30

Concurrency, distribution, client-server and the Internet

Like humans, computers can team up with their peers to achieve results that none of them could obtain alone; unlike humans, they can do many things at once (or with the appearance of simultaneity), and do all of them well. So far, however, the discussion has implicitly assumed that the computation is *sequential* — proceeds along a single thread of control. We should now see what happens when this assumption no longer holds, as we move to *concurrent* (also known as *parallel*) computation.

Concurrency is not a new subject, but for a long time interest in it remained mostly confined to four application areas: operating systems, networking, implementation of database management systems, and high-speed scientific software. Although strategic and prestigious, these tasks involve only a small subset of the software development community.

Things have changed. Concurrency is quickly becoming a required component of just about every type of application, including some which had traditionally been thought of as fundamentally sequential in nature. Beyond mere concurrency, our systems, whether or not *client-server*, must increasingly become *distributed* over networks, including the network of networks — the *Internet*. This evolution gives particular urgency to the central question of this chapter: can we apply object-oriented ideas in a concurrent and distributed context?

Not only is this possible: object technology can help us develop concurrent and distributed applications simply and elegantly.

30.1 A SNEAK PREVIEW

As usual, this discussion will not throw a pre-cooked answer at you, but instead will carefully build a solution from a detailed analysis of the problem and an exploration of possible avenues, including a few dead ends. Although necessary to make you understand the techniques in depth, this thoroughness might lead you to believe that they are complex; that would be inexcusable, since the concurrency mechanism on which we will finally settle is in fact characterized by almost incredible simplicity. To avoid this risk, we will begin by examining a summary of the mechanism, without any of the rationale.

Warning: SPOILER! (*The next section is 30.2, page*)

If you hate “spoilers”, preferring to start with the full statement of the issues and to let the drama proceed to its dénouement step by step and inference by inference, ignore the one-page summary that follows and skip directly to the next section.

The extension covering full-fledged concurrency and distribution will be as minimal as it can get starting from a sequential notation: a single new keyword — **separate**. How is this possible? We use the fundamental scheme of O-O computation: feature call, *x*.*f* (*a*), executed on behalf of some object O1 and calling *f* on the object O2 attached to *x*, with the argument *a*. But instead of a single processor that handles operations on all objects, we may now rely on different processors for O1 and O2 — so that the computation on O1 can move ahead without waiting for the call to terminate, since another processor handles it.

Because the effect of a call now depends on whether the objects are handled by the same processor or different ones, the software text must tell us unambiguously what the intent is for any *x*. Hence the need for the new keyword: rather than just *x*: *SOME_TYPE*, we declare *x*: **separate** *SOME_TYPE* to indicate that *x* is handled by a different processor, so that calls of target *x* can proceed in parallel with the rest of the computation. With such a declaration, any creation instruction !! *x*.*make* (…) will spawn off a new processor — a new thread of control — to handle future calls on *x*.

Nowhere in the software text should we have to specify *which* processor to use. All we state, through the **separate** declaration, is that two objects are handled by different processors, since this radically affects the system's semantics. Actual processor assignment can wait until run time. Nor do we settle too early on the exact nature of processors: a processor can be implemented by a piece of hardware (a computer), but just as well by a task (process) of the operating system, or, on a multithreaded OS, just a thread of such a task. Viewed by the software, "processor" is an abstract concept; you can execute the same concurrent application on widely different architectures (time-sharing on one computer, distributed network with many computers, threads within one Unix or Windows task…) without any change to its source text. All you will change is a "Concurrency Configuration File" which specifies the last-minute mapping of abstract processors to physical resources.

We need to specify synchronization constraints. The conventions are straightforward:

- No special mechanism is required for a client to resynchronize with its supplier after a separate call *x*.*f* (*a*) has gone off in parallel. The client will wait when and if it needs to: when it requests information on the object through a query call, as in *value* := *x*.*some_query*. This automatic mechanism is called *wait by necessity*.
- To obtain exclusive access to a separate object O2, it suffices to use the attached entity *a* as an argument to the corresponding call, as in *r* (*a*).
- A routine precondition involving a separate argument such as *a* causes the client to wait until the precondition holds.
- To guarantee that we can control our software and predict the result (in particular, rest assured that class invariants will be maintained), we must allow the processor in charge of an object to execute at most one routine at any given time.
- We may, however, need to *interrupt* the execution of a routine to let a new, high-priority client take over. This will cause an exception, so that the spurned client can take the appropriate corrective measures — most likely retrying after a while.

This covers most of the mechanism, which will enable us to build the most advanced concurrent and distributed applications through the full extent of O-O techniques, from multiple inheritance to Design by Contract — as we will now study in detail, forgetting for a while all that we have read in this short preview.

A complete summary appears in 30.11, page 1025.

30.2 THE RISE OF CONCURRENCY

Back to square one. We must first review the various forms of concurrency, to understand how the evolution of our field requires most software developers to make concurrency part of their mindset. In addition to the traditional concepts of multiprocessing and multiprogramming, the past few years have introduced two innovative concepts: object request brokers and remote execution through the Net.

Multiprocessing

More and more, we want to use the formidable amount of computing power available around us; less and less, we are willing to wait for the computer (although we have become quite comfortable with the idea that the computer is waiting for us). So if one processing unit would not bring us quickly enough the result that we need, we will want to rely on several units working in parallel. This form of concurrency is known as multiprocessing.

Spectacular applications of multiprocessing have involved researchers relying on hundreds of computers scattered over the Internet, at times when the computers' (presumably consenting) owners did not need them, to solve computationally intensive problems such as breaking cryptographic algorithms. Such efforts do not just apply to computing research: Hollywood's insatiable demand for realistic computer graphics has played its part in fueling progress in this area; the preparation of the movie *Toy Story*, one of the first to involve artificial characters only (only the voices are human), relied at some point on a network of more than one hundred high-end workstations — more economical, it seems, than one hundred professional animators.

Multiprocessing is also ubiquitous in high-speed scientific computing, to solve ever larger problems of physics, engineering, meteorology, statistics, investment banking.

More routinely, many computing installations use some form of *load balancing*: automatically dispatching computations among the various computers available at any particular time on the local network of an organization.

Another form of multiprocessing is the computing architecture known as **client-server computing**, which assigns various specialized roles to the computers on a network: the biggest and most expensive machines, of which a typical company network will have just one or a few, are "servers" handling shared databases, heavy computations and other strategic central resources; the cheaper machines, ubiquitously located wherever there is an end user, handle decentralizable tasks such as the human interface and simple computations; they forward to the servers any task that exceeds their competence.

The current popularity of the client-server approach is a swing of the pendulum away from the trend of the preceding decade. Initially (nineteen-sixties and seventies) architectures were centralized, forcing users to compete for resources. The personal computer and workstation revolution of the eighties was largely about empowering users with resources theretofore reserved to the Center (the "glass house" in industry jargon). Then they discovered the obvious: a personal computer cannot do everything, and some resources *must* be shared. Hence the emergence of client-server architectures in the nineties. The inevitable cynical comment — that we are back to the one-mainframe-many-terminals architecture of our youth, only with more expensive terminals now called "client workstations" — is not really justified: the industry is simply searching, through trial and error, for the proper tradeoff between decentralization and sharing.

Multiprogramming

The other main form of concurrency is multiprogramming, which involves a single computer working on several tasks at once.

If we consider general-purpose systems (excluding processors that are embedded in an application device, be it a washing machine or an airplane instrument, and single-mindedly repeat a fixed set of operations), computers are almost always multi-programmed, performing operating system tasks in parallel with application tasks. In a strict form of multiprogramming the parallelism is apparent rather than real: at any single time the processing unit is actually working on just one job; but the time to switch between jobs is so short that an outside observer can believe they proceed concurrently. In addition, the processing unit itself may do several things in parallel (as in the advance fetch schemes of many computers, where each clock cycle loads the next instruction at the same time it executes the current one), or may actually be a combination of several processing units, so that multiprogramming becomes intertwined with multiprocessing.

A common application of multiprogramming is *time-sharing*, allowing a single machine to serve several users at once. But except in the case of very powerful "mainframe" computers this idea is considered much less attractive now than it was when computers were a precious rarity. Today we consider our time to be the more valuable resource, so we want the system to do several things at once just for us. In particular, *multi-windowing* user interfaces allow several applications to proceed in parallel: in one window we browse the Web, in another we edit a document, in yet another we compile and test some software. All this requires powerful concurrency mechanisms.

Providing each computer user with a multi-windowing, multiprogramming interface is the responsibility of the operating system. But increasingly the users of the software we develop want to have concurrency *within one application*. The reason is always the same: they know that computing power is available by the bountiful, and they do not want to wait idly. So if it takes a while to load incoming messages in an e-mail system, you will want to be able to send an outgoing message while this operation proceeds. With a good Web browser you can access a new site while loading pages from another. In a stock trading system, you may at any single time be accessing market information from several stock exchanges, buying here, selling there, and monitoring a client's portfolio.

It is this need for intra-application concurrency which has suddenly brought the whole subject of concurrent computing to the forefront of software development and made it of interest far beyond its original constituencies. Meanwhile, all the traditional applications remain as important as ever, with new developments in operating systems, the Internet, local area networks, and scientific computing — where the continual quest for speed demands ever higher levels of multiprocessing.

Object request brokers

Another important recent development has been the emergence of the CORBA proposal from the Object Management Group, and the OLE 2/ActiveX architecture from Microsoft. Although the precise goals, details and markets differ, both efforts promise substantial progress towards distributed computing.

The general purpose is to allow applications to access each other's objects and services as conveniently as possible, either locally or across a network. The CORBA effort (more precisely its CORBA 2 stage, clearly the interesting one) has also placed particular emphasis on interoperability:

- CORBA-aware applications can coöperate even if they are based on "object request brokers" from different vendors.
- Interoperability also applies to the language level: an application written in one of the supported languages can access objects from an application written in another. The interaction goes through an intermediate language called IDL (Interface Definition Language); supported languages have an official IDL binding, which maps the constructs of the language to those of IDL.

IDL is a common-denominator O-O language centered on the notion of interface. An IDL interface for a class is similar in spirit to a short form, although more rudimentary (IDL in particular does not support assertions); it describes the set of features available on a certain abstraction. From a class written in an O-O language such as the notation of this book, tools will derive an IDL interface, making the class and its instances of interest to client software. A client written in the same language or another can, through an IDL interface, access across a network the features provided by such a supplier.

Remote execution

Another development of the late nineties is the mechanism for remote execution through the World-Wide Web.

The first Web browsers made it not just possible but also convenient to explore information stored on remote computers anywhere in the world, and to follow logical connections, or *hyperlinks*, at the click of a button. But this was a passive mechanism: someone prepared some information, and everyone else accessed it read-only.

The next step was to move to an active setup where clicking on a link actually triggers execution of an operation. This assumes the presence, within the Web browser, of an execution engine which can recognize the downloaded information as executable code, and execute it. The execution engine can be a built-in part of the browser, or it may be dynamically attached to it in response to the downloading of information of the corresponding type. This latter solution is known as a **plug-in** mechanism and assumes that users interested in a particular execution mechanism can download the execution engine, usually free, from the Internet.

This idea was first made popular by Java in late 1995 and 1996; Java execution engines have become widely available. Plug-ins have since appeared for many other mechanisms. An alternative to providing a specific plug-in is to generate, from any source language, code for a widely available engine, such as a Java engine; several compiler vendors have indeed started to provide generators of Java "bytecode" (the low-level portable code that the Java engine can execute).

For the notation of this book the two avenues have been pursued: ISE has a free execution engine; and at the time of writing a project is in progress to generate Java bytecode.

Either approach raises the potential of **security** problems: how much do you trust someone's application? If you are not careful, clicking on an innocent-looking hyperlink could unleash a vicious program that destroys files on your computer, or steals your personal information. More precisely you should not, as a user, be the one asked to be careful: the responsibility is on the provider of an execution engine and the associated library of basic facilities. Some widely publicized Java security failures in 1996 caused considerable worries about the issue.

The solution is to use carefully designed and certified execution engines and libraries coming from reputable sources. Often they will have two versions:

- One version is meant for unlimited Internet usage, based on a severely restricted execution engine.

  In ISE's tool the only I/O library facilities in this restricted tool only read and write to and from the terminal, not files. The "external" mechanism of the language has also been removed, so that a vicious application cannot cause mischief by going to C, say, to perform file manipulations. The Java "Virtual Machine" (the engine) is also draconian in what it permits Internet "applets" to do with the file system of your computer.

- The other version has fewer or no such restrictions, and provides the full power of the libraries, file I/O in particular. It is meant for applications that will run on a secure Intranet (internal company network) rather than the wilderness of the Internet.

In spite of the insecurity specter, the prospect of unfettered remote execution, a new step in the ongoing revolution in the way we distribute software, has generated enormous excitement, which shows no sign of abating.

30.3 FROM PROCESSES TO OBJECTS

To support all these mind-boggling developments, requiring ever more use of concurrent processing, we need powerful software support. How are we going to program these things? Object technology, of course, suggests itself.

Cited in [Matsuoka 1993].

Robin Milner is said to have exclaimed, in a 1991 workshop at an O-O conference, "*I can't understand why objects* [of O-O languages] *are not concurrent in the first place*". Even if only in the second or third place, how do we go about making objects concurrent?

If we start from non-O-O concurrency work, we will find that it largely relies on the notion of *process*. A process is a program unit that acts like a special-purpose computer: it executes a certain algorithm, usually repeating it until some external event triggers termination. A typical example is the process that manages a printer, repeatedly executing

> "Wait until there is at least a job in the print queue"
> "Get the next print job and remove it from the queue"
> "Print the job"

Various concurrency models differ in how processes are scheduled and synchronized, compete for shared hardware resources, and exchange information. In some concurrent programming languages, you directly describe a process; in others, such as Ada, you may also describe process *types*, which at run time are instantiated into processes, much as the classes of object-oriented software are instantiated into objects.

Similarities

The correspondence seems indeed clear. As we start exploring how to combine ideas from concurrent programming and object-oriented software construction, it seems natural to identify processes with objects, and process types with classes. Anyone who has studied concurrent computing and discovers O-O development, or the other way around, will be struck by the similarities between these two technologies:

- Both rely on autonomous, encapsulated modules: processes or process types; classes.
- Like processes and unlike the subroutines of sequential, non-O-O approaches, objects will, from each activation to the next, retain the values they contain.
- To build reasonable concurrent systems, it is indispensable in practice to enforce heavy restrictions on how modules can exchange information; otherwise things quickly get out of hand. The O-O approach, as we have seen, places similarly severe restrictions on inter-module communication.
- The basic mechanism for such communication may loosely be described, in both cases, under the general label of "message passing".

