end***

any of the precondition clauses which does not involve any separate call on a separate formal argument is a correctness condition: any client must ensure that condition prior to any call, otherwise the call is in error. Any precondition clause involving a call of the form $a.some\_condition$, where a is a separate formal argument, is a wait condition which will cause calls to block if it is not satisfied.

These observations may be expressed as a proof rule which, for separate computation, replaces the sequential rule given on page 10:

$$\frac{\left\{ INV \wedge \bigwedge_{p \in Pre\,(f)} p \right\} \quad body\,(f) \quad \left\{ INV \wedge \bigwedge_{q \in Post\,(f)} q \right\}}{\left\{ \bigwedge_{p \in Nonsep\_pre\,(f)} t.p' \right\} \quad t.f \quad \left\{ \bigwedge_{q \in Nonsep\_post\,(f)} t.q' \right\}}$$

Here *Nonsep_pre* (*f*) is the set of clauses in *f*'s precondition which do not involve any separate calls, and similarly for *Nonsep_post* (*f*).

This rule captures in part the essence of parallel computation. To prove a routine correct, we must still prove the same conditions (those above the line) as in the sequential rule. But the consequences on the properties of a call (below the line) are different: the client has fewer properties to ensure before the call, since, as discussed above, trying to ensure the separate part of the precondition would be futile anyway; but we also obtain fewer guarantees on output. The former difference may be considered good news for the client, the latter is bad news.

The separate clauses in preconditions and postconditions thus join the invariant as properties that must be included as part of the internal proof of the routine body, but are not directly usable as properties of the call.

The rule also serves to restore the symmetry between preconditions and postconditions, a job which will be completed below by considering the lazy wait technique. The discussion so far was essentially based on an analysis of the properties of preconditions.

Comments on the use of preconditions

The idea that assertions, and in particular preconditions, may have two different semantics – sometimes correctness conditions, sometimes wait conditions – is somewhat disturbing. Yet in the design of this mechanism the idea kept coming back each time it was discarded.

One possible objection, however, is unjustified. In Eiffel, run-time assertion checking may be turned on or off 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 just as much as the *do* clause. They are part of an important property of the routine: its 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 is precisely that the **do** clauses of its routines are compatible with the 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. (This is particularly difficult to explain to C programmers, who when they accept assertions at all tend to see them just as executable constructs, meant to check certain properties at run time.)

Assertions, then, are always part of the software, whether or not they are monitored at run-time for debugging purposes. The difference between their separate and non-separate clauses is simply that the semantics does not require a non-separate clause to be evaluated at run time if the software is correct, but does require the run-time checking of separate clauses.

Lazy wait

One more rule is needed for a full definition of the semantics. This rule will determine how two processors are resynchronized when one needs a result from another.

The rule, which may be called lazy wait by analogy with the so-called lazy evaluation of programs written in functional languages such as Lisp, requires a client to wait when it absolutely needs information resulting from a call, but no sooner.

Consider a routine containing a separate call:

```
r (..., t: separate SOME_TYPE, ...) is
        do
                ...; t.f (...); --*
                other_instructions
        end
```

Assume r is executed on an object C_OBJ and, as part of its execution, executes the call marked $*$, with target t. Let T_OBJ be the object attached to t. The above rules indicate that t must be a formal argument of r; so C_OBJ will have obtained hold of T_OBJ prior to the call.

For the above syntax to be legal, f must be a procedure. The call to f proceeds when it is satisfiable and selected by the processor of T_OBJ, which then starts executing it. But C_OBJ is handled by another processor, and that processor does not need to wait for the call to terminate: it should immediately continue with its further business, *other_instructions*. This will be the rule according to the lazy wait policy: separate procedure calls should not (except for one special case discussed below) cause the client to wait.

Assume the *other_instructions* contain more procedure calls on the same separate target T_OBJ. These calls, like the first one, will not make the client wait. It is essential for consistency, however, that the order in which they will eventually get serviced be the order in which the client, here C_OBJ, has requested them. The semantic rule must guarantee this property of order preservation.

When then, if ever, must C_OBJ wait for the termination of the call to f? The lazy answer is that an object will wait when it needs the result of a query (function or attribute call) on a separate object. Assume for example that the *other_instructions* above are of the form

other_instruction_1; ... *other_instruction_n*;
k := ***t*****.***some_value*

where *some_value* is an integer attribute or function applicable to t, and k is an auxiliary entity, also of integer type. To assign the value of the last call to k we need access to T_OBJ; so the call to f must have been completed, as well as any intervening call with T_OBJ as its target, and if *some_value* is a function we must also wait for the computation of that function to complete. (Remember that the entire fragment appears in a routine of which t is a formal argument, so there is no danger of another client sneaking in and grabbing T_OBJ between the call to f and the call to *some_value*: C_OBJ has exclusive separate access to T_OBJ throughout the execution of this fragment.)

The lazy wait policy requires that a client which performs a query on a separate target shall not proceed until all earlier calls have been completed in the order logged, and the query itself has been executed to completion.

Together with the rule that a call proceeds when it is satisfiable, this property is the principal semantic difference between concurrent and sequential object-oriented computation. (It would need to be included in the semantics of calls defined in chapter 21 of [14].) The lazy policy may be seen as a form of waiting on the postcondition, in the same sense that the basic mechanism causes waiting on the precondition.

A note is in order on the rule that clients do not wait for separate procedure calls to terminate. At first this may be viewed as just an optimization: forcing clients to wait for the termination of such calls before proceeding would simply make less use of the available parallelism. But such a policy would be misguided. If the *other_instructions* of the above pattern are empty, as in

r (..., *t*: ***separate*** *SOME_TYPE*, ...) ***is***
    do ... *t***.***f* (...) ***end***

the execution of r will terminate immediately after launching the call to f, without waiting for that call to terminate. This may cause the caller of r to relinquish the reservation it made on the client of T_OBJ, which may be crucial if that object is needed by other clients.

A common variant of this scheme, which would not work without the ability to continue execution after starting a separate procedure call, is a loop which gets a number of previously created objects started with their own lives. For example:

```
launch (a: ARRAY [separate X]) is
        -- Get every element of a started.
    require
        -- No element of a is void
    local
        i: INTEGER
    do
        from i := 1 until i > a.count loop
            launch_one (a @ i); i := i + 1
        end
    end

launch_one (p: separate X) is
        -- Get p started.
    require
        p /= Void
    do p.live end
```

(The notation *a* @ *i* denotes the *i*-th element of *a*. @ is an infix feature of the Kernel Library class *ARRAY*. The array must have been previously initialized to hold non-void elements, numbered from 1 to *a.count*.)