So it is not surprising that many people have had a "Eureka!" when first thinking, Milner-like, about making objects concurrent. The unification, it seems, should come easily.

This first impression is unfortunately wrong: after the similarities, one soon stumbles into the discrepancies.

Active objects

Building on the analogies just summarized, a number of proposals for concurrent O-O mechanisms (see the bibliographical notes) have introduced a notion of "active object". An active object is an object that is also a process: it has its own program to execute. In a definition from a book on Java:

From: *Doug Lea, "Concurrent Programming in Java", Addison-Wesley, 1996.*

> *Each object is a single, identifiable process-like entity* (*not unlike a Unix process*) *with state and behavior.*

This notion, however, raises difficult problems.

The most significant one is easy to see. A process has its own agenda: as illustrated by the printer example, it relentlessly executes a certain sequence of actions. Not so with classes and objects. An object does not *do* one thing; it is a repository of services (the features of the generating class), and just waits for the next client to solicit one of those services — chosen by the client, not the object. If we make the object active, it becomes responsible for the scheduling of its operations. This creates a conflict with the clients, which have a very clear view of what the scheduling should be: they just want the supplier, whenever they need a particular service, to be ready to provide it immediately!

The problem arises in non-object-oriented approaches to concurrency and has led to mechanisms for **synchronizing** processes — that is to say, specifying when and how each is ready to communicate, waiting if necessary for the other to be ready too. For example in a very simple, unbuffered producer-consumer scheme we may have a *producer* process that repeatedly executes

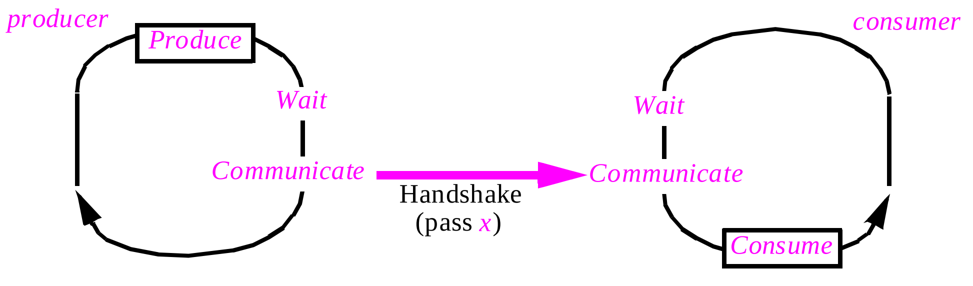
> "Make it known that *producer* is not ready"
> "Perform some computation that produces a value *x*"
> "Make it known that *producer* is ready"
> "Wait for *consumer* to be ready"
> "Pass *x* to *consumer*"

and a *consumer* process that repeatedly executes

Handshake

> "Make it known that *consumer* is ready"
> "Wait for *producer* to be ready"
> "Get *x* from *producer*"
> "Make it known that *consumer* is not ready"
> "Perform some computation that uses the value *x*"

a scheme which we may also view pictorially:



A simple producer-consumer scheme

Communication occurs when both processes are ready for each other; this is sometimes called a *handshake* or *rendez-vous*. The design of synchronization mechanisms — enabling us in particular to express precisely the instructions to "Make it known that *process* is ready" and "Wait for *process* to be ready" — has been a fertile area of research and development for several decades.

All this is fine for processes, the concurrent equivalent of traditional sequential programs which "do one thing"; indeed, a concurrent system built with processes is like a sequential system with several main programs. But in the object-oriented approach we have rejected the notion of main program and instead defined software units that stand ready to provide any one of a number of possible features.

Reconciling this view with the notion of process requires elaborate synchronization constructs to make sure that each supplier is ready to execute a feature when the client needs it. The reconciliation is particularly delicate when both client and supplier are active objects, since each has its own agenda.

All this does not make it *impossible* to devise mechanisms based on the notion of active object, as evidenced by the abundant literature on the subject (to which the bibliographical notes to this chapter give many references). But this evidence also shows the complexity of the proposed solutions, of which none has gained wide acceptance, suggesting that the active object approach is not the right one.

Active objects clash with inheritance

Doubts about the suitability of the active object approach grow as one starts looking at how it combines with other O-O mechanisms, especially inheritance.

If a class *B* inherits from a class *A* and both are active (that is to say, describe instances that must be active objects), what happens in *B* to the description of *A*'s process? In many cases you will need to add some new instructions, but without special language mechanisms this means that you will almost always have to redefine and rewrite the entire process part — not an attractive proposition.

See "Sequencing and inheritance", page 1121, as part of the discussion of Simula.

Here is an example of special language mechanism. Although the Simula 67 language does not support concurrency, it has a notion of active object: a Simula class can, besides its features, include a set of instructions, called the body of the class, so that we can talk of executing an object — meaning executing the body of its generating class. The body of a class *A* can include a special instruction **inner**, which has no effect in the class itself but, in a proper descendant *B*, stands for the body of *B*. So if the body of *A* reads

some_initialization; **inner**; *some_termination_actions*

and the body of *B* reads

specific_B_actions

then execution of that body actually means executing

some_initialization; *specific_B_actions*; *some_termination_actions*

Although the need for a mechanism of this kind is clear in a language supporting the notion of active object, objections immediately come to mind: the notation is misleading, since if you just read the body of *B* you will get a wrong view of what the execution does; it forces the parent to plan in detail for its descendants, going against basic O-O concepts (the Open-Closed principle); and it only works in a single-inheritance language.

Even with a different notation, the basic problem will remain: how to combine the process specification of a class with those of its proper descendants; how to reconcile parents' process specifications in the case of multiple inheritance.

Later in this chapter we will see other problems, known as the "inheritance anomaly" and arising from the use of inheritance with synchronization constraints.

"Synchronization for concurrent O-O computation", page 980

Faced with these difficulties, some of the early O-O concurrency proposals preferred to stay away from inheritance altogether. Although justifiable as a temporary measure to help understand the issues by separating concerns, this exclusion of inheritance cannot be sustained in a definitive approach to the construction of concurrent object-oriented software; this would be like cutting the arm because the finger itches. (For good measure, some of the literature adds that inheritance is a complex and messy notion anyway, as if telling the patient, after the operation, that having an arm was a bad idea in the first place.)

The inference that we may draw is simpler and less extreme. The problem is not object technology per se, in particular inheritance; it is not concurrency; it is not even the combination of these ideas. What causes trouble is the notion of active object.

Processes programmed

As we prepare to get rid of active objects it is useful to note that we will not really be renouncing anything. An object is able to perform many operations: all the features of its generating class. By turning it into a process, we select one of these operations as the only one that really counts. There is absolutely no benefit in doing this! Why limit ourselves to one algorithm when we can have as many as we want?

Another way to express this observation is that the notion of process need not be a built-in concept in the concurrency mechanism; processes can be *programmed* simply as routines. Consider for example the concept of printer process cited at the beginning of this chapter. The object-oriented view tells us to focus on the object type, printer, and to treat the process as just one routine, say *live*, of the corresponding class:

```
indexing
    description: "Printers handling one print job at a time"
    note: "A better version, based on a general class PROCESS, %
              %appears below under the name PRINTER"
class
    PRINTER_1
feature -- Status report
    stop_requested: BOOLEAN is do … end
    oldest: JOB is do … end
feature -- Basic operations
    setup is do … end
    wait_for_job is do … end
    remove_oldest is do … end
    print (j: JOB) is do … end
```

```
feature -- Process behavior
    live is
            -- Do the printer thing.
        do
            from setup until stop_requested loop
                wait_for_job; print (oldest); remove_oldest
            end
        end
    … Other features …
end -- class PRINTER_1
```

Note the provision for Other features: although so far *live* and the supporting features have claimed all our attention, we can endow processes with many other features if we want to, encouraged by the O-O approach developed elsewhere in this book. Turning *PRINTER_1* objects into processes would mean limiting this freedom; that would be a major loss of expressive power, without any visible benefit.

By abstracting from this example, which describes a particular process type simply as a class, we can try to provide a more general description of all process types through a deferred class — a *behavior class* as we have often encountered in previous chapters. Procedure *live* will apply to all processes. We could leave it deferred, but it is not too much of a commitment to note that most processes will need some initialization, some termination, and in-between a basic step repeated some number of times. So we can already effect a few things at the most abstract level:

```
indexing
    description: "The most general notion of process"
deferred class
    PROCESS
feature -- Status report
    over: BOOLEAN is
            -- Must execution terminate now?
        deferred
        end
feature -- Basic operations
    setup is
            -- Prepare to execute process operations (default: nothing).
        do
        end
    step is
            -- Execute basic process operations.
        deferred
        end
```

```
	wrapup is
			-- Execute termination operations (default: nothing).
		do
		end
feature -- Process behavior
	live is
			-- Perform process lifecycle.
		do
			from setup until over loop
				step
			end
			wrapup
		end
end -- class PROCESS
```

A point of methodology: whereas *step* is deferred, *setup* and *wrapup* are effective procedures, defined as doing nothing. This way we force every effective descendant to provide a specific implementation of *step*, the basic process action; but in the not infrequent cases that require no particular setup or termination operation we avoid bothering the descendants. This choice between a deferred version and a null effective version occurs regularly in the design of deferred classes, and you should resolve it based on your appreciation of the likely characteristics of descendants. A wrong guess is not a disaster; it will just lead to more effectings or more redefinitions in descendants.

From this pattern we may define a more specialized class, covering printers:

```
indexing
	description: "Printers handling one print job at a time"
	note: “Revised version based on class PROCESS"
class PRINTER inherit
	PROCESS
		rename over as stop_requested end
feature -- Status report
	stop_requested: BOOLEAN
			-- Is the next job in the queue a request to shut down?
	oldest: JOB is
			-- The oldest job in the queue
		do … end
feature -- Basic operations
	step is
			-- Process one job.
		do
			wait_for_job; print (oldest); remove_oldest
		end
```

```
        wait_for_job is
                    -- Wait until job queue is not empty.
                do
                    …
                ensure
                    oldest /= Void
                end
        remove_oldest is
                    -- Remove oldest job from queue.
                require
                    oldest /= Void
                do
                    if oldest.is_stop_request then stop_requested := True end
                    "Remove oldest from queue"
                end
        print (j: JOB) is
                    -- Print j, unless it is just a stop request.
                require
                    j /= Void
                do
                    if not j.is_stop_request then "Print the text associated with j" end
                end
    end -- class PRINTER
```

Exercise E30.1, page 1035.

The class assumes that a request to shut off the printer is sent as a special print job *j* for which *j.is_stop_request* is true. (It would be cleaner to avoid making *print* and *remove_oldest* aware of the special case of the "stop request" job; this is easy to improve.)

The benefits of O-O modeling are apparent here. In the same way that going from main program to classes broadens our perspective by giving us abstract objects that are not limited to "doing just one thing", considering a printer process as an object described by a class opens up the possibility of new, useful features. With a printer we can do more than execute its normal printing operation as covered by *live* (which we should perhaps have renamed *operate* when inheriting it from *PROCESS*); we might want to add such features as *perform_internal_test*, *switch_to_Postscript_level_1* or *set_resolution*. The equalizing effect of the O-O method is as important here as in sequential software.

More generally, the classes sketched in this section show how we can use the normal object-oriented mechanisms — classes, inheritance, deferred elements, partially implemented patterns — to implement processes. There is nothing wrong with the concept of process in an O-O context; indeed, we will need it in many concurrent applications. But rather than a primitive mechanism it will simply be covered by a **library class** *PROCESS* based on the version given earlier in this section, or perhaps several such classes covering variants of the notion.

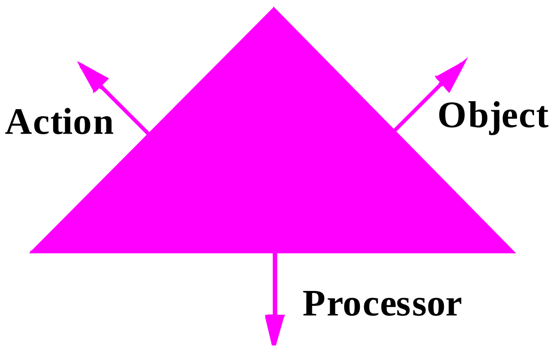
For the basic new construct of concurrent object technology, we must look elsewhere.

30.4 INTRODUCING CONCURRENT EXECUTION

What — if not the notion of process — fundamentally distinguishes concurrent from sequential computation?

Processors

To narrow down the specifics of concurrency, it is useful to take a new look at the figure which helped us lay the very foundations of object technology by examining the three basic ingredients of computation:



The three forces of computation

(*This figure first appeared on page 101*.)

To perform a computation is to use certain *processors* to apply certain *actions* to certain *objects*. At the beginning of this book we discovered how object technology addresses fundamental issues of reusability and extendibility by building software architectures in which actions are attached to objects (more precisely, object types) rather than the other way around.

What about processors? Clearly we need a mechanism to execute the actions on the objects. But in sequential computation there is just one thread of control, hence just one processor; so it is taken for granted and remains implicit most of the time.

In a concurrent context, however, we will have two or more processors. This property is of course essential to the idea of concurrency and we can take it as the definition of the notion. This is the basic answer to the question asked above: processors (not processes) will be the principal new concept for adding concurrency to the framework of sequential object-oriented computation. A concurrent system may have any number of processors, as opposed to just one for a sequential system.

The nature of processors

Definition: processor

A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects.

This is an abstract notion, it should not be confused with that of physical processing device, for which the rest of this chapter will use the term **CPU**, common in computer engineering to denote the processing units of computers. “CPU” is an abbreviation of “Central Processing Unit” even though there is most of the time nothing central about CPUs. You can use a CPU to implement a processor; but the notion of processor is much more abstract and general. A processor can be, for example:

- A computer (with its CPU) on a network.
- A task, also called process, as supported on operating systems such as Unix, Windows and many others.
- A coroutine. (Coroutines, covered in detail later in this chapter, simulate true concurrency by taking turns at execution on a single CPU; after each interruption, each coroutine resumes its execution where it last left it.)
- A “thread” as supported by such multi-threaded operating systems as Solaris, OS/2 and Windows NT.

Threads are mini-processes. A true process can itself contain many threads, which it manages directly; the operating system (OS) only sees the process, not its threads. Usually the threads of a process will all share the same address space (in object-oriented terms, they potentially have access to the same set of objects), whereas each process has its own address space. We may view threads as coroutines within a process. The main advantage of threads is efficiency: whereas creating a process and synchronizing it with other processes are expensive operations, requiring direct OS intervention (to allocate the address space and the code of the process), the corresponding operations on threads are much simpler, do not involve any expensive OS operations, and so can be faster by a factor of several hundreds or even several thousands.