If, as may well be the case, procedure *live* of class *X* describes an infinite process, this scheme only works if each loop iteration proceeds immediately after starting *launch_one*; otherwise the loop would never get beyond its first iteration.

Passing non-separate references

One technical point remains to be clarified regarding the passing of arguments. In some cases we may wish, in a class *B*, to include a separate call of the form

```
t.f (a)
```

where the actual argument *a* is of a reference type, but not separate. The corresponding routine, in the supplier class *C*, could be of the form

```
f (x: SOME_TYPE) is
    do ... y := x ... end
```

where *y* is some attribute of *C*.

All the entities involved – *a*, *x*, *y* – denote references to objects. The assignment enables *C* to retain a reference to an object A_OBJ handled by the client processor, so that later on, after the termination of *f*, a routine of *C* may execute a call of the form

```
y.some_routine (...)
```

The object to which this call applies is A_OBJ; but the call should be separate since the processors handling the instances of *B* and *C* are different! Unless *x*, and hence *y* too, are declared as ***separate***, the call will have the wrong semantics.

A language rule will require, then, that any such formal argument should be declared as separate. In the example, *SOME_TYPE* must be a separate type. The rule on assignments seen above then requires *y* also to be declared as separate.

The case examined here has a consequence on the semantics of calls: because the body of *f* may perform a call with target A_OBJ, we cannot let the client proceed even if *f* is a procedure. The call to *f* should block the client until *f* terminates. This will be the general semantic rule if we allow separate calls having non-separate references as actual arguments: any such call will block the client with no possibility of lazy wait.

This rule introduces a small but unpleasant complication. We could avoid it by disallowing non-separate reference arguments altogether in separate calls. An example below will show, however, that it is convenient (although not strictly indispensable) to keep this possibility open.

The preceding discussion only applies to reference types. If *a* is of an expanded type, its possible values are objects, and attachment (in particular argument passing) implies copying an object onto another rather than assigning a reference, so the above problem does not arise. This possibility immediately suggests, however, a new validity constraint: no expanded type may include separate attributes.

Duels

The above discussion concludes the presentation of the basic semantic model. In practice, we need an extra degree of flexibility to have a fully usable mechanism.

The major remaining problem is how a client can avoid waiting on a non-available separate supplier, or one that does not satisfy a separate precondition clause. As noted above in the example of reading a file that may suddenly have become unreadable, the appropriate policy in such a case may be to cause an exception in the client, not to make it wait.

How best should we provide such flexibility? The situation is comparable to the problem of exerting fine control on exception handling in sequential object-oriented computation. The basic exception mechanism is extremely simple, providing a Rescue clause enabling a routine to catch exceptions, and a Retry instruction enabling the routine to try again after attempting to correct the cause of the exception. These facilities suffice in most cases. To obtain further facilities for more sophisticated cases, without complicating the language or changing the basic semantics of exception handling, programmers may use the class *EXCEPTIONS* from the Kernel Library, offering such features as the integer code of the last exception (so that a Rescue clause may discriminate between different causes of exception) and many others. Classes needing these facilities simply inherit from *EXCEPTIONS*.

Here we may use a comparable approach by defining a simple library class *CONCURRENCY*, from which classes may freely inherit. This class will include a procedure *immediate_service*. A call to this procedure alters the normal response to a non-satisfiable call: instead of waiting if the object is not available or a separate condition is not satisfied, it will cause an exception in the client. Procedure *normal_service* will serve to restore standard behavior.

This can be used immediately in the file reading example, assuming *FILE* now becomes a separate class:

```
analyze (f: FILE) is
    do
        immediate_service;
        if f.readable then
            ... Various read operations of form f.input_routine ...
        else
            ... Report impossibility in some way ...
        end;
        normal_service
    rescue
        ... Take corrective action, or report problem ...
    end
```

In the contract method of software construction, exceptions should be reserved for truly exceptional cases which would be hard or impossible to handle otherwise without adding great complication to the text of the software. (This is in marked contrast with, for example, the Ada or CLU mechanisms which essentially use exceptions as inter-procedural control structures.) The above use satisfies this requirement since it applies to a case that is unlikely to occur often – the case in which a file was checked to be readable but becomes unreadable or inexistent before its actual use because of the actions of some independent agent. Although infrequent, this case must still be handled properly.

A similar mechanism may be used to provide a safe form of express messages. Assume we want to allow a VIP client, the "challenger", immediate access to a separate object; if any other client, the "holder", currently holds it, it will be interrupted. The only acceptable solution in this case, in light of the Design by Contract principles, is to make the holder's call fail, so that the holder will be forced to take corrective action (if it has a Rescue clause) or to fail. Because it has such drastic consequences for the holder, however, this behavior should only occur if the holder has accepted the possibility by calling procedure *yield* of class *CONCURRENCY*. In this case, a challenger that has called *immediate_service* and finds the desired object non-available will not get an exception itself, but will actually get the object, and cause an exception in the holder. (The immediate recipient of the exception is the common supplier, which after executing its Rescue clause to restore its invariant will normally pass on the exception to the holder.) To restore the normal (no-yield) behavior, the holder will call procedure *insist*.

The result of the potential conflict between a holder and a challenger, which we may call a **duel**, is specified in all cases by the following table. The default behavior for each duelist, as well as the overall default in the absence of any call to the procedures of class *CONCURRENCY* (top-left table entry), are underlined.

Challenger → **↓ Holder ↓**	*normal_service*	*immediate_service*
insist	Challenger waits	Exception in challenger
yield	Challenger waits	Exception in both.

Figure 5: The semantics of duels

5 EXAMPLES

To illustrate the mechanism, here now are a few examples which have been chosen from diverse backgrounds.

The dining philosophers

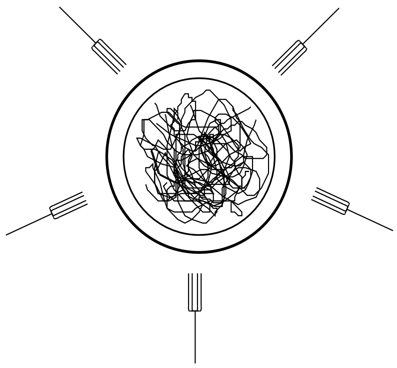

Figure 6: Dining philosophers

We begin with the inevitable dining philosophers. It seems hard to provide a simpler form for describing the philosophers' behavior than the following class:

```
separate class PHILOSOPHER creation

        make

inherit

        PROCESS
                rename setup as getup end
```

```
feature {BUTLER}
        step is
                        -- Think and eat
                do
                        think; eat (left, right)
                end

feature {NONE}
                        -- The two required forks:
        left, right: separate FORK;
        getup is do ... end;
        think is do ... end;
        eat (l, r: separate FORK) is
                        -- Eat, having grabbed l and r.
                do ... end
feature {NONE} -- Creation
        make (l, r: separate FORK) is
                        -- Assign l and r as the required forks.
                do
                        left := l; right := r
                end
end
```

The entire synchronization requirement is embodied by the call to *eat*, which uses arguments *left* and *right* representing the two necessary forks, thus reserving these objects. It would seem hard to have a simpler solution to the dining philosophers problem.

Of course that simplicity comes from the mechanism's ability to reserve several resources through a single call having two or more separate arguments, here *left* and *right*. If we restricted the separate arguments to at most one per call, the solution would use one of the published algorithms (see for example [4]) for getting hold of two forks one after the other without causing deadlock.

Thanks to the use of multiple object reservation through arguments, the solution described here does not produce deadlock, but it is not guaranteed to be fair; some of the philosophers can conspire to starve the others. Here too the literature provides various solutions, which may be integrated into the above scheme.

The notation ***feature*** {*BUTLER*} indicates that the features which follow are exported only to class *BUTLER*, the controller class, given below, and its descendants if any. The notation ***feature*** {*NONE*}, already encountered in earlier examples, similarly makes features available to *NONE* only. *NONE*, a Kernel Library class, has no descendants, so this implies making the features secret. Most of the features appearing in the examples below will be secret or selectively exported in this way. Such a situation is much less common in sequential object-oriented programming, but the difference of style is no accident: in concurrent object-oriented programming, many classes are the equivalent of processes in ordinary concurrent programming, which need few or no exported features.

Here now are the missing classes. Forks have no immediately relevant properties:

```
class FORK end
```

A butler is used to set up and start a session:

```
class BUTLER creation
    make
feature
            -- The number both of philosophers and of forks:
    count: INTEGER;
    launch is
                -- Start a full session.
        local
            i: INTEGER
        do
            from
                i := 1
            until
                i > count
            loop
                launch_one (participants @ i);
                i := i + 1
            end
        end
feature {NONE}
    launch_one (p: PHILOSOPHER) is
                -- Let one philosopher start his actual life.
        do
            p.live
        end;
    participants: ARRAY [PHILOSOPHER];
    cutlery: ARRAY [FORK]
```

```
feature {NONE} -- Creation
    make (n: INTEGER) is
            -- Initialize a session with n philosophers.
        require
            n >= 0
        do
            count := n;
            !! participants.make (1, count); !! cutlery.make (1, count);
            make_philosophers
        ensure
            count = n
        end;

    make_philosophers is
            -- Set up philosophers.
        local
            i: INTEGER; p: PHILOSOPHER; left, right: FORK;
        do
            from
                i := 1
            until
                i > count
            loop
                left := cutlery @ i;
                i := i + 1;
                if i > count then i := 1 end;
                right := cutlery @ i;
                !! p.make (left, right);
                participants.put (p, i)
            end
        end
invariant
    count >= 0; participants.count = count; cutlery.count = count
end
```

(The creation procedure *make* for arrays, used by the procedure *make* of *BUTLER* on targets *cutlery* and *participants*, allocates an array by using as bounds the arguments supplied. Also, the last instruction of *make_philosophers* uses the procedure *put* for arrays: if *a* is an array, *a*.*put* (*v*, *i*) sets to *v* the value of the element of index *i* in *a*.)

Note how *launch* and *launch_one*, using a pattern discussed in the presentation of lazy wait, rely on the property that the call *p*.*live* will not cause waiting, allowing the loop to proceed to the next philosopher.

(***NOTE 4***)
Reminder on the creation mechanism.
The general form of a creation instruction is

!! *t*.*make* (...)

which creates a new object, attaches reference *t* to that object, initializes its fields to predefined values determined from the corresponding attribute types (0 for integers and reals, null for characters, false for booleans, and void references for reference types), and applies procedure *make* to the object. This form assumes that the base class of *t*'s type lists *make* as one of its creation procedures at the beginning of the class text:

```
class C creation
    make, ...
```

(The name *make* is recommended for the most common creation procedure of a class, but is of course not required.) Alternatively, if the ***creation*** clause is absent, the form of the creation instruction is just

!! *t*

In both cases *t* is said to be the target of the creation instruction.
It is common, as in class *PHILOSOPHER* and other separate classes used in this article, to have creation procedures such as *make* appear in a clause of the form ***feature*** {*NONE*}. This means that it is not possible for a client to call *make* on a created object, as in *p*.*make* (*f1, f2*). But because the procedure is listed in the ***creation*** clause it is available for creation calls, as in !! *p*.*make* (*f1, f2*).

Making full use of hardware parallelism

The following example, although somewhat academic, illustrates how lazy wait can be used to draw the maximum benefit from any available hardware parallelism.

Consider the following extract (not involving concurrency) from a class describing binary trees:

```
class BINARY_TREE [G] feature
      left, right: BINARY_TREE [G];
      ... Other features ...
      nodes: INTEGER is
                  -- Number of nodes in this tree
            do
                  Result := node_count (left) + node_count (right) + 1
            end
```

```
feature {NONE}
    node_count (b: BINARY_TREE [G]): INTEGER is
            -- Number of nodes in b
        do
            if b /= Void then
                Result := b.nodes
            end
        end
end
```

Function *nodes* makes a standard use of recursion to compute the number of nodes in a tree. The recursion is indirect, through *node_count*.