The difference between processors and CPUs was clearly expressed by Henry Lieberman (for a different concurrency model):

[Lieberman 1987], *page 22. Square brackets signal differences in terminology.*

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.*

To avoid any misunderstanding, be sure to remember that throughout this chapter the “processors” denote virtual threads of control; any reference to the physical units of computation uses the term CPU.

At some point before or during you will need to assign computational resources to the processors. The mapping will be expressed by a “Concurrency Control File”, as described below, or associated library facilities.

Handling an object

Any feature call must be handled (executed) by some processor. More generally, any object O2 is *handled* by a certain processor, its *handler*; the handler is responsible for executing all calls on O2 (all calls of the form $x.f(a)$ where x is attached to O2).

We may go further and specify that the handler is assigned to the object at the time of creation, and remains the same throughout the object's life. This assumption will help keep the mechanism simple. It may seem restrictive at first, since some distributed systems may need to support *object migration* across a network. But we can address this need in at least two other ways:

- By allowing the reassignment of a processor to a different CPU (with this solution, all objects handled by a processor will migrate together).
- By treating object migration as the creation of a new object.

The dual semantics of calls

With multiple processors, we face a possible departure from the usual semantics of the fundamental operation of object-oriented computation, feature call, of one of the forms

```
x.f (a)                -- if f is a command
y := x.f (a)           -- if f is a query
```

As before, let O2 be the object attached to *x* at the time of the call, and O1 the object on whose behalf the call is executed. (In other words, the instruction in either form is part of a call to a certain routine, whose execution uses O1 as its target.)

We have grown accustomed to understanding the effect of the call as the execution of *f*'s body applied to O2, using *a* as argument, and returning a result in the query case. If the call is part of a sequence of instructions, as with

```
… previous_instruction; x.f (a); next_instruction; …
```

(or the equivalent in the query case), the execution of *next_instruction* will not commence until after the completion of *f*.

Not so any more with multiple processors. The very purpose of concurrent architectures is to enable the client computation to proceed without waiting for the supplier to have completed its job, if that job is handled by another processor. In the example of print controllers, sketched at the beginning of this chapter, a client application will want to send a print request (a "job") and continue immediately with its own agenda.

So instead of one call semantics we now have two cases:

- If O1 and O2 have the same handler, any further operation on O1 (*next_instruction*) must wait until the call terminates. Such calls are said to be **synchronous**.
- If O1 and O2 are handled by different processors, operations on O1 can proceed as soon as it has initiated the call on O2. Such calls are said to be **asynchronous**.

The asynchronous case is particularly interesting for a command, since the remainder of the computation may not need any of the effects of the call on O2 until much later (if at all: O1 may just be responsible for spawning one or more concurrent computations and then terminating). For a query, we need the result, as in the above example where we assign it to *y*, but as explained below we might be able to proceed concurrently anyway.

Separate entities

A general rule of software construction is that a semantic difference should always be reflected by a difference in the software text.

Now that we have two variants of call semantics we must make sure that the software text incontrovertibly indicates which one is intended in each case. What determines the answer is whether the call's target, O2, has the same handler (the same processor) as the call's originator, O1. So rather than the call itself we should mark *x*, the entity denoting the target object. In accordance with the static typing policy, developed in earlier chapters to favor clarity and safety, the mark should appear in the declaration of *x*.

This reasoning yields the only **notational extension** supporting concurrency. Along with the usual

x: *SOME_TYPE*

we allow ourselves the declaration form

x: **separate** *SOME_TYPE*

to express that *x* may become attached to objects handled by a different processor. If a class is meant to be used only to declare separate entities, you can also declare it as

separate class *X* … The rest as usual …

instead of just **class** *X* … or **deferred class** *X* ….

"Expanded types", page 254.

The convention is the same as for declaring an expanded status: you can declare *y* as being of type **expanded** *T*, or equivalently just as *T* if *T* itself is a class declared as **expanded class** *T*… The three possibilities — expanded, deferred, separate — are mutually exclusive, so at most one qualifying keyword may appear before **class**.

It is quite remarkable that this addition of a single keyword suffices to turn our sequential object-oriented notation into one supporting general concurrent computation.

Some straightforward terminology. We may apply the word "separate" to various elements, both static (appearing in the software text) and dynamic (existing at run time). Statically: a *separate class* is a class declared as **separate class** …; a *separate type* is based on a separate class; a *separate entity* is declared of a separate type, or as **separate** *T* for some *T*; *x*.*f* (…) is a *separate call* if its target *x* is a separate entity. Dynamically: the value of a separate entity is a *separate reference*; if not void, it will be attached to an object handled by another processor — a *separate object*.

Typical examples of separate class include:

- *BOUNDED_BUFFER*, to describe a buffer structure that enables various concurrent components to exchange data (some components, the producers, depositing objects into the buffer, and others, the consumers, acquiring objects from it).
- *PRINTER*, perhaps better called *PRINT_CONTROLLER*, to control one or more printers. By treating the print controllers as separate objects, applications do not need to wait for the print job to complete (unlike early Macintoshes, with which you were stuck until the last page had come out of the printer).

- *DATABASE*, which in the client part of a client-server architecture may serve to describe the database hosted by a distant server machine, to which the client may send queries through the network.
- *BROWSER_WINDOW*, in a Web browser that allows you to spawn a new window where you can examine different Web pages.

Obtaining separate objects

In practice, as illustrated by the preceding examples, separate objects will be of two kinds:

- In the first case an application will want to spawn a *new* separate object, grabbing the next available processor. (Remember that we can always get a new processor; since processors are not material resources but abstract facilities, their number is not bounded.) This is typically the case with *BROWSER_WINDOW*: you create a new window when you need one. A *BOUNDED_BUFFER* or *PRINT_CONTROLLER* may also be created in this way.
- An application may simply need to access an *existing* separate object, usually shared between many different clients. This is the case in the *DATABASE* example: the client application uses an entity *db_server*: **separate** *DATABASE* to access the database through such separate calls as *db_server*.*ask_query* (*sql_query*). The server must have at some stage obtained the value of *server* — the database handle — from the outside. Accesses to existing *BOUNDED_BUFFER* or *PRINT_CONTROLLER* objects will use a similar scheme.

The separate object is said to be **created** in the first case and **external** in the second.

To obtain a created object, you simply use the creation instruction. If *x* is a separate entity, the creation instruction

!! *x*.*make* (…)

will, in addition to its usual effect of creating and initializing a new object, assign a new processor to handle that object. Such an instruction is called a *separate creation*.

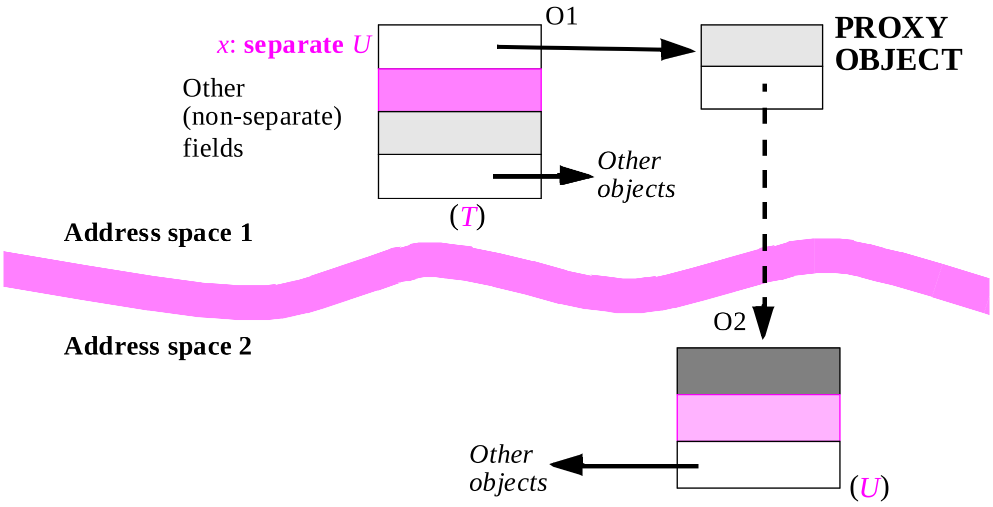
To obtain an existing external object, you will typically use an external routine, such as

server (*name*: *STRING*; … Other arguments …): **separate** *DATABASE*

where the arguments serve to identify the requested object. Such a routine will typically send a message over the network and obtain in return a reference to the object.

A word about possible implementations may be useful here to visualize the notion of separate object. Assume each of the processors is associated with a *task* (process) of an operating system such as Windows or Unix, with its own address space; this is of course just one of many concurrent architectures. Then one way to represent a separate object within a task is to use a small local object, known as a **proxy**:

A proxy for a separate object



The figure shows an object O1, instance of a class *T* with an attribute *x*: **separate** *U*. The corresponding reference field in O1 is conceptually attached to an object O2, handled by another processor. Internally, however, the reference leads to a proxy object, handled by the same processor as O1. The proxy is an internal object, not visible to the author of the concurrent application. It contains enough information to identify O2: the task that serves as O2's handler, and O2's address within that task. All operations on *x* on behalf of O1 or other clients from the same task will go through the proxy. Any other processor that also handles objects containing separate references to O2 will have its own proxy for O2.

Be sure to note that this is only one possible technique, not a required property of the model. Operating system tasks with separate address spaces are just one way to implement processors. With threads, for example, the techniques may be different.

Objects here, and objects there

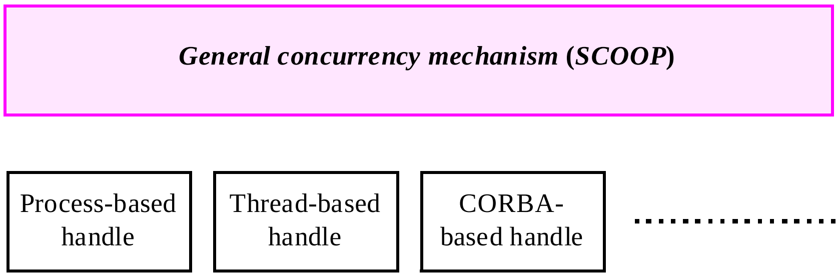
When first presented with the notion of separate entity, some people complain that it is over-committing: "I do not want to know where the object resides! I just want to request the operation, *x*.*f* (…), and let the machinery do the rest — execute *f* on *x* wherever *x* is."

Although legitimate, this desire to avoid over-commitment does not obviate the need for **separate** declarations. It is true that the *precise* location of an object is often an implementation detail that should not affect the software. But one "yes or no" property of the object's location remains relevant: whether the object is handled by the *same* processor or by another. This is a fundamental semantic difference since it determines whether calls on the object are synchronous or asynchronous — cause the client to wait, or not. Ignoring this property in the software would not be a convenience; it would be a mistake.

Once we know the object is separate, it should not in most cases matter for the functionality of our software (although it may matter for its performance) whether the object belongs to another thread of the same process, another process on the same computer, another computer in the same room, another room in the same building, another site on the company's private network, or another Internet node half-way around the world. But it matters that it is separate.

A concurrency architecture

The use of **separate** declarations to cover the fundamental boolean property "is this object here, or is it elsewhere?" while leaving room for various physical implementations of concurrency suggests a two-level architecture, similar to what is available for the graphical mechanisms (with the *Vision* library sitting on top of platform-specific libraries):



Two-level architecture for concurrency mechanism

(*See a similar architecture for graphical libraries on page 1067*.)

At the highest level the mechanism is platform-independent. This is the level which most applications use, and which this chapter describes. To perform concurrent computation, applications simply use the **separate** mechanism.

Internally, the implementation will rely on some practical concurrent architecture (lower level on the figure). The figure lists some possibilities:

- There may be an implementation using processes (tasks) as provided by the operating system. Each processor is associated with a process. This solution supports distributed computing: the process of a separate object can be on a remote machine as well as a local one. For non-distributed processing, it has the advantage that processes are stable and well known, and the disadvantage that they are CPU-intensive; both the creation of a new process and the exchange of information between processes are expensive operations.
- There may be an implementation using threads. Threads, as already noted, are a lighter alternative to processes, minimizing the cost of creation and context switching. Threads, however, have to reside on the same machine.
- A CORBA implementation is also possible, using CORBA distribution mechanisms as the physical layer to exchange objects across the network.
- Other possible mechanisms include PVM (Parallel Virtual Machine), the Linda language for concurrent programming, Java threads…

As always with such two-level architectures, the correspondence between high-level constructs and the actual platform mapping (the *handle* in terms of a previous chapter) is in most cases automatic, so that application developers will see the highest level only. But mechanisms must be available to let them access the lower level if they need to (and, of course, are ready to renounce platform-independence).

Mapping the processors: the Concurrency Control File

On Ace files see "Assembling a system", page 198.

If the software does not specify the physical CPUs, this specification must appear somewhere else. Here is a way to take care of it. This is only one possible solution, not a fundamental part of the approach; the exact format is not essential, but any configuration mechanism will somehow have to provide the same information.

Our example format is a "Concurrency Control File" (CCF) describing the concurrent computing resources available to our software. CCFs are similar in purpose and outlook to Ace files used to control system assembly. A typical CCF looks like this:

```
creation
    local_nodes:
        system
            "pushkin" (2): "c:\system1\appl.exe"
            "akhmatova" (4): "/home/users/syst1"
            Current: "c:\system1\appl2.exe"
        end
    remote_nodes:
        system
            "lermontov": "c:\system1\appl.exe"
            "tiuchev" (2): "/usr/bin/syst2"
        end
end

external
    Ingres_handler: "mandelstam" port 9000
    ATM_handler: "pasternak" port 8001
end

default
    port: 8001; instance: 10
end
```

Defaults are available for all properties of interest, so that each of the three possible parts (**creation**, **external**, **default**) is optional, as well as the CCF as a whole.

The **creation** part specifies what CPUs to use for separate creations (instructions of the form *!! x.make (…)* for separate *x*). The example uses two CPU groups: *local_nodes*, presumably covering local machines, and *remote_nodes*. The software can select a CPU group through a call such as

set_cpu_group ("local_nodes")

directing subsequent separate creations to use the CPU group *local_nodes* until the next call to *set_cpu_group*. This procedure comes from a class *CONCURRENCY* providing facilities for controlling the mechanism; we will encounter a few more of its features below.