In a concurrent environment offering many processors, it is tempting to study how we could offload all the separate node computations to different processors. Declaring the class as ***separate***, replacing *nodes* by an attribute and introducing procedures does the job:

```
separate class BINARY_TREE [G] feature
    left, right: BINARY_TREE [G];
    ... Other features ...
    nodes: INTEGER;
    update_nodes is
            -- Update nodes to reflect the number of nodes in this tree.
        do
            nodes := 1;
            compute_nodes (left); compute_nodes (right);
            adjust_nodes (left); adjust_nodes (right)
        end
feature {NONE}
    compute_nodes (b: BINARY_TREE [G]) is
            -- Update information about the number of nodes in b.
        do
            if b /= Void then
                b.update_nodes
            end
        end;
```

```
        adjust_nodes (b: BINARY_TREE [G]) is
                    -- Adjust number of nodes from those in b.
              do
                    if b /= Void then
                          nodes := nodes + b.nodes
                    end
              end
end
```

The recursive calls to *compute_nodes* will now be started in parallel. The addition operations wait for these two parallel computations to complete.

If an unbounded number of CPUs (physical processors) are available, this solution seems to make the optimal possible use of the hardware parallelism. If there are fewer CPUs than nodes in the tree, the speedup over sequential computation will depend on how well the implementation allocates CPUs to the (virtual) processors.

The presence of two tests for vacuity of *b* may appear unpleasant. It results, however, from the need to separate the parallelizable part – the procedure calls, launched concurrently on *left* and *right* – from the additions, which by nature must wait for their operands to become ready.

Coroutines

One of the requirements stated at the beginning of the article was that the mechanism should support coroutine programming. Here are two classes which achieve this goal.

```
separate class COROUTINE creation
        make
feature {COROUTINE}
        resume (i: INTEGER) is
                    -- Wake up coroutine of identifier i and go to sleep.
              do
                    actual_resume (i, controller)
              end
feature {NONE}
        controller: COROUTINE_CONTROLLER;
        identifier: INTEGER;
        actual_resume (i: INTEGER; c: COROUTINE_CONTROLLER) is
                    -- Wake up coroutine of identifier i and go to sleep.
                    -- (Actual work of resume).
              do
                    c.set_next (i);
                    request (c)
              end;
```

```
        request (c: COROUTINE_CONTROLLER) is
                -- Request eventual re-awakening by c.
            require
                c.is_next (identifier)
            do
                -- No action necessary
            end

feature {NONE} -- Creation

        make (i: INTEGER; c: COROUTINE_CONTROLLER) is
                -- Assign i as identifier and c as controller.
            do
                identifier := i;
                controller := c
            end
end -- class COROUTINE

separate class COROUTINE_CONTROLLER feature {NONE}

        next: INTEGER

feature {COROUTINE}

        set_next (i: INTEGER) is
                -- Select i as the identifier of the next coroutine to be awakened.
            do
                next := i
            end;

        is_next (i: INTEGER): BOOLEAN is
                -- Is i the index of the next coroutine to be awakened?
            do
                Result := (next = i)
            end
end -- class COROUTINE_CONTROLLER
```

One or more coroutines will share one coroutine controller. (The coroutine controller will be created through a "once" function, not shown above; once functions are the Eiffel mechanism used to permit sharing of information without falling into the dangers of global variables. See [11] and [14].) Each coroutine has an integer identifier. To resume a coroutine of identifier i, procedure *resume* will, through *actual_resume*, set the *next* attribute of the controller to i, and then block, waiting on the precondition $next = j$, where j is the coroutine's own identifier. This ensures the desired behavior.

This solution looks like normal concurrent software, but it is organized in such a way that (if all coroutines have different identifiers) at most one coroutine may proceed at any one time, so it is useless to allocate more than one physical CPU for them. (The controller could actually make use of a separate CPU, but its actions are so simple that this would not change much.)

The recourse to integer identifiers is necessary since giving *resume* an argument of type *COROUTINE*, a separate type, would cause deadlock. Using Eiffel's "unique" declaration (remotely similar to Pascal's enumerated types) programmers do not need to worry about assigning such values manually. This use of integers also has an interesting consequence: if we allow two or more coroutines to have the same identifier, then with a single CPU we obtain a non-deterministic mechanism: a call to *resume* (i) will cause the restarting of any coroutine whose identifier has value i. With more than one CPU a call *resume* (i) will allow all coroutines of identifier i to proceed in parallel.

So the above scheme, which for a single CPU provides a coroutine mechanism, doubles up in the case of several CPU as a mechanism for controlling the maximum number of processes of a certain type which may be simultaneously active.

Locks and semaphores

Assume we want to allow a number of clients (the "lockers") to obtain exclusive access to certain resources (the "lockables") without having to enclose the exclusive access sections in routines.

Here is a solution:

```
class LOCKER feature
    grab (resource: separate LOCKABLE) is
            -- Request exclusive access to resource.
        require
            not resource.locked
        do
            resource.set_holder (Current)
        end

    release (resource: separate LOCKABLE) is
        require
            resource.is_held (Current)
        do
            resource.release
        end
end

class LOCKABLE feature {LOCKER}
    set_holder (l: separate LOCKER) is
            -- Designate l as holder.
        require
            l /= Void
        do
            holder := l
        ensure
            locked
        end;

    locked: BOOLEAN is
            -- Is resource reserved by a locker?
        do
            Result := (holder /= Void)
        end;
```

```
        is_held (l: separate LOCKER): BOOLEAN is
                        -- Is resource reserved by l?
                do
                        Result := (holder = l)
                end;

        release is
                        -- Release from current holder.
                do
                        holder := Void
                ensure
                        not locked
                end

feature {NONE}

        holder: separate LOCKER

end
```

Any class describing resources will inherit from *LOCKABLE*. The proper functioning of the mechanism assumes that every locker performs sequences of *grab* and *release* operations, in this order. Other behavior will usually result in deadlock. We can once again rely on the power of object-oriented computation to enforce the required protocol; rather than trusting every locker to behave, we may require lockers to go through procedure *use* in descendants of the following class:

```
deferred class LOCKING feature

        resource: separate LOCKABLE;

        use is
                        -- Make disciplined use of resource.
                require
                        resource /= Void
                do
                        from
                                !! lock; setup
                        until
                                over
                        loop
                                lock.grab (resource);
                                exclusive_actions;
                                lock.release (resource)
                        end;

                        finalize
                end;
```

```
        set_resource (r: separate LOCKABLE) is
                    -- Select r as resource for use.
            require
                    r /= Void
            do
                    resource := r
            ensure
                    resource /= Void
            end
feature {NONE}
        lock: LOCKER;

        exclusive_actions
                    -- Operations executed while resource is under exclusive access
            deferred
            end;

        setup
                    -- Initial action; by default: do nothing.
            do
            end;

        over: BOOLEAN is
                    -- Is locking behavior finished?
            deferred
            end;

        finalize
                    -- Final action; by default: do nothing.
            do
            end
end
```

An effective descendant of *LOCKING* will effect (provide non-deferred implementations of) *exclusive_actions* and *over*, and may redefine *setup* and *finalize*.

Whether or not we go through class *LOCKING*, a *grab* does not reserve the corresponding lockable for all competing clients: it only excludes other lockers observing the protocol. To exclude any possible client from accessing a resource, you must enclose the operations accessing the resource in a routine to which you pass it as argument.

The reader may have noted that in routine *grab* of class *LOCKER* the separate call to *set_holder* passes *Current* as argument. *Current*, a reference to the current object (the call's client) is a non-separate reference. Accordingly, as discussed above, the corresponding formal argument of *set_holder* is declared as ***separate***. Without the possibility of passing non-separate references as arguments to separate calls, we would need to rely, as with the above coroutine example, on a scheme associating an integer with every instance of class *LOCKER*.

Based on the pattern provided by the above classes, it is not difficult to write classes implementing semaphores under their various forms.

An elevator control system

The following example shows a case where object-orientation and the mechanism defined in this article can be used to achieve a pleasantly decentralized event-driven architecture.

The example describes software for an elevator control system, with several elevators serving many floors. The design below is somewhat fanatically object-oriented in the sense that every significant type of component in the physical system – for example the notion of individual button in an elevator cabin, marked with a floor number – has an associated separate class, so that each corresponding object such as a button has its own virtual thread of control (processor). The benefit is that the system is entirely event-driven; in particular, it does not need to include any loop for examining repeatedly the status of objects, for example whether any button has been pressed.

The class texts below are only sketched, but provide a good idea of what a complete solution would be. In most cases the creation procedures have not been included; the reader may fill in the required initializations.

It is convenient to start with class *MOTOR* although it is straightforward and not the most important. This class describes the motor associated with one elevator cabin, and the interface with the mechanical hardware:

```
separate class MOTOR feature {ELEVATOR}
        move (floor: INTEGER) is
                        -- Go to floor; once there, report.
                do
                        "Direct the physical device to move to floor";
                        signal_stopped (cabin)
                end;
        signal_stopped (e: ELEVATOR) is
                        -- Report that elevator stopped on level e.
                do
                        e.record_stop (position)
                end
feature {NONE}
        cabin: ELEVATOR;
        position: INTEGER is
                        -- Current floor level
                do
                        Result := "The current floor level, read from physical sensors"
                end
end
```

The creation procedure of this class must associate an elevator, *cabin*, with every motor. Class *ELEVATOR* includes the reverse information through attribute *puller*, indicating the motor pulling the current elevator.

The reason for making an elevator and its motor separate objects is to reduce the grain of locking: once an elevator has sent a *move* request to its motor, it is free again, thanks to the lazy wait policy, to accept requests from buttons either inside or outside the cabin. It will resynchronize with its motor upon receipt of a call to procedure *record_stop*, through *signal_stopped*. The actual time during which an instance of *ELEVATOR* will be reserved by a call from either a *MOTOR* or *BUTTON* object is very short.

```
separate class ELEVATOR creation

	make

feature {BUTTON}

	accept (floor: INTEGER) is
				-- Record and process a request to go to floor.
		do
				record (floor);
				if not moving then
					process_request
				end
		end;

feature {MOTOR}

	record_stop (floor: INTEGER) is
				-- Record information that elevator has stopped on floor.
		do
				moving := false; position := floor;
				process_request
		end;

feature {DISPATCHER}

	position: INTEGER; moving: BOOLEAN

feature {NONE}

	puller: MOTOR;

	pending: QUEUE [INTEGER];
				-- The queue of pending requests
				-- (each identified by the number of the destination floor)

	record (floor: INTEGER) is
				-- Record request to go to floor.
		do
				"Algorithm to insert request for floor into pending"
		end;
```

```
        process_request is
                    -- Handle next pending request, if any.
              local
                    floor: INTEGER
              do
                    if not pending.empty then
                          floor := pending.item;
                          actual_process (puller, floor);
                          pending.remove
                    end
              end;

        actual_process (m: separate MOTOR; floor: INTEGER) is
                    -- Direct m to go to floor.
              do
                    moving := true; m.move (floor)
              end

end
```

There are two kinds of button: floor buttons, which passengers press to call the elevator to a certain floor, and cabin buttons which are inside a cabin and are pressed by passengers to request a move to a certain floor. There is of course a difference between the corresponding kinds of request: a request from a cabin button is directed to the cabin to which the button belongs, whereas a floor button request may be handled by any elevator. A request of the latter kind will be sent to a dispatcher object, which will poll the various elevators and select the one which can best handle the request. (The precise selection algorithm is left unimplemented below since it is irrelevant to this discussion; the same applies to the algorithm used by elevators to manage their *pending* queue of requests in class *ELEVATOR* above.)

Class *FLOOR_BUTTON* assumes that there is only one button on each floor. It is not difficult to update the design to support two buttons, one for up requests and the other for down requests.

It is convenient although not essential to have a common parent *BUTTON* for the classes representing the two kinds of button. Remember that the features exported by *ELEVATOR* to *BUTTON* are, through the standard rules of selective information hiding, also exported to the two descendants of this class.

```
separate class BUTTON feature

        target: INTEGER;

end

separate class CABIN_BUTTON inherit BUTTON feature

        target: INTEGER;

        cabin: ELEVATOR;
```

```
        request is
                -- Send to enclosing elevator a request to stop on level target.
            do
                actual_request (cabin)
            end;
        actual_request (e: ELEVATOR) is
                -- Get hold of e and send a request to stop on level target.
            do
                e.accept (target)
            end
end

separate class FLOOR_BUTTON inherit BUTTON feature
        controller: DISPATCHER;
        request is
                -- Send to dispatcher a request to stop on level target.
            do
                actual_request (controller)
            end
        actual_request (d: DISPATCHER) is
                -- Send to d a request to stop on level target.
            do
                d.accept (target)
            end
end
```

The question of switching button lights on and off has been ignored. It is not hard to add calls to routines which will take care of this.