The corresponding CCF entry specifies what CPUs to use for *local_nodes*: the first two objects will be created on machine *pushkin*, the next four on *akhmatova*, and the next ten on the current machine (the one which executes the creation instructions); after that the allocation scheme will repeat itself — two objects on *pushkin* and so on. In the absence of a processor count, as with *Current* here, the value is taken from the **instance** entry in the **default** part (here 10) if present, and is 1 otherwise. The system used to create each instance is an executable specified in each entry, such as *c:\system1\appl*.*exe* for *pushkin* (obviously a machine running Windows or OS/2).

In this example the processors are all mapped to processes. The CCF also supports assigning processors to threads (in the thread-based handle) or other concurrency mechanisms, although we need not concern ourselves with the details.

The **external** part specifies where to look for existing external separate objects. The CCF refers to these objects through abstract names, *Ingres_handler* and *ATM_handler* in the example, which the software will use as arguments to the functions that establish a connection with such an object. For example with the *server* function as assumed earlier

server (*name*: *STRING*; … Other arguments …): **separate** *DATABASE*

a call of the form *server* ("*Ingres_handler*", …) will yield a separate object denoting the Ingres database server. The CCF indicates that the corresponding object resides on machine *mandelstam* and is accessible on port 9000. In the absence of a port specification the value used is drawn from the **defaults** part or, barring that, a universal default.

The CCF is separate from the software. You may compile a concurrent or distributed application without any reference to a specific hardware and network architecture; then at run time each separate component of the application will use its CCF to connect to other existing components (**external** parts) and to create new components (**creation** parts).

This sketch of CCF conventions has shown how we can map the abstract concepts of concurrent O-O computation — processors, created separate objects, external separate objects — to physical resources. As noted, these conventions are only an example of what can be done, and they are not part of the basic concurrency mechanism. But they demonstrate that it is possible to decouple the software architecture of a concurrent system from the concurrent hardware architecture available at any particular stage.

Library mechanisms

With a CCF-like approach, the application software will, most of the time, not concern itself with the physical concurrency architecture. Some application developers may, however, need to exert a finer degree of control from within the application, at the possible expense of dynamic reconfigurability. Some CCF functionalities must then be accessible directly to the application, enabling it, for example, to select a specific process or thread for a certain processor. They will be available through libraries as part of the two-level concurrency architecture; it does not raise any difficult problem. We will encounter the need for more library mechanisms later in this chapter.

At the other extreme, some applications may want unlimited run-time reconfigurability. It is not enough then to have the ability to read a CCF or similar configuration information at start-up time and then be stuck with it. But we cannot either expect to re-read the configuration before each operation, as this would kill performance. The solution is once again to use a library mechanism: a procedure must be available to read or re-read the configuration information dynamically, allowing the application to adapt to a new configuration when (and only when) it is ready to do so.

Validity rules: unmasking traitors

Because the semantics of calls is different for separate and non-separate objects, it is essential to guarantee that a non-separate entity (declared as *x*: *T* for non-separate *T*) can never become attached to a separate object. Otherwise a call *x*•*f* (*a*) would wrongly be understood — by the compiler, among others — as synchronous, whereas the attached object is in fact separate and requires asynchronous processing. Such a reference, falsely declared as non-separate while having its loyalties on the other side, will be called a **traitor**. We need a simple validity rule to guarantee that our software has no traitor — that every representative or lobbyist of a separate power is duly registered as such with the appropriate authorities.

The rule will have four parts. The first part eliminates the risk of producing traitors through attachment, that is to say assignment or argument passing:

Separateness consistency rule (1)

If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.

An attachment of target *x* and source *y* is either an assignment *x* := *y* or a call *f* (…, *y*, …) where the actual argument corresponding to *x* is *y*. Having such an attachment with *y* separate but not *x* would make *x* a traitor, since we could use *x* to access a separate object (the object attached to *y*) under a non-separate name, as if it were a local object with synchronous call. The rule disallows this.

Note that syntactically *x* is an entity but *y* may be any expression. This means that the rule assumes we have defined the notion of "separate expression", in line with previous definitions. A simple expression is an entity; more complex expressions are function calls (remember in particular that an infix expression such as *a* + *b* is formally considered a call, similar to something like *a*•*plus* (*b*)). So the definition is immediate: an expression is separate if it is either a separate entity or a separate call.

As will be clear from the rest of the discussion, permitting an attachment of a non-separate source to a separate target is harmless — although usually not very useful.

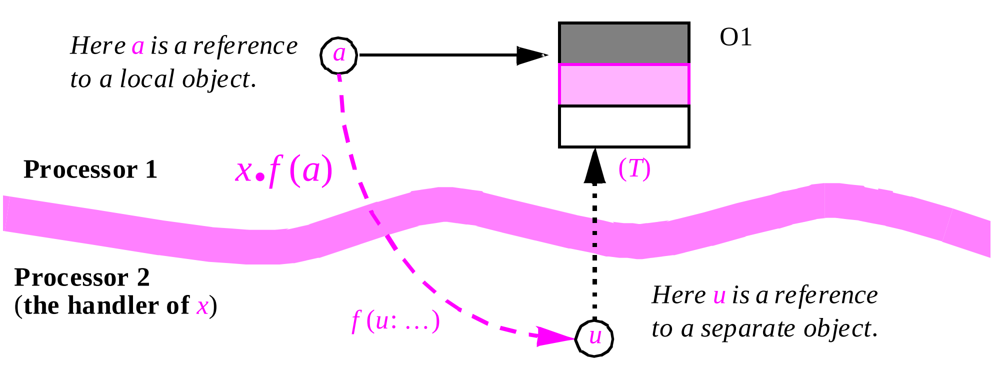
We need a complementary rule covering the case in which a client passes to a separate supplier a reference to a local object. Assume the separate call

x.*f* (*a*)

where *a*, of type *T*, is not separate, although *x* is. The declaration of routine *f*, for the generating class of *x*, will be of the form

f (*u*: … *SOME_TYPE*)

and the type *T* of *a* must conform to *SOME_TYPE*. But this is not sufficient! Viewed from the supplier's side (that is to say, from the handler of *x*), the object O1 attached to *a* has a different handler; so unless the corresponding formal argument *u* is declared as separate it would become a traitor, giving access to a separate object as if it were non-separate:


Here a is a reference to a local object.
a
O1
Processor 1
x.f (a)
(T)
Processor 2 (the handler of x)
f (u: …)
u
Here u is a reference to a separate object.

Passing a reference as argument to a separate call

So *SOME_TYPE* must be separate; for example it may be **separate** *T*. Hence the second consistency rule:

Separateness consistency rule (2)

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

The issue only arises for arguments of reference type. The other case, expanded types, including in particular the basic types such as *INTEGER*, is considered next.

As an application of the technique, consider an object that spawns several separate objects, giving them a way to rely later on its resources; it is saying to them, in effect, "*Here is my business card*; *call me if you need to*". A typical example would be an operating system's kernel that creates several separate objects and stands ready to perform operations for them when they ask. The creation calls will be of the form

!! *subsystem*.*make* (*Current*, … Other arguments …)

where *Current* is the "business card" enabling *subsystem* to remember its progenitor, and ask for its help in case of need. Because *Current* is a reference, the corresponding formal argument in *make* must be declared as separate. Most likely, *make* will be of the form

```
make (p: separate PROGENITOR_TYPE; … Other arguments …) is
    do
        progenitor := p
        … Rest of subsystem initialization operations …
    end
```

keeping the value of the progenitor argument in an attribute *progenitor* of the enclosing class. The second separateness consistency rule requires *p* to be declared as separate; so the first rule requires the same of attribute *progenitor*. Later calls for progenitor resources, of the form *progenitor*.*some_resource* (…) will, correctly, be treated as separate calls.

A similar rule is needed for function results:

Separateness consistency rule (3)

If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.

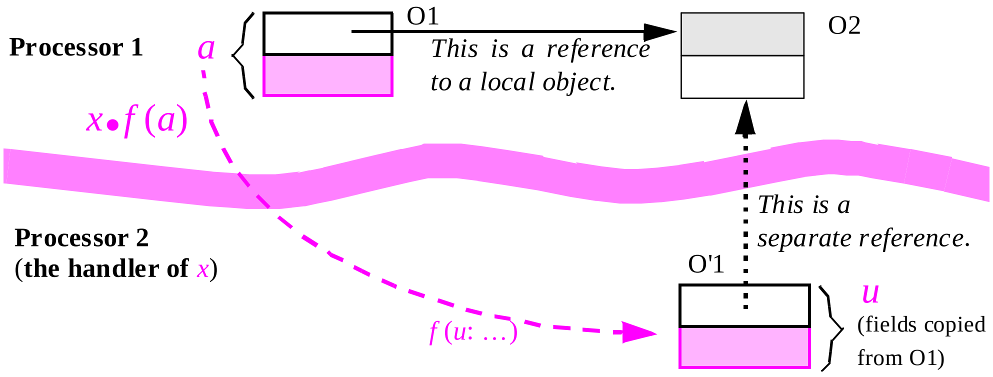
Since the last two rules only apply to actual arguments and results of reference types, we need one more rule for the other case, expanded types:

Separateness consistency rule (4)

If an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

In other words, the only expanded values that we can pass in a separate call are "completely expanded" objects, with no references to other objects. Otherwise we could again run into traitor trouble since attaching an expanded value implies copying an object:

Passing to a separate call an object with references


Processor 1
a
O1
This is a reference to a local object.
O2
x.f (a)
Processor 2 (the handler of x)
This is a separate reference.
O'1
u
(fields copied from O1)
f (u: …)

The figure illustrates the case in which the formal argument *u* is itself expanded. Then the attachment is simply a copy of the fields of the object O1 onto those of the object O'1 attached to *u*. Permitting O1 to contain a reference would produce a traitor field in O'1. The problem would also arise if O1 had a subobject with a reference; hence the mention "directly or indirectly" in the rule.

The effect of reattachment in these cases was defined in "ATTACHMENT: REFERENCE AND VALUE SEMANTICS", 8.8, page 261.

If the formal argument *u* is a reference, the attachment is a clone; the call would create a new object O'1 similar to the one on the last figure and attach reference *u* to it. In this case the solution is to create the clone explicitly on the client's side, before the call:

```
a: expanded SOME_TYPE; a1: SOME_TYPE
…
a1 := a;            -- This clones the object and attaches a1 to the clone.
x.f (a1)
```

As per the second validity rule, the formal argument *u* must be of a separate reference type, **separate** *SOME_TYPE* or conforming; the call on the last line makes *u* a separate reference attached to the newly created clone on the client's side.

Importing object structures

A consequence of the separateness consistency rules is that it is not possible to use the *clone* function (from the universal class *ANY*) to obtain an object handled by another processor. The function is declared as

See "Object cloning and equality", page 245.

```
clone (other: GENERAL): like other is
            -- New object, field-by-field identical to other
    …
```

so that an attempt to use *y* := *clone* (*x*) for separate *x* would violate part 1 of the rule: *x*, which is separate, does not conform to *other* which is not. This is what we want: a separate object running on a machine in Vladivostok may contain (non-separate) references to objects that are in Vladivostok too; but then if you could clone it in Kansas City, the resulting object would contain traitors — references to those objects, now separate, even though in the generating class the corresponding attributes are not declared as separate.

The following function, also in class *GENERAL*, enables us to clone separate object structures without producing traitors:

```
deep_import (other: separate GENERAL): GENERAL is
            -- New object, field-by-field identical to other
    …
```

You should be familiar with the notions of shallow and deep clone; see "Deep clone and comparison", page 247.

The result is a non-separate object structure, recursively duplicated from the separate structure starting at *other*. For the reasons just explained, a *shallow* import operation could yield traitors; so what we need is the equivalent of *deep_clone* applied to a separate object. Function *deep_import* provides it. It will produce a copy of the entire structure, making all the object copies non-separate. (It may of course still contain separate references if the original structure contained references to objects handled by another processor.)

For the developers of distributed systems, *deep_import* is a convenient and powerful mechanism, through which you can transfer possibly large object structures across a network without the need to write any specialized software, and with the guarantee that the exact structure (including cycles etc.) will be faithfully duplicated.

30.5 SYNCHRONIZATION ISSUES

We have our basic mechanism for starting concurrent executions (separate creation) and for requesting operations from these executions (the usual feature call mechanism). Any concurrent computation, object-oriented or not, must also provide ways to **synchronize** concurrent executions, that is to say to define timing dependencies between them.

If you are familiar with concurrency issues, you may have been surprised by the announcement that a single language mechanism, **separate** declarations, is enough to add full concurrency support to our sequential object-oriented framework. Surely we need specific synchronization mechanisms too? Actually no. The basic O-O constructs suffice to cover a wide range of synchronization needs, provided we adapt the definition of their semantics when they are applied to separate elements. It is a testimony of the power of the object-oriented method that it adapts so simply and gracefully to concurrent computation.

Synchronization *vs.* communication

To understand how we should support synchronization in object-oriented concurrency, it is useful to begin with a review of non-O-O solutions. Processes (the concurrent units in most of these solutions) need mechanisms of two kinds:

- *Synchronization* mechanisms enforce timing constraints. A typical constraint might state that a certain operation of a process, such as accessing a database item, may only occur after a certain operation of another process, such as initializing the item.
- *Communication* mechanisms allow processes to exchange information, which in the object-oriented case will be in the form of objects (including the special case of simple values such as integers) or object structures.

A simple classification of approaches to concurrency rests on the observation that some of them focus on the synchronization mechanism and then use ordinary non-concurrent techniques such as argument passing for communication, whereas others treat communication as the fundamental issue and deduce synchronization from it. We may talk about *synchronization-based* and *communication-based* mechanisms.

Synchronization-based mechanisms

The best known and most elementary synchronization-based mechanism is the **semaphore**, a locking tool for controlling shared resources. A semaphore is an object on which two operations are available: *reserve* and *free* (traditionally called *P* and *V*, but more suggestive names are preferable). At any time the semaphore is either reserved by a certain client or free. If it is free and a client executes *reserve*, the semaphore becomes reserved by that client. If the client that has reserved it executes *free*, the semaphore becomes free. If the semaphore is reserved by a client and another executes *reserve*, the new client will wait until the semaphore is free again. The following table summarizes this specification:

Semaphore operations

STATE OPERATION	Free	Reserved by me	Reserved by someone else
reserve	Becomes reserved by me.	▨	I wait.
free	▨	Becomes free.	▨

Events represented by shaded entries are not supposed to occur; they can be treated either as errors or as having no effect.