Here finally is class *DISPATCHER*. To refine the algorithm that selects an elevator in procedure *accept*, it will need to access the attributes *position* and *moving* of class *ELEVATOR*, which in the full system should be complemented by a boolean attribute *going_up*. Such accesses will not cause any problem as the above design ensures that *ELEVATOR* objects never get reserved for a long time.

```
separate class DISPATCHER creation
    make
feature {FLOOR_BUTTON}
    accept (floor: INTEGER) is
            -- Handle a request to send an elevator to floor.
        local
            index: INTEGER; chosen: ELEVATOR;
        do
            "Algorithm to determine the elevator
             which should handle the request for floor";
            index := "The index of the chosen elevator";
            chosen := elevators @ index;
            send_request (chosen, floor)
        end
feature {NONE}
    send_request (e: ELEVATOR; floor: INTEGER) is
            -- Send to e a request to go to floor.
        do
            e.accept (floor)
        end;
    elevators: ARRAY [ELEVATOR];
feature {NONE} -- Creation
    make is
            -- Set up the array of elevators.
        do
            "Initialize array elevators"
        end;
end
```

A watchdog mechanism

Together with the previous one, the last example helps show the applicability of the mechanism to real-time problems. It also illustrates the concept of duel introduced above.

We want to enable a class to perform a call to a certain procedure *action*, with the provision that the call will be interrupted, and a boolean attribute *failed* set to true, if the procedure has not completed its execution after *t* seconds. The only basic timing mechanism available is a procedure *wait* (*t*), which will execute for *t* seconds.

Here is the solution, using a duel. A class that needs the mechanism should inherit from class *TIMED* and provide an effective version of the procedure *action* which, in *TIMED*, is deferred. To let *action* execute for at most *t* seconds, it suffices to call *timed_action* (*t*). This procedure sets up a watchdog (an instance of class *WATCHDOG*), which executes *wait* (*t*) and then interrupts its client. If, however, *action* has been completed in the meantime, it is the client that interrupts the watchdog.

```
deferred class TIMED inherit
    CONCURRENCY
feature {NONE}
    failed: BOOLEAN; alarm: WATCHDOG;
    timed_action (t: REAL) is
            -- Execute action, but interrupt after t seconds if not complete.
            -- If interrupted before completion, set failed to true.
        do
            set_alarm (t); action; unset_alarm (t); failed := false
        rescue
            failed := true
        end;
    set_alarm (t: REAL) is
            -- Set alarm to interrupt Current after t seconds.
        do
                -- Create alarm if necessary:
            if alarm = Void then !! alarm end;
            yield; actual_set (alarm, t)
        end;
    unset_alarm (t: REAL) is
            -- Remove the last alarm set.
        do
            insist;
            immediate_service; actual_unset (alarm); normal_service
        end;
    action is
            -- The action to be performed under watchdog control
        deferred
        end
feature {NONE} -- Actual access to watchdog
    actual_set (a: WATCHDOG; t: INTEGER) is
            -- Request a to set an alarm for t seconds.
        require
            a /= Void
        do
            a.set (Current, t)
        end;
```

```
    actual_unset (a: WATCHDOG; t: INTEGER) is
            -- Request a to remove last alarm set.
        require
            a /= Void
        do
            a.unset
        end

feature {WATCHDOG} -- The interrupting operation

    stop is
            -- Empty action to let watchdog interrupt a call to timed_action.
        do
            -- Nothing
        end

end

separate class WATCHDOG feature {TIMED}

        set (caller: separate TIMED; t: REAL) is
                -- Set an alarm after t seconds for client caller.
            do
                yield;
                wait (t);
                immediate_service; caller.stop; normal_service
            rescue
                terminated_before_time := true
            end;

        unset is
                -- Remove alarm.
            do
                terminated_before_time := false
            end

feature {NONE}

        terminated_before_time: BOOLEAN;

end
```

In class *WATCHDOG*, the boolean attribute *terminated_before_time* and the Rescue clause are not actually needed; they have been added for clarity. In a more robust version, the Rescue clauses should test for the type of exception, so as to use the above scheme only for exceptions caused by a call from a client having requested *immediate_service*.

6 THE MECHANISM

Here now is the precise description of the mechanism that results from the preceding discussion. The description consists of three parts: syntax; validity rules (static semantic constraints); semantics. It must be understood as an extension to the Eiffel language specification as given in [14].

Syntax

The syntactic extension involves just one new keyword, ***separate***. (In preparation for this extension, the list of keywords in appendix G of [14] already included ***separate***.) A declaration of an entity or function, which normally appears as *x*: *TYPE*, may now also be of the form

-- /6/
    x: ***separate*** *TYPE*

In addition, a class declaration, which normally begins with one of ***class*** *C*, ***deferred class*** *C* or ***expanded class*** *C*, may now be of the form:

-- /7/
    separate class *C* ...

In this case *C* will be called a "separate class". The syntactic convention indicates that separate status for a class is incompatible both with expanded status and with deferred status. As in the case of expanded and deferred classes, the property of being separate is not inherited: a class is separate or not according to its own declaration only, regardless of the separateness status of its parents.

An entity or function *x* will be said to be separate if either it is declared under form /6/ above, or its type is based on a separate class (declared under form /7/). It is not an error for both of these to apply: in form /6/, *TYPE* may be based on a separate class, although in this case the use of ***separate*** in the declaration of *x* is redundant.

Constraints

A number of validity rules apply to constructs involving separate entities.

The first rule applies to a declaration of the form /6/ above:

> In a type of the form ***separate*** *TYPE*, the base class of *TYPE* must be neither deferred nor expanded.

The next rule governs the combination of separate and non-separate elements in an attachment. The term "attachment" covers both assignment and argument passing, which have the same semantics. An attachment of *y* to *x* is either an assignment of the form *x* := *y*, or an argument passing in a call of the form *f* (..., *y*, ...) or *t*.*f* (..., *y*, ...), where the formal argument of the routine *f*, at the position corresponding to *y*, is *x*.