The policy for deciding which client gets through when two or more are waiting for a semaphore that gets freed may be part of the semaphore's specification, or may be left unspecified. (Usually clients expect a *fairness* property guaranteeing that if everyone gaining access to the semaphore ultimately frees it no one will wait forever.)

This description covers *binary* semaphores. The *integer* variant lets at most *n* clients through at any given time, for some *n*, rather than at most one.

Although many practical developments still rely on them, semaphores are widely considered too low-level for building large, reliable systems. But they provide a good starting point for discussing more advanced techniques.

Critical regions are a more abstract approach. A critical region is a sequence of instructions that may be executed by at most one client at a time. To ensure exclusive access to a certain object *a* you may write something like

hold *a* **then** … Operations involving fields of *a* …**end**

where the critical region is delimited by **then** … **end**. Only one client can execute the critical region at any given time; others executing a **hold** will wait.

Most applications need a more general variant, the **conditional critical region**, in which execution of the critical region is subject to a boolean condition. Consider a buffer shared by a producer, which can only write into the buffer if it is not full, and a consumer, which can only read from it if it is not empty; they may use the two respective schemes

hold *buffer* **when not** *buffer*.*full* **then** "Write into buffer, making it not empty" **end**

hold *buffer* **when not** *buffer*.*empty* **then** "Read from buffer, making it not full" **end**

Such interplay between input and output conditions cries for introducing assertions and giving them a role in synchronization, an idea to be exploited later in this chapter.

Another well-known synchronization-based mechanism, combining the notion of critical region with the modular structure of some modern programming languages, is the **monitor**. A monitor is a program module, not unlike the packages of Modula or Ada. The basic synchronization mechanism is simple: mutual exclusion at the routine level. At most one client may execute a routine of the monitor at any given time.

Also interesting is the notion of **path expression**. A path expression specifies the possible sequencing of a set of processes. For example the expression

$$init\ ;\ (reader^{*} \mid writer)^{+}\ ;\ finish$$

prescribes the following behavior: first an *init* process; then a state in which at any time either one *writer* process or any number of *reader* processes may be active; then a *finish* process. The asterisk * means any number of concurrent instances; the semicolon ; indicates sequencing; | means "either-or"; $^+$ means any number of successive repetitions. An argument often cited in favor of path expressions is that they specify the processes and the synchronization separately, avoiding interference between the description of individual algorithmic tasks and the description of their scheduling.

Yet another category of techniques for specifying synchronization relies on analyzing the set of **states** through which a system or system component can go, and transitions between these states. **Petri nets**, in particular, rely on graphical descriptions of the transitions. Although intuitive for simple hardware devices, such techniques quickly yield a combinatorial explosion in the number of states and transitions, and make it hard to work hierarchically (specifying subsystems independently, then recursively embedding their specifications in those of bigger systems). So they do not seem applicable to large, evolutionary software systems.

Communication-based mechanisms

Starting with Hoare's "Communicating Sequential Processes" (CSP) in the late seventies, most non-O-O concurrency work has focused on communication-based approaches.

The rationale is easy to understand. If you have solved the synchronization problem, you must still find a way to make concurrent units communicate. But if you devise a good communication mechanism you might very well have solved synchronization too: because two units cannot communicate unless the sender is ready to send and the receiver ready to receive, communication implies synchronization; pure synchronization may be viewed as the extreme case of communicating an empty message. If your communication mechanism is general enough, it will provide *all* the synchronization you need.

CSP is based on this "I communicate, therefore I synchronize" view. The starting point is a generalization of a fundamental concept of computing, input and output: a process receives information *v* from a certain "channel" *c* through the construct *c* ? *v*; it sends information to a channel through the construct *c* ! *v*. Channel input and output are only two among the possible examples of *events*.

For more flexibility CSP introduces the notion of non-deterministic wait, represented by the symbol ■, enabling a process to wait on several possible events and execute the action associated with the first that occurs. Assume for example a system enabling a bank's customers to make inquiries and transfers on their accounts, and the bank manager to check what is going on:

```
(balance_enquiry ? customer →
    (ask_password.customer ? password →
        (password_valid → (balance_out.customer ! balance)
       ▮ (password_invalid → (denial.customer ! denial_message)))
▮ transfer_request ? customer → …
▮ control_operation ? manager → …)
```

In the initial state the system stands ready to accept one of three possible input events: a *balance_enquiry* or *transfer_request* from a *customer*, or a *control_operation* from a *manager*. The first event that occurs will trigger the behavior described, using the same mechanisms, on the right of the corresponding arrow.

The right side of the arrow has only been filled in for the first event: after getting a *balance_enquiry* relative to a certain *customer*, you send the *customer* an *ask_password* event from which you expect to get the *password*; you validate the password, as a result sending to the *customer* one of two possible messages: *balance_out*, with the *balance* as argument, or *denial*.

Once the event's processing is complete, the system returns to its initial state, listening to possible input events.

The original version of CSP was a major influence on the concurrency mechanism of Ada, whose "tasks" are processes able to wait on several possible "entries" through an "accept" instruction. The Occam language, a direct implementation of CSP, is the primary programming tool for the *transputer*, a family of microprocessors designed specifically by Inmos (now SGS-Thomson) for the construction of highly concurrent architectures.

The example reflects the most recent CSP rather than the original 1978 version; see bibliographical notes. On Ada concurrency see p. 1091.

Synchronization for concurrent O-O computation

Many of the ideas just reviewed will help us find the right approach to concurrency in an object-oriented context. In the final form of the solution you will recognize concepts coming from CSP as well as monitors and conditional critical regions.

The CSP emphasis on communication seems right for us, since the central technique of our model of computation — calling a feature, with arguments, on an object — is a communication mechanism. But there is another reason for preferring a communication-based solution: a synchronization-based mechanism can conflict with inheritance.

This conflict is most obvious if we consider path expressions. The idea of using path expressions has attracted many researchers on O-O concurrency as a way to specify the actual processing, given by the features of a class, separately from the synchronization constraints, given by path expressions. The purely computational aspects of the software, which may have existed prior to the introduction of concurrency, will thus remain untainted by concurrency concerns. So for example if a class *BUFFER* has the features *remove* (remove the oldest element of the buffer) and *put* (add an element), we may express the synchronization through constraints such as

```
empty: {put}
partial: {put, remove}
full: {remove}
```

Notation and example from [Matsuoka 1993], which introduced the term "inheritance anomaly". For more details on the example see exercise E30.3, page 1035.

using a path-expression-like notation which lists three possible states and, for each of them, the permitted operations. But then assume you want a descendant *NEW_BUFFER* to provide an extra feature *remove_two* which removes two buffer items at a time (with a buffer size of at least three). Then you need an almost completely new set of states:

```
empty: {put}
partial_one: {put, remove}     -- State in which the buffer contains exactly one item
partial_two_or_more: {put, remove, remove_two}
full: {remove, remove_two}
```

and if the routines specify what states they produce in each possible case, they must all be redefined from *BUFFER* to *NEW_BUFFER*, defeating the purpose of inheritance.

This problem, and similar ones identified by several researchers, have been dubbed the **inheritance anomaly**, and have led some concurrent O-O language designers to view inheritance with suspicion. The first versions of the POOL parallel object-oriented language, for example, excluded inheritance (see the bibliographical notes).

Concerns about the "inheritance anomaly" have sparked an abundant literature proposing solutions, which generally try to decrease the amount of redefinition by looking for modular ways of specifying the synchronization constraints, so that descendants can describe the changes more incrementally, instead of having to redefine everything.

On closer examination, however, the problem does not appear to be inheritance, or even any inherent conflict between inheritance and concurrency, but instead the idea of specifying synchronization constraints separately from the routines themselves. (The formalisms discussed actually do not quite meet this goal anyway, since the routines must specify their exit states.)

To the reader of this book, familiar with the principles of Design by Contract, the technique using explicit states and a list of the features applicable in each state will look too low-level. The specifications of *BUFFER* and *NEW_BUFFER* obscure fundamental properties that we have learned to characterize through preconditions: *put* should state **require not** *full*; similarly, *remove_two* should state **require** *count >= 2*; and so on. This more compact and more abstract specification is easier to explain, to adapt (changing a routine's precondition does not affect any other routine), and to relate to the views of outsiders such as customers. State-based techniques appear more restrictive and error-prone. They also raise the risk of combinatorial explosion mentioned in relation to Petri nets and other state-based models: for the above elementary examples the number of states is already three in one case and four in the other, suggesting that in a complex system it might become unmanageable.

See "Active objects clash with inheritance", page 959.

The "inheritance anomaly" only occurs because such specifications tend to be rigid and fragile: change anything, and the whole specification crumbles.

At the beginning of this chapter we saw another apparent inheritance-concurrency clash; but the culprit turned out to be the notion of active object. In both cases inheritance is at odds not with concurrency but with a particular approach to concurrency (active objects, state-based specifications); rather than dismissing or limiting inheritance — cutting the arm whose finger itches — the solution is to look for better concurrency mechanisms.

One of the practical consequences of this discussion is that we should try to rely, for synchronization in concurrent computation, on what we already have in the object-oriented model, in particular assertions. Preconditions will indeed play a central role for synchronization, although we will need to adapt their semantics from the sequential case.

30.6 ACCESSING SEPARATE OBJECTS

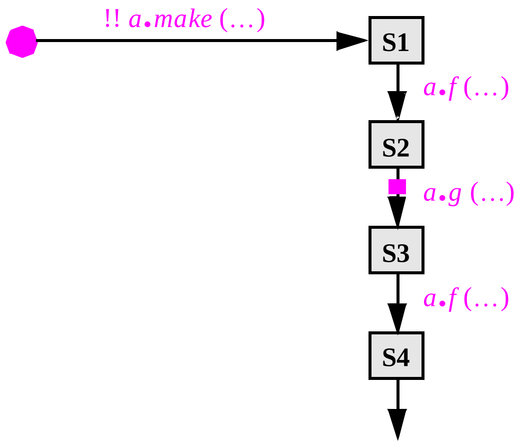
We now have enough background to devise the proper synchronization mechanisms for our concurrent object-oriented systems.

Concurrent accesses to an object

The first question to address is how many executions may proceed concurrently on an object. The answer was in fact implicit in the definition of the notions of processor and handler: if all calls to features on an object are executed by its handler (the processor in charge of it), and a processor is a single thread of execution, it follows that at most one feature may be executing on a given object at any time.

Should we not allow several routines to execute concurrently on a given object? The main incentive for answering *no* is to retain the ability to reason on our software.

The study of class correctness in an earlier chapter provides the proper perspective. We saw the lifecycle of an object pictured as this:



The life of an object

(*This figure first appeared on page 366.*)

In this view the object is externally observable only in the states marked as shaded squares: just after creation (S1), after every application of a feature by a client (S2 and subsequent states). These have been called the “stable times” of the object’s life. A consequence was the formal rule: to prove the correctness of the class, we only have to

These rules have been slightly simplified. The full versions appear in "WHEN IS A CLASS CORRECT?", 11.9, page 370.

verify one property for each creation procedure, and one property for each exported feature. If *p* is a creation procedure, the property to check is

$$\{Default \textbf{ and } pre_p\}\ \ Body_p\ \ \{post_p \textbf{ and } INV\}$$

meaning: if you execute the body of *p* when the object has been initialized to the default values and the precondition of *p* holds, you will end up satisfying the postcondition and the invariant. For an exported routine *r*, the property to check is

$$\{pre_r \textbf{ and } INV\}\ \ Body_r\ \ \{post_r \textbf{ and } INV\}$$

meaning: if you execute *r* when the precondition and the invariant are satisfied, you will end up satisfying the postcondition and the invariant.

So the number of things to check is very limited; there are no complicated run-time scenarios to analyze. This is important even in a somewhat informal approach to software development, which still requires the ability to reason about the software execution by examining the software text. The informal version of the preceding properties is that you can understand the class by looking at its routines separately from each other — convincing yourself, however informally, that each routine will deliver the intended final state starting from the expected initial state.

Introduce concurrent execution into this simple, consistent world, and all hell breaks loose. Even plain interleaving, in which we would start executing a routine, interrupt it in favor of another, switch back to the first and so on, would deprive us from any ability to use straightforward reasoning on our software texts. We simply would not have any clue as to what can happen at run-time; trying to guess would force us to examine all possible interleavings, immediately leading to a combinatorial explosion of cases to consider.

So for simplicity and consistency we will let at most one routine execute on any particular object at any particular time. Note, however, that in a case of emergency, or if a client keeps an object for too long, we should be able to **interrupt** the client, as long as we do so in a sufficiently violent way — triggering an exception — to ensure that the unfortunate client will receive a notification, enabling it to take corrective action if appropriate. The mechanism of *duels*, explained later, offers that possibility.

The end of the discussion section examines whether any circumstances would allow us to relax the prohibition of concurrent accesses to a single object.

Reserving an object

We need a way for a client to obtain exclusive access to a certain resource, represented by a certain object.

An idea which seems attractive at first (but will not suffice) would be simply to rely on the notion of separate call. Consider, executed on behalf of a certain client object O1, the call *x*.*f* (…), for separate *x* attached at run time to O2. Once the call has started executing, we have seen that O1 can safely move to its next business without waiting for the call's termination; but this execution of the call cannot start until O2 is free for O1. So we might decide that before starting the call the client will wait until the target object is free.

Unfortunately this simple scheme is not sufficient, because it does not allow the client to decide how long to retain an object. Assume O2 is some shared data structure such as a buffer, and the corresponding class provides procedure *remove* to remove an element. A client O1 may need to remove two consecutive elements, but just writing

buffer●*remove*; *buffer*●*remove*

will not do: between the two instructions, any other client can jump in and perform operations on the shared structure! So the two elements might not be adjacent.

One solution is to add to the generating class of *buffer* (or of a descendant) a procedure *remove_two* that removes two elements at once. But in the general case that is unrealistic: you cannot change your suppliers for every synchronization need of *your own client code*. There must be a way for the client to reserve a supplier object for as long as it needs, using the supplier class as it is.

In other words, we need something like a critical region mechanism. The syntax introduced earlier was

On critical regions see "Synchronization-based mechanisms", page 978.

hold *a* **then** *actions_requiring_exclusive_access* **end**

or the conditional variant

hold *a* **when** *a*●*some_property* **then** *actions_requiring_exclusive_access* **end**

We will, however, go for a simpler approach, perhaps surprising at first. The convention will simply be that if *a* is a non-void separate expression a call of the form

actions_requiring_exclusive_access (*a*)

causes the caller to wait until the object attached to *a* is available. In other words, there is no need for a **hold** instruction; to reserve a separate object, you simply use it as actual argument in a call.

Note that waiting only makes sense if the routine contains at least one call *x*●*some_routine* on the formal argument *x* corresponding to *a*. Otherwise, for example if all it does is a "business card" assignment *some_attribute* := *x*, there is no need to wait. This is specified in the full form of the rule, also involving preconditions, which appears later in this chapter.

"Separate call semantics", page 996.

Other policies are possible, and indeed some authors have proposed retaining a **hold** instruction (see the bibliographical notes). But the use of argument passing as the object reservation mechanism helps keep the concurrency model simple and easy to learn. One of the observations justifying this policy is that with the **hold** scheme shown above it will be tempting for developers, in line with the general "Encapsulate Repetition" motto of O-O development, to gather in a routine the actions that require exclusive access to an object; this trend was foreseen in the above summary of the **hold** instruction, where the actions appear as a single routine *actions_requiring_exclusive_access*. But then such a routine will need an argument representing the object; here we go further and consider that the presence of such an argument suffices to achieve object reservation.

This convention also means that, paradoxically enough, *most separate calls do not need to wait*. When we are executing the body of a routine that has a separate formal argument *a*, we know that we have already reserved the attached object, so any call with target *a* can proceed immediately. As we have seen, there is no need to wait for the call to terminate. In the general case, with a routine of the form

```
r (a: separate SOME_TYPE) is
        do
                …; a.r1 (…); …
                …; a.r2 (…); …
        end
```

an implementation can continue executing the intermediate instructions without waiting for any of the calls to terminate, as long as it logs all the calls on *a* so that they will be executed in the order requested. (We have yet to see how to wait for a separate call to terminate if that is what we want; so far, we just start calls and never wait!)

If a routine has two or more separate arguments, a client call will wait until it can reserve *all* the corresponding objects. This requirement is hard on the compiler, which will have to generate code using protocols for multiple simultaneous reservations; for that reason, an implementation might at first impose the restriction that a routine may have at most one separate formal argument. But if the full mechanism is implemented it provides considerable benefits to application developers; as a typical example, studied later in this chapter, the famous "dining philosophers" problem admits an almost trivial solution.

Accessing separate objects

The last example shows how to use, as the target of a separate call, a formal argument, itself separate, of the enclosing routine *r*. An advantage is that we do not need to worry about how to get access to the target object: this was taken care of by the call to *r*, which had to reserve the object — waiting if necessary until it is free.

We can go further and make this scheme the *only* one for separate calls:

Separate Call rule

The target of a separate call must be a formal argument of the routine in which the call appears.

Remember that a call *a.r* (…) is separate if the target *a* is itself an entity or expression declared as separate. So if we have a separate entity *a* we cannot call a feature on it unless *a* is a formal argument of the enclosing routine. If, for example, *attrib* is an attribute declared as separate, we must use, instead of *attrib.r* (…), the call *rf* (*attrib*, …) with

```
rf (x: separate SOME_TYPE; … Other arguments …) is
                -- Call r on x.
        do
                x.r (…)
        end
```

This rule may appear to place an undue burden on developers of concurrent applications, since it forces them to encapsulate all uses of separate objects in routines. It may indeed be possible to devise a variant of this chapter's model which does not include the Separate Call rule; but as you start using the model you will, I think, realize that the rule is in fact of great help. It encourages developers to identify accesses to separate objects and separate them from the rest of the computation. Most importantly, it avoids grave errors that would be almost bound to happen without it.

The following case is typical. Assume a shared data structure — such as, once again, a buffer — with features *remove* to remove an element and *count* to query the number of elements. Then it is quite "natural" to write

According to David Gries, "natural" is one of the most dangerous words in software discussions.

```
if buffer.count >= 2 then
        buffer.remove; buffer.remove
end
```

The intent is presumably to remove two elements. But, as we have already seen, this will not always work — at least not unless we have secured exclusive access to *buffer*. Otherwise between the time you test *count* and the time you execute the first *remove*, any other client can come in and remove an element, so that you will end up trying to apply *remove* to an empty structure.

Another example, assuming that we follow the style of previous chapters and include a feature *item*, side-effect-free, to return the element that *remove* removes, is

```
if not buffer.empty then
        value := buffer.item; buffer.remove
end
```

Without a protection on *buffer*, another client may add or remove an element between the calls to *item* and *remove*. If the author of the above extract thinks that the effect is to access an element and remove it, he will be right some of the time; but if this is not your lucky day you will access an element and remove another — so that you may for example (if you repeat the above scheme) access the same element twice! Very wrong.

By making *buffer* an argument of the enclosing routine, we avoid these problems: *buffer* is guaranteed to be reserved for the duration of the routine's call.

Of course the fault in the examples cited lies with the developer, who was not careful enough. But without the Separate Call rule such errors are too easy to make. What makes things really bad is that the run-time behavior is non-deterministic, since it depends on the relative speed of the clients. The bug will be intermittent, here one minute, gone the next. Worse yet, it will probably occur rarely: after all (using the first example) a competing client has to be quite lucky to squeeze in between your test of *count* and your first call to *remove*. So the bug may be very hard to reproduce and isolate.

Such tricky bugs are responsible for the nightmarish reputation of concurrent system debugging. Any rule that can significantly decrease their likelihood of occurring is a big potential help.

With the Separate Call rule you will write the examples as the following procedures, assuming a separate type *BOUNDED_BUFFER* detailed below:

```
remove_two (buffer: BOUNDED_BUFFER) is
        -- Remove oldest two items.
    do
        if buffer.count >= 2 then
            buffer.remove; buffer.remove
        end
    end

get_and_remove (buffer: BOUNDED_BUFFER) is
        -- Assign oldest item to value, and remove it.
    do
        if not buffer.empty then
            value := buffer.item; buffer.remove
        end
    end
```

These procedures may be part of some application class; preferably, they will appear in a class *BUFFER_ACCESS* which encapsulates buffer manipulation operations, and serves as parent to application classes needing to use buffers of the appropriate type.

The procedures both seem to be crying for a precondition. We will shortly see to it that they can get one.

Wait by necessity

Assume that a separate call such as *buffer.remove* has been started, after waiting if necessary for any separate arguments to become available. We have seen that from then on it does not block the client, which can proceed with the rest of its computation. But surely the client may need to resynchronize with the supplier. When should we wait for the call to terminate?

See the bibliographical notes.

It would seem that we need a special mechanism, as has indeed been proposed by some concurrent O-O languages such as Hybrid, to reunite the parent computation with its prodigal call. But instead we can use the idea of wait by necessity, due to Denis Caromel. The goal is to wait when we truly need to, but no earlier.

When does the client need to be sure that a call *a.r* (…), for separate *a* attached to a separate object O1, is finished? Not when it is doing something else on other objects, separate or not; not even necessarily when it has started a new procedure call *a.r* (…) on the same separate object since, as we have seen, a smart implementation can simply log such calls so that they will be processed in the order emitted (an essential requirement, of course); but when we need to access some property of O1. *Then* we require the object to be available, and all preceding calls on it to have been finished.

You will remember the division of features into *commands* (procedures), which perform some transformation on the target object, and *queries* (functions and attributes) which return information about it. Command calls do not need to wait, but query calls may.

Consider for example a separate stack *s* and the successive calls

```
s.put (x1); ... Other instructions ...; s.put (x2); ... Other instructions ...; value := s.item
```

(which because of the Separate Call rule must appear in a routine of which *s* is a formal argument). Assuming none of the Other instructions uses *s*, the only one that requires us to wait is the last instruction since it needs some information about the stack, its top value (which in this case should be *x2*).

These observations yield the basic concept of wait by necessity: once a separate call has started, a client only needs to wait for its termination if the call is to a query. A more precise rule will be given below, after we look at a practical example.

Wait by necessity (also called "lazy wait", and similar to mechanisms of "call by necessity" and "lazy evaluation" familiar to Lispers and students of theoretical computing science) is a convenient rule which allows you to start parallel computations as you need and avoid unnecessary waiting, but be reassured that the computation *will* wait when it must.

A multi-launcher

Here is a typical example showing the benefits of wait by necessity. Assume that a certain object must create a set of other objects, each of which goes off on its own:

```
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 := a.lower until i > a.upper 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 dining philosophers", page 1003.

If, as may well be the case, procedure *live* of class *X* describes an infinite process, this scheme relies on the guarantee that each loop iteration will proceed immediately after starting *launch_one*, without waiting for the call to terminate: otherwise the loop would never get beyond its first iteration. One of the examples below uses this scheme.

"Coroutine concepts", page 1118.

Readers familiar with coroutine-based discrete event simulation, studied in a later chapter, will recognize a scheme very close to what happens when you start a simulated process and want to gain control back, as permitted by Simula's **detach** instruction.

An optimization

(This section examines a fine point and may be skipped on first reading.)

To wrap up this discussion of wait by necessity we need to examine more carefully when a client should wait for a separate call to terminate.

We have seen that only query calls should cause waiting. But we may go further by examining whether the query's result is of an expanded type or a reference type. (For the *s*.*item* example, assuming *s* of type *STACK* [*SOME_TYPE*], this is determined by *SOME_TYPE*.) If the type is expanded, for example if it is *INTEGER* or another of the basic types, there is no choice: we need the value, so the client computation must wait until the query has computed its result. But for a reference type, one can imagine that a smart implementation could still proceed while the result, a separate object, is being computed; in particular, if the implementation uses proxies for separate objects, the proxy object itself can be created immediately, so that the reference to it is available even if the proxy does not yet refer to the desired separate object.

This optimization, however, complicates the concurrency mechanism because it means proxies must have a "ready or not" boolean attribute, and all operations on separate references must wait until the proxy is ready. It also seems to prescribe a particular implementation — through proxies. So we will not retain it as part of the basic rule:

Wait by necessity

If a client has started one or more calls on a certain separate object, and it executes on that object a call to a query, that call will only proceed after all the earlier ones have been completed, and any further client operations will wait for the query call to terminate.

To account for the possible optimization just discussed, replace "*a call to a query*" by "*a call to a query returning of expanded type*".

Avoiding deadlock

Along with several typical and important examples of passing separate references to separate calls, we have seen that it is also possible to pass non-separate references, as long as the corresponding formal arguments are declared as separate (since, on the supplier's side, they represent foreign objects, and we do not want any traitors). Non-separate references raise a risk of deadlock and must be handled carefully.

The normal way of passing non-separate references is what we have called the *business card* scheme: we use a separate call of the form *x*.*f* (*a*) where *x* is separate but *a* is not; that is to say, *a* is a reference to a local object of the client, possibly *Current* itself; on the supplier side, *f* is of the form

```
f (u: separate SOME_TYPE) is
        do
                local_reference := u
        end
```

where *local_reference*, also of type **separate** *SOME_TYPE*, is an attribute of the enclosing supplier class. Later on, in routines other than *f*, the supplier may use *local_reference* to request operations on objects on the original client's side, through separate calls of the form *local_reference*.*some_routine* (…)

This scheme is sound. Assume, however, that *f* did more, for example that it included a call of the form *u*.*g* (…) for some *g*. This is likely to produce deadlock: the client (the handler for the object attached to *u* and *a*) is busy executing *f* or, with wait by necessity, may be executing another call that has reserved the same object.

The following rule will avoid this kind of situation:

Business Card principle

If a separate call uses a non-separate actual argument of a reference type, the routine should only use the corresponding formal as source of assignments.

At present this is a only methodological guideline although it may be desirable to introduce a formal validity rule (an exercise asks you to explore this idea further.) Some more comments on deadlocks appear in the discussion section.

Exercise E30.4, page 1035; see also E30.13, page 1036.

30.7 WAIT CONDITIONS

One synchronization rule remains to be seen. It will deal with two questions at once:

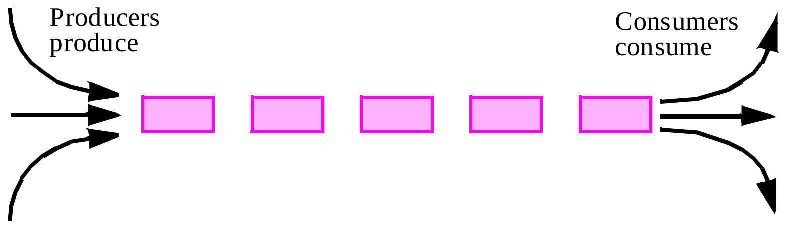
- How can we make a client wait until a certain condition is satisfied, as in conditional critical regions?
- What is the meaning of assertions, in particular preconditions, in a concurrent context?

A buffer is a separate queue

We need a working example. To study what happens to assertions, it is interesting to take a closer look at a notion that is ubiquitous in concurrent application (and has already appeared informally several times in this chapter): **bounded buffers**. A bounded buffer, illustrated by the top figure on the facing page, allows different components of a concurrent system to exchange data, produced by some and consumed by others, without forcing each producer that has generated an object to wait until a consumer is ready to use it, and conversely. Instead, communication occurs through a shared structure, the buffer; producers deposit their wares into the buffer, and consumers get their material from it. In a bounded implementation the structure can only hold a certain number *maxcount* of items, and so it can get full. But waits will only occur when a consumer needs to consume and the buffer is empty, or when a producer needs to produce and the buffer is full. In a well-regulated system such events will be much more infrequent than with unbuffered communication, and their frequency will decrease as the buffer's capacity grows. True, a new source of delays arises because buffer access must be exclusive: at most one client may at any one time be performing a deposit (*put*) or retrieval (*item*, *remove*) operation. But these are very simple and fast operations, so any resulting wait is typically short.

In most cases the time sequence in which objects have been produced is relevant to the consumers, so the buffer must maintain a **first-in**, **first-out** policy (FIFO): an object deposited before another must be retrieved before it. The behavior is similar to that of train cars being added at one end of a single track and removed at the other end:

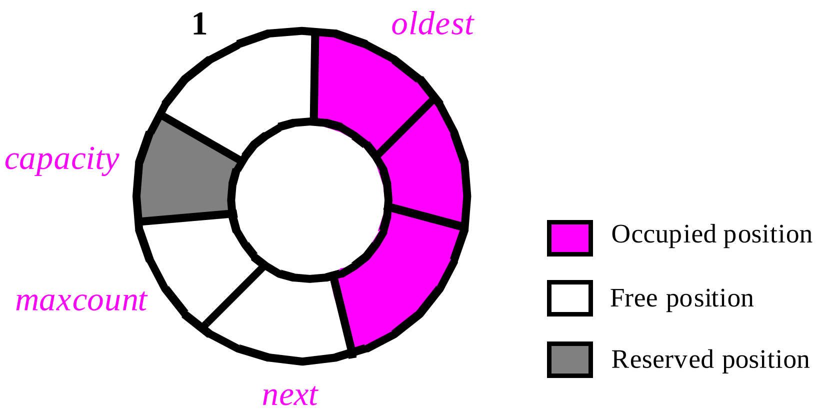
Bounded buffer



The bounded queue of the Undoing design pattern used a similar representation. See page 710.

A typical implementation — not essential to the discussion, but giving us a more concrete view — can use an array *representation* of size *capacity* = *maxcount* + *1*, managed circularly; the integer *oldest* will be the index of the oldest item, and *next* the index of the position to be used for inserting the next item that comes in. We can picture the array as being torn into a ring so that positions 1 and *capacity* are conceptually adjacent:

Bounded buffer implemented by an array



The procedure *put* used by a producer to add an item *x* will be implemented as

representation.*put* (*x*, *next*); *next* := (*next* \\ *maxcount*) + *1*

where \\ is the integer remainder operation; the query *item* used by consumers to obtain the oldest element simply returns *representation* @ *oldest* (the array element at index *oldest*); and procedure *remove* simply executes *oldest*:= (*oldest* \\ *maxcount*) + *1*. The array entry at index *capacity*, shaded in gray on the figure, is kept free; this makes it possible to distinguish between the test for *empty*, expressed as *next = oldest*, and the test for *full*, expressed as *(next \\ maxcount) + 1 = oldest*.

For an alternative technique see "Unconstrained genericity", page 1181.

The structure, with its FIFO policy, and the circular array representation, are of course not concurrency-specific: what we have is simply a **bounded queue** similar to many of the structures studied in preceding chapters. Writing the corresponding class — directly applicable to the Undoing design pattern — is not hard; here is a short form of the class, in simplified form (main features only, header comments removed, principal assertion clauses only):