Here is the attachment rule:

In an attachment of *y* to *x*, if the source *y* is separate, the target *x* must also be separate.

It is permitted to redefine an entity from separate to non-separate and conversely (with the corresponding constraints on polymorphic attachment, not detailed here).

The next rules, explained earlier, apply to separate calls (calls whose targets are separate).

For a separate call to be valid, the call must appear in a routine and its target must be a formal argument of the enclosing routine.

If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.

If an actual argument of a separate call is of an expanded type, the corresponding base class may not have any attribute of reference type.

The last constraint removes any possibility that the very process of evaluating a precondition clause (as part of an access to a separate object), or more generally an assertion, could lead to further blocking:

If an actual argument *a* to a function call is separate and the call is part of an assertion, then the call must be part of a routine and *a* must be a formal argument of that routine.

This rule (which implies that a function call appearing in a class invariant may not use a separate entity as actual argument) was not introduced in the earlier discussion. Without it, we could have a routine of the form

```
f (x: SOME_TYPE) is
        require
                some_property (separate_attribute)
        do ... end
```

where *separate_attribute* is a separate attribute of the enclosing class. But then the evaluation of *f*'s precondition, either as part of assertion monitoring for correctness, or as a synchronization condition if the actual argument corresponding to *x* in a call is itself separate, could cause blocking if the attached object is not available! Such behavior is clearly unacceptable.

This concludes the list of validity constraints. The semantics will be discussed in three parts: creation; object states; calls.

Semantics of creation

If the target *t* of a creation instruction (which appears after the second exclamation mark in, for example, !! *t*.*make* (...)) is non-separate, the newly created object will be handled by the same processor as the creating object. If, however, *t* is separate, the new object will be allocated to a new processor.

Object states

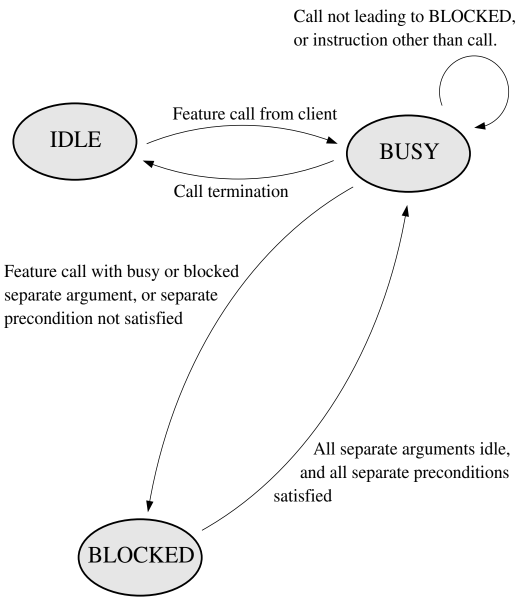

Figure 7: States of an object and transitions

Once it has been created, an object will be in either of three states:

- Busy: a routine is being executed on the object for the benefit of a client object, and is not blocked.
- Blocked: the last routine to be started on the object has requested access to an object handled by a different processor, but the processor was not available or a separate precondition was not satisfied.
- Idle: none of the above. There is no pending client request on the object.

In the object-oriented style of programming, a call is always executed on behalf of "someone" else – a client. The client is almost always another object. In sequential computation, there is only one exception: to start the execution of a system is to create an object – the root of the system's execution – and execute a creation procedure. The root object is the only object whose client is an external agent (the human who interactively starts the execution, or another system) rather than another object. With concurrent computation, the only new property here is that a system may have more than one root object.

The above diagram informally shows the transitions between states, as discussed in the following two sections. A processor is said to be available if and only if every one of the object that it handles is either idle or blocked.

Semantics of calls

To study the semantics of calls, we may use the general form $t.f$ (..., s, ...) and assume that f is a routine. (If f is an attribute, we may replace calls to f, for the purpose of this discussion, by calls to a function whose sole purpose is to return the value of the attribute; this may affect performance but does not change the semantics.)

The introduction of concurrency affects the semantics of a call only if one or more of the elements involved – target and actual arguments – are separate.

Let us assume first that one or more of the actual arguments are separate, but the target *t* is not. (The effect of having the target separate is examined in the next section.) The call is executed as part of the execution of a routine on a certain object, say C_OBJ, which may only be in a busy state at that stage. The property which determines whether the calls proceeds or blocks may be defined as follows:

> **Definition** (*satisfiable call*)
> A call to a routine *f* is satisfiable if and only if every separate actual argument *a* having a non-void value, and hence attached to a separate object A_OBJ, satisfies the following three conditions: A_OBJ is idle; the processor handling A_OBJ is available; Every separate clause of the precondition of *f*, when evaluated for A_OBJ and the actual arguments given, has value true. In this definition an assertion clause is separate if and only if it includes a call whose target is separate.

If the call is satisfiable, it proceeds immediately: C_OBJ remains in the busy state and A_OBJ enters the busy state, in which it executes *f*. When the execution of *f* terminates. At that stage A_OBJ returns to the idle state. If the call is not satisfiable, C_OBJ enters the blocked state. The call attempt has no immediate effect on its target and actual arguments. The processor handling C_OBJ becomes available; in particular, if any object also handled by that processor had previously blocked on a call that is now satisfiable, the processor will select one such object, return it to the busy state and execute the corresponding call as described above. If there is more than one selectable candidate, the semantics does not specify which one will be selected, but does require the processor to select one.

If a processor is available and there is no immediately satisfiable call, the processor remains available until a state is reached in which there is a satisfiable call for that processor.

A note is in order on exactly when preconditions clauses will cause blocking in the above definition of satisfiability. A precondition clause is separate (and so may clause blocking) only if it includes a call on a separate target. Other precondition clauses remain correctness conditions, not blocking conditions; for example, a precondition of the form $i > 1$, where *i* is an integer entity (expanded, and hence non-separate) is a normal correctness condition, which must be ensured before the call by the client. Failure to ensure this condition at the time of the call indicates a bug in the client, not a run-time waiting condition. Similarly, for separate *s1* and *s2*, preconditions clauses of the form *s1 /= Void* or *s1 = s2* are correctness conditions since they do not involve any calls. Both are tests for reference equality, which may be performed regardless of the status (idle, busy, waiting) of the objects attached to *s1* and *s2* if any. In particular, separate actual arguments may cause blocking only if they are non-void.

An adaptation of the above description is applicable to classes inheriting from class *CONCURRENCY*. The effect of calls to procedures *immediate_service*, *normal_service*, *insist* and *yield* is defined by the table on page 30.

Separate targets and lazy wait

The final semantic change is the effect of having a separate target *t* in a call of the form ***t*.*f*** *(..., s, ...)*. This appears in a routine *r* of which *t* must be a formal argument. When the call is executed, the object T_OBJ attached to *t* is busy (it became busy when the latest call to *r* was selected) and "reserved" by *r*. The previous discussion still applies; the only new effect is the lazy wait policy, which affects the client executing the above call.

> **Lazy wait policy on satisfiable calls with separate targets**:
> Once a satisfiable call has been selected for execution:
> - If the call is to a procedure, the client continues its execution unless one of its actual arguments is a non-separate reference.
> - If the call is to a query (function or attribute), or includes a non-separate reference among its actual arguments, the client's execution waits until the completion of the code.

7 CONCLUSION AND OPEN ISSUES

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

I consider the present article only as a first step towards a solution of the problem under study. The following points are open to criticism:

- The dual semantics of assertions on separate and non-separate targets (see the above discussion).
- The rule that the target of a separate call must be a formal argument of the enclosing routine, which may may require adding apparently superfluous *actual_xxx* routines.

In addition, much practical and theoretical work remains to be done in the following areas:

- Devising the practical facilities for associating processors, as defined in this article, with physical resources: computers on a network, tasks in an operating system, physical processors in a multiprocessing system.
- Implementing the mechanism in various environments – shared memory, distributed systems and others.
- Exploring the proof rules further and performing proofs of significant systems.
- Studying whether it is possible, by observing certain restrictions in the use of the mechanisms described above, to guarantee deadlock avoidance.
- Finding out whether the mechanism can be adapted to ensure fairness, and if so, how (perhaps through library facilities, as was done to achieve the semantics of duels).

One may hope that these questions can be answered in a way that preserves the advantages of the concurrency mechanism described here, in particular its minimal departure from the concepts of sequential object-oriented computation and its compatibility with the assertion concepts which are so essential to a proper understanding of this field.

ACKNOWLEDGMENTS

(Names are included here to acknowledge comments, criticism and help, not to imply endorsement in any way.) Early work leading to this article was presented in March 1990 at the TOOLS EUROPE conference in Paris [12]. The mechanism described here was the topic of a presentation at the TOOLS PACIFIC conference in Sydney in December 1992. In the meantime, I greatly benefited from the work done by John Potter and Ghinwa Jalloul from University of Technology, Sydney (UTS), starting from the original article, and from several discussions with them. (The ***hold*** construct discussed on page 21 was their suggestion.) I am also grateful to UTS for the opportunity to work on the topic of object-oriented concurrency during my stay there from August to October, 1992. The design of the lazy wait facility was influenced by the work of Denis Caromel [6]. Important insights were gained at various stages of this work from discussions with Richard Bielak, John Bruno, Carlo Ghezzi, Peter Löhr, Dino Mandrioli, Jean-Marc Nerson, Robert Switzer and Kim Waldén.

REFERENCES

[1] Gul Agha, “Concurrent Object-Oriented Programming”, *Communications of the ACM*, vol. 33, no. 9, pp. 125-141, September 1990.

[2] Pierre America and Frank van der Linden, “A parallel Object-Oriented Language with Inheritance and Subtyping”, in *Proceedings of ECOOP-OOPSLA 90*, Ottawa, Canada, 1990.

[3] Pierre America, “Designing an Object-Oriented Programming Language with Behavioural Subtyping”, in *Foundations of Object-Oriented Languages*, ed. J.W. de Bakker, W.P. de Roever, G. Rozenberg, pp. 60-90, Springer-Verlag, Berlin-New York, 1991.

[4] Moshe Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall International, Hemel Hempstead, 1991.

[5] Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug and Kristen Nygaard, *Simula Begin*, Studentliteratur and Auerbach Publishers, 1973.

[6] Denis Caromel, “Concurrency: An Object-Oriented Approach.”, in *TOOLS 9* (Proceedings of TOOLS EUROPE 1990), ed. Jean Bézivin et al., pp. 183-197, SOL/Angkor, Paris, 1990.

[7] Denis Caromel, “Towards a Method of Object-Oriented Concurrent Programming”, *Communications of the ACM*, September 1993. To appear.

[8] Brent T. Hailpern, *Verifying Concurrent Processes Using Temporal Logic*, Lecture Notes in Computer Science 129, Springer-Verlag, Berlin-New York, 1982.

[9] C.A.R. Hoare, “Procedures and Parameters: An Axiomatic Approach”, in *Symposium on the Semantics of Programming Languages*, ed. Erwin Engeler, pp. 103-116, Lecture Notes in Mathematics 188, Springer-Verlag, Berlin-New York, 1971. Reprinted in C.A.R. Hoare and C. B. Jones (ed.), *Essays in Computing Science*, Prentice Hall International, Hemel Hempstead, 1989, pages 75-88.

[10] Henry Lieberman, “Concurrent Object-Oriented Programming in Act 1”, in *Object-oriented Concurrent Programming*, ed. Akinori Yonezawa and Mario Tokoro, pp. 9-36, MIT Press, Cambridge (Mass.), 1987.

[11] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[12] Bertrand Meyer, “Sequential and Concurrent Object-Oriented Programming”, in *TOOLS 90* (*Technology of Object-Oriented Languages and Systems*), pp. 17-28, Angkor/SOL, Paris, June 1990.

[13] Bertrand Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990.

[14] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1991.

[15] Bertrand Meyer, “Applying "Design by Contract"”, *IEEE Computer*, vol. 40-51, no. 10, October 1992.

[16] Michael Papathomas, “Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming”, PhD Thesis, Université de Genève, 1992.

[17]SIS, “Data Processing- Programming Languages – SIMULA”, Svensk Standard SS 63 61 14, Standardiseringskommissionen i Sverige (Swedish Standards Institute), 20 May 1987.

[18]Akinori Yonezawa and Mario Tokoro (Eds), *Object-oriented Concurrent Programming*, MIT Press, Cambridge (Mass.), 1987.