```
class interface BOUNDED_QUEUE [G] feature
    empty, full: BOOLEAN
    put (x: G)
        require
            not full
        ensure
            not empty
    remove
        require
            not empty
        ensure
            not full
    item: G
        require
            not empty
end -- class interface BOUNDED_QUEUE
```

Obtaining from this description a class describing bounded buffers is about as simple as we could dream:

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

The **separate** qualifier applies only to the class where it appears, not its heirs. So a separate class may, as here, inherit from a non-separate one, and conversely. The convention is the same as with the other two qualifiers applicable to a class: **expanded** and **deferred**. As noted, the three properties are mutually exclusive, so that at most one of the qualifiers may appear before the keyword **class**.

We see once again the fundamental simplicity of concurrent O-O software development, and the smooth transition from sequential to concurrent concepts, made possible in particular by the method's focus on encapsulation. A bounded buffer (a notion for which you will find many complicated descriptions if you look at the concurrency literature) is nothing else than a bounded queue made separate.

Preconditions under concurrent execution

Let us examine a typical use of a bounded buffer *buffer* by a client, for example a producer that needs to deposit a certain object *y* using the procedure *put*. Assume that *buffer* is an attribute of the enclosing class, having been declared, for some type *T* which is also the type of *y*, as *buffer*: *BOUNDED_BUFFER* [*T*].

The client may for example have initialized *buffer* to a reference to the actual buffer passed by its creation procedure, using the *business card* scheme suggested earlier:

```
make (b: BOUNDED_BUFFER [T], …) is do …; buffer := b; … end
```

Because *buffer*, being declared of a separate type, is a separate entity, any call of the form *buffer*.*put* (*y*) is a separate call and has to appear in a routine of which *buffer* is an argument. So we should instead use *put* (*buffer*, *y*) where *put* (a routine of the client class, not to be confused with the *put* of *BOUNDED_BUFFER*, which it calls) is declared as

```
put (b: BOUNDED_BUFFER [T]; x: T) is
        -- Insert x into b. (First attempt.)
    do
        b.put (x)
    end
```

Well, this is not quite right. Procedure *put* of *BOUNDED_BUFFER* has a precondition, **not** *full*. Since it does not make sense to try to insert *x* into *b* if *b* is full, we should mimic this precondition for our new procedure in the client class:

```
put (b: BOUNDED_BUFFER [T]; x: T) is
        -- Insert x into b.
    require
        not b.full
    do
        b.put (x)
    end
```

Better. How can we call this procedure with a specific *buffer* and *y*? We must make sure, of course, that the precondition is satisfied on input. One way is to test:

```
if not full (buffer) then put (buffer, y)        -- [PUT1]
```

but we could also rely on the context of the call as in

```
remove (buffer); put (buffer, y)                 -- [PUT2]
```

where the postcondition of *remove* includes **not** *full*. (Example PUT2 assumes that its initial state satisfies the appropriate precondition, **not** *empty*, for *remove* itself.)

Is this going to work? The answer, disappointing in light of the earlier comments about the unpredictability of bugs in concurrent systems, is *maybe*. Between the test for *full* and the call for *put* in the PUT1 variant, or between *remove* and *put* in PUT2, any other client may have interfered and made the buffer full again. This is the same flaw that required us, earlier on, to provide an object reservation mechanism through encapsulation.

We could try encapsulation again by writing PUT1 or PUT2 as a procedure to which *buffer* will be passed as argument, giving for PUT1:

```
put_if_possible (b: BOUNDED_BUFFER [T]; x: T) is
            -- Insert x into b if possible; otherwise set was_full to true.
      do
            if b.full then was_full:= True else
                  put (b, x); was_full := False
            end
      end
```

But this does not really help me as a client. First, having to check *was_ full* on return is a nuisance; then, what do I do if it is true? Try again, probably — but with no more guarantee of result. What I probably want is a way to execute *put* when the buffer is indisputably non-full, even if I have to **wait** for this to be the case.

The precondition paradox

This situation that we have just uncovered is disturbing because it seems to invalidate, in a concurrent context, the basic methodological guideline for getting software right: Design by Contract. With a queue, that is to say in sequential computation, we have been used to precisely defined specifications of mutual obligations and benefits:

put	**OBLIGATIONS**	**BENEFITS**
Client	(***Satisfy precondition***:) Only call *put* (*x*) on a non-full queue.	(***From postcondition***:) Get new, non-empty queue with *x* added.
Supplier	(***Satisfy postcondition***:) Update queue to add *x* and ensure **not** *empty*.	(***From precondition***:) Processing protected by assumption that queue not full.

A contract**: **routine** put **for bounded queues

(From the example for stacks on page 343.)

Implicit behind such contracts is a **no hidden clause** principle: the precondition is the only requirement that a client must satisfy to get served. If you call *put* with a non-full queue, you are entitled to the routine's result, as expressed by the postcondition.

But in a concurrent context, with a separate supplier such as a *BOUNDED_BUFFER*, things are rather distressing for the client: however hard we try to please the supplier by ensuring its stated precondition, we can never be sure to meet its expectations! To execute correctly, however, the suppliers still need the precondition. For example the body of routine *put* in class *BOUNDED_QUEUE* (which is the same as in *BOUNDED_BUFFER*) will most likely not work unless *full* is guaranteed to be false.

To summarize: suppliers cannot do their work without the guarantee that the precondition holds; but for separate arguments the clients are *unable* to ensure these preconditions. This may be called the **concurrent precondition paradox**.

There is a similar *postcondition* paradox: on return from a separate call to *put*, we cannot any more be sure that **not** *empty* and other postcondition clauses hold for the client. These properties are satisfied just after the routine's termination; but some other client may invalidate them before the caller gets restarted. Because the problem is even more serious for preconditions, which determine the correct execution of suppliers, the rest of the discussion mainly considers preconditions.

The paradoxes only arise for separate formal arguments. For a non-separate argument — in particular for an expanded value such as an integer — we can continue to rely on the usual properties of assertions. But this not much consolation.

Although this has not yet been widely recognized in the literature, the concurrent precondition paradox is one of the central issues of concurrent O-O software construction, and the futility of trying to retain habitual assertion semantics is one of the principal factors distinguishing concurrent computation from its sequential variants.

Exercise E30.6, page 1036.

The precondition paradox may also arise in situations that are not ordinarily thought of as involving concurrency, such as accessing a file. This is explored in an exercise.

The concurrent semantics of preconditions

To resolve the concurrent precondition paradox we assess the situation through three observations:

A1 • Suppliers need the preconditions to protect their routine bodies. Here *put* will never work, in class *BOUNDED_BUFFER* as in *BOUNDED_QUEUE*, unless the routine has the guarantee that on entry the queue is non-full.

A2 • Separate clients cannot rely any more on the usual (sequential) semantics of preconditions. Testing for *full* before calling your buffer supplier gives you no guarantee at all.

A3 • Because each client may be vying with others for resource access, a client may be prepared to wait before it gets its resources — if this guarantees correct processing after the wait.

The conclusion seems inescapable: we still need preconditions, if only for the suppliers' sake, but they must be given a different semantics. Instead of being a *correctness condition*, as in the sequential context, a precondition applying to a separate argument will be a **wait condition**. This will apply to what we may call "separate precondition clauses": any precondition clause involving a call whose target is a separate argument. A typical separate precondition clause is **not** *b*.*full* for *put*.

Here is the rule:

Separate call semantics

Before it can start executing the routine's body, a separate call must wait until every blocking object is free and every separate precondition clause is satisfied.

In this definition, an object is *blocking* if it is attached to an actual argument, and the routine uses the corresponding formal as the target of at least one call.

A separate object is free if it is not being used as an actual argument of a separate call (implying that no routine is being executed on it).

The rule only causes waiting for separate arguments appearing as call targets somewhere in the routine's body (it uses the word "blocking" for the corresponding objects since they can block the call from proceeding). With a routine of the "business card" form

```
r (x: separate SOME_TYPE) is do some_attribute := x end
```

or some other scheme that does not contain a call of the form *x*.*some_routine*, there is no need to wait on the actual argument corresponding to *x*.

If there is such a call the short form of the class must reflect it for the benefit of client authors. It will present the routine header as *r* (*x*: **blocking** *SOME_TYPE*)…

With this rule the above version of *put* in a client class achieves the desired result:

```
put (b: BOUNDED_BUFFER [T]; x: T) is
        require
                not b.full
        do
                b.put (x)
        ensure
                not b.empty
        end
```

A call of the form *put* (*buffer*, *y*), from a producer client, will wait until *buffer* is free (available) and not full. If *buffer* is free but full, the call cannot be satisfied; but some other client, a consumer, may get access to it (since the precondition of interest to consumers, **not** *b*.*empty*, will be satisfied in this case); after such a client has removed an item, making the buffer non-full, the producer client can now have its call executed.

Which client should the implementation let through if two or more satisfy the conditions of the rule (blocking objects free, preconditions satisfied)? Some people, for fear of overspecifying, prefer to leave such decisions to the compiler, while providing library features allowing an application to specify a particular policy. It seems better to define a default first-in-first-out policy, which enhances portability and helps towards solving the issue of fairness. Library mechanisms can still be available to application writers who wish to override the default.

Be sure to note that the special semantics of preconditions as wait conditions only applies to what we have called separate precondition clauses, that is to say, clauses involving a condition of the form *b*.*some_property* where *b* is a separate argument. A non-separate clause, such as *i* > = *0* where *i* is an integer, or *b* /= *Void* even if *b* is separate (this does not involve a call on *b*), will keep its usual correctness semantics since the concurrent precondition paradox does not apply in such cases: if the client ensures the stated condition before the call, it will still hold when the routine starts; if the condition does not hold, no amount of waiting would change the situation.

Assertions, sequential and concurrent

The idea that assertions, and in particular preconditions, may have two different semantics — sometimes correctness conditions, sometimes wait conditions — may have surprised you. But there is no way around it: the sequential semantics is inapplicable in the case of separate precondition clauses.

See "Monitoring assertions at run time", page 393.

One possible objection must be answered. We have seen that a mere compilation switch can turn run-time assertion checking on or off. Is it not dangerous, then, to attach that much semantic importance to preconditions in concurrent object-oriented systems? No, it is not. The assertions are an integral part of the software, whether or not they are enabled at run time. Because in a correct sequential system the assertions will always hold, we may turn off assertion checking for efficiency if we think we have removed all the bugs; but conceptually the assertions are still there. With concurrency the only difference is that certain assertions — the separate precondition clauses — may be violated at run time even for a correct system, and serve as wait conditions. So the assertion monitoring options must not apply to these clauses.

A validity constraint

To avert deadlock situations, we need to impose a validity constraint on precondition and postcondition clauses. Assume we permitted routines 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. Nothing in this example, save *separate_attribute*, need be separate. 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.

This is not acceptable and is prohibited by the following rule:

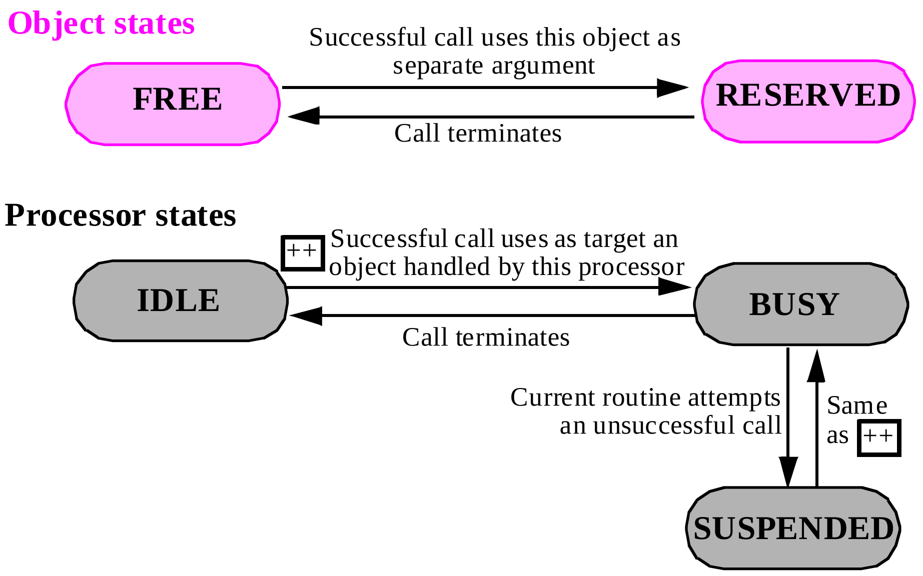
As a consequence, the assertion may not appear in a class invariant, which is not part of a routine.

Assertion Argument rule

If an assertion contains a function call, any actual argument of that call must, if separate, be a formal argument of the enclosing routine.

States and transitions

The following figure summarizes some of the preceding discussion by showing the various possible states in which objects and processors may be, and how they will change state as a result of calls.


Object states
Successful call uses this object as separate argument
FREE
RESERVED
Call terminates
Processor states
++
Successful call uses as target an object handled by this processor
IDLE
BUSY
Call terminates
Current routine attempts an unsuccessful call
Same as ++
SUSPENDED

Object and processor states and transitions

A call is *successful* if the handler of its target is idle or suspended, all its non-void separate arguments are attached to free objects, and the corresponding separate precondition clauses, if any, are true. Note that this makes the definitions of object and processor states mutually dependent.

30.8 REQUESTING SPECIAL SERVICE

We have completed the review of the basic communication and synchronization policy. For more flexibility, it is useful to define a few mechanisms that will allow interrupting the normal processing in some cases.

Because these facilities are add-ons intended for convenience, rather than a part of the basic concurrency model, they are available not as language constructs but as library features. We will assume a class *CONCURRENCY*, which classes needing these special mechanisms can inherit. A similar approach has already been used twice in this book:

- To complement the basic exception handling rules when finer control is desired, through the library class *EXCEPTIONS*.
- To complement the default memory management and garbage collection mechanism when finer control is desired, through the library class *MEMORY*.

"ADVANCED EXCEPTION HANDLING", 12.6, page 431; "Class MEMORY", page 309.

Express messages

[Yonezawa 1987a].

The ABCL/1 concurrent language introduced the notion of "express message" for when we want to let a supplier object serve a certain VIP client immediately, even though the supplier may be busy with another client.

See "Concurrent accesses to an object", page 982.

The comments on static binding were on page 514.

In some approaches an express message will just interrupt the normal message, get serviced, and then let the normal message be resumed. But this is unacceptable, as we saw earlier in this chapter when we found out that at most one execution should be active on any object at any given time: the express message, like any exported feature, needs an initial state satisfying the invariant; but who knows in what state the interrupted routine will be when it is forced to yield to the express message? And who knows what state the express message will produce as a result? All this opens the way to what the discussion of static binding called "*one of the worst events that could occur during the execution of a software system*": producing an inconsistent object. As we saw then: "*if such a situation can arise, we can no longer hope to predict what execution will do*".

See also "Permitting concurrent access?", page 1031.

This does not mean, however, that we should reject the notion of express message altogether. We may indeed need to interrupt a client — either because we have something more important to do with the object it has reserved, or because it is overextending its welcome to retain it. But such an interruption is not a polite request to step aside for a while. It is *murder*, at least attempted murder. To take our rival's place we shoot at it, so that it will die unless it can recover in the hospital. In software terms, the interrupting client must cause an **exception** in its rival, which will either retry (the hospital) or fail.

Such behavior, however, assumes that the challenger is somehow stronger than the holder. If not, the one that will get an exception is the challenger.

Duels and their semantics

The almost inescapable metaphor suggests that instead of the "express message" terminology we talk about the attempt to snatch a shared object from its current holder as a *duel* (the result, in an earlier era, of trying to snatch away someone's legitimate spouse). An object has executed the instruction

r (*b*)

where *b* is separate. After possibly waiting for the object of its desires, *b*, to become free, and for separate precondition clauses to hold, it has captured *b*, becoming its current *holder*. The execution of *r* on *b* has started on behalf of the holder, but is not finished. Another separate object, the *challenger*, executes

s (*c*)

where *c*, also separate, is attached to the same object as the holder's *b*. Normally, the challenger will wait until the call to *r* is over. What if the challenger is impatient?

Through procedures in class *CONCURRENCY* we can provide the necessary flexibility. On the holder's side we have *yield*, which means: "I am willing to release my hold if someone more worthy comes along". Most holders, of course, are not so accommodating: unless it makes an explicit call to *yield*, a holder will retain its hold. To return to this default behavior, you may use the procedure *retain.*

On the challenger's side we can use two kinds of request to get special treatment:

- *demand* means "now or never!". If you cannot immediately capture the object of your dreams (that is to say, if the holder has not called *yield*), you will get an exception. (This is the old suicide threat trick, as in *Così fan tutte*.)
- *insist* is more gentle: you try to interrupt the holder's routine, but if that is impossible you accept the common lot — waiting until the object is freed.

To return to the default behavior of waiting for the holder to finish, use *wait_turn*.

Exercise E30.5, page 1035, asks you to add priorities.

A call to one of these *CONCURRENCY* procedures will retain its effect until another supersedes it. Note that the two sets of facilities are not exclusive; for example a challenger could use both *insist* to request special treatment and *yield* to accept being interrupted by another. A priority scheme can be added, so that challengers will only defer to others with higher priorities, but we can ignore this refinement here.

The following table shows the result of a duel — a conflict between a holder and a challenger — in all possible cases. The default options and behavior, in the absence of any call to *CONCURRENCY* procedures, are underlined.

The semantics of duels

Challenger → **↓ Holder**	<u>*wait_turn*</u>	*demand*	*insist*
<u>*retain*</u>	<u>Challenger waits</u>	Exception in challenger	Challenger waits
yield	Challenger waits	Exception in holder's routine.	Exception in holder's routine.

The "holder's routine" that gets an exception in the two rightmost bottom entries is the supplier routine being executed on behalf of the holder. In the absence of a **retry**, it will pass on the exception to the holder, and the challenger will get the object.

As you will remember, every kind of exception has a code, accessible through class *EXCEPTIONS*. To distinguish an exception caused by one of the situations appearing in the above table, *EXCEPTIONS* provides the boolean query *is_concurrency_interrupt*.

Interrupt handling: the Secretary-Receptionist Algorithm

Here is an example using duels. Assume a certain controller object has started off a number of partner objects, and then proceeds with its own work, which needs a certain resource *shared*. But the other objects may need access to the shared resource, and the controller is willing to interrupt its current task to let any of them proceed; when the partner is done, the controller resumes the last interrupted task.

This general description covers among others the case of an operating system kernel (the controller) which starts off input-output processors (the partners), but does not wait for an I/O operation to complete, since I/O is typically several orders of magnitude slower than computation. When an I/O operation terminates, its processor can interrupt the kernel to request attention. This is the traditional interrupt-driven scheme for handling I/O — and the problem which gave the original impetus, many years ago, to the study of concurrency.

The general scheme may be called the *Secretary-Receptionist Algorithm* by analogy with what you find in many organizations: a receptionist sits near the entrance to greet, register and direct visitors, but this is not a full-time job; the receptionist is also entrusted with some other work, usually secretarial. When a visitor shows up, the receptionist interrupts his work, takes care of the visitor, and then goes back to the interrupted task.

Restarting a task after it has been started and interrupted may require some cleanup; this is why the following procedure passes to *operate* the value of *interrupted*, which will enable *operate* to find out whether the current task has already been attempted. The first argument of *operate*, here *next*, identifies the task to perform. The procedure is assumed to be part of a class that inherits from both *CONCURRENCY* (for *yield* and *retain*) and *EXCEPTIONS* (for *is_concurrency_interrupt*). Procedure *operate* could take a long time to execute, and so is the interruptible part.

```
execute_interruptibly is
            -- Perform own set of actions, but take interrupts
            -- (the Secretary-Receptionist Algorithm).
      local
            done, next: INTEGER; interrupted: BOOLEAN
      do
            from done := 0 until termination_criterion loop
                  if interrupted then
                        process_interruption (shared); interrupted := False
                  else
                        next := done + 1; yield
                        operate (next, shared, interrupted)-- This is the interruptible part.
                        retain; done := next
                  end
            end
      rescue
            if is_concurrency_interrupt then
                  interrupted := True; retry
            end
      end
```

Some of the steps performed by the controller may actually have been requested by one of the interrupting partners. In an I/O interrupt, for example, the I/O processor will signal the end of an operation and (in the input case) the availability of the data just read.

The interrupting partner may use the object *shared* to deposit that information; to interrupt the controller, it will execute

```
insist; interrupt (shared); wait_turn
                              -- Request controller's attention, interrupting it if necessary.
                              -- Deposit any needed information into the object shared.
```

This is the reason why *process_interruption*, like *operate*, uses *shared* as argument: it may have to analyze the *shared* object to detect information passed by the interrupting partner. This will allow it, if necessary, to set up one of its upcoming tasks, to be executed on behalf of that partner. Note that *process_interruption*, unlike *operate*, is not interruptible; any other partner that becomes ready while it is executing will have to wait (otherwise some partner requests might get lost). So *process_interruption* should only perform simple operations — registering information for future processing. If that is not possible, you may use a slightly different scheme in which *process_interruption* relies on a separate object other than *shared*.

We have one more precaution to take. Although partners' requests can be processed later (through calls to *operate* in upcoming steps), it is essential that none of these requests be lost. With the scheme as given, after a partner executes an *interrupt*, another one could do the same, overriding the information deposited by the first, before the controller has had the time to register that information by executing *process_interruption*. This case is not acceptable. To avoid it, we can just add to the generating class of *shared* a boolean attribute *deposited* with the associated setting and resetting procedures. Then *interrupt* will have the precondition **not** *shared*.*deposited*, so as to wait until the previous partner has been registered, and will execute the call *shared*.*set_deposited* before returning; *process_interruption* will execute *shared*.*set_not_deposited* before exiting.

The partners are initialized by "business card" calls of the form !! *partner*.*make* (*shared*, ...) which pass them a reference to *shared* to be retained for future needs.

Procedure *execute_interruptibly* has been spelled out in full, with the application-specific elements represented by calls to routines *operate*, *process_interruption*, *termination_criterion* that are assumed to be deferred, in the behavior class style. This prepares for the procedure's possible inclusion into a concurrency library.

About the rest of this chapter

With the presentation of the duel mechanism we have finished defining the set of necessary concurrency tools. The rest of this chapter provides an extensive set of examples, from diverse application areas, illustrating the use of these tools. After the examples you will find:

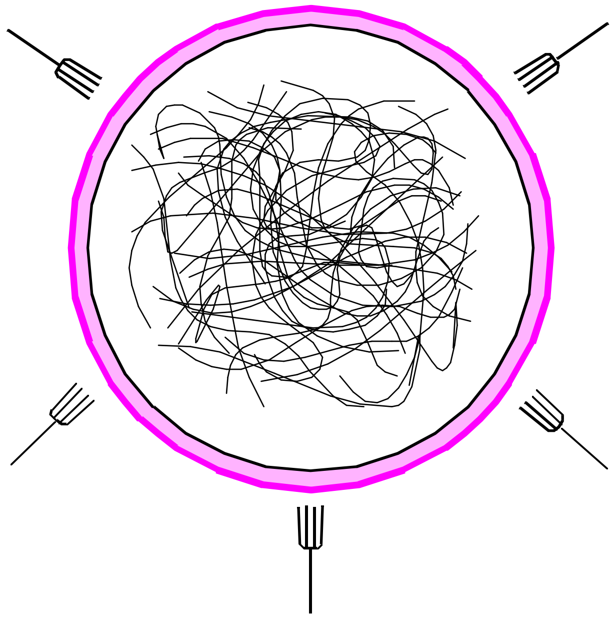
- A sketch of a proof rule, for mathematically-inclined readers. *30.10, page 1022*.
- A summary of the concurrency mechanism, with syntax, validity rules and semantics. *30.11, page 1025*.
- A discussion of the mechanism's goals and of further work needed. *30.12, page 1028*.
- A detailed bibliography of other work in this area. *30.14, page 1033*.

30.9 EXAMPLES

To illustrate the mechanism, here now are a few examples chosen from diverse backgrounds — from traditional concurrent programming examples through large-scale multiprocessing to real-time applications.

The dining philosophers

The philosophers' spaghetti plate

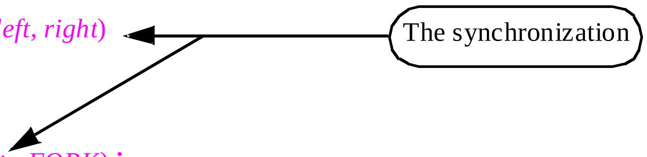


Dijkstra's famous "dining philosophers", an artificial example meant to illustrate the behavior of operating system processes vying for shared resources, is an obligatory part of any discussion on concurrency. Five philosophers around a table spend their time thinking, then eating, then thinking again and so on. To eat the spaghetti, each needs access to the fork immediately to his left and to his right — creating contention and possible deadlock.

The class is definitely ***not*** *what people mean by "spaghetti code".*

The following class describes the philosopher's behavior. Thanks to the mechanism for reserving objects through separate arguments, there is essentially (in contrast with the usual solutions in the literature) no explicit synchronization code:

```
separate class PHILOSOPHER creation
        make
inherit
        GENERAL_PHILOSOPHER
        PROCESS
                rename setup as getup undefine getup end
feature {BUTLER}
        step is
                        -- Perform a philosopher's tasks.
                do
                        think
                        eat (left, right)
                end
feature {NONE}
        eat (l, r: separate FORK) is
                        -- Eat, having grabbed l and r.
                do … end
end -- class PHILOSOPHER
```



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.

The simplicity of this solution comes from the mechanism's ability to reserve several resources through a single call having several 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 many published algorithms for getting hold of two forks one after the other without causing deadlock.

The principal procedure of class *PHILOSOPHER* does not appear above since it comes from the behavior class *PROCESS*: procedure *live*, which as given in *PROCESS* simply executes **from** *setup* **until** *over* **loop** *step* **end**, so all we need to redefine here is *step*. I hope you will enjoy the renaming of *setup* as *getup* — denoting the philosopher's initial operation.

Class PROCESS appeared on page 961. wrapup remains an empty

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.

To avoid confusion of genres the concurrency-independent features of a philosopher have been kept in a class *GENERAL_PHILOSOPHER*:

```
class GENERAL_PHILOSOPHER creation
    make
feature -- Initialization

    make (l, r: separate FORK) is
            -- Define l as left and r as right forks.
        do
            left := l; right := r
        end
feature {NONE} -- Implementation

    left, right: separate FORK
            -- The two required forks

    getup is
            -- Take any necessary initialization action.
        do … end

    think is
            -- Any appropriate action or lack thereof.
        do … end
end -- class GENERAL_PHILOSOPHER
```

The rest of the system simply takes care of initialization and of describing the auxiliary abstractions. 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

    count: INTEGER
        -- The number of both philosophers and forks

    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} -- Initialization
    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
                p := philosophers @ i
                left := cutlery @ i
                right := cutlery @ ((i \\ count) + 1
                !! p.make (left, right)
                i := i + 1
            end
        end
invariant
    count >= 0; participants.count = count; cutlery.count = count
end
```

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

See "A multi-launcher", page 988.

Making full use of hardware parallelism

The following example illustrates how to use wait by necessity to draw the maximum benefit from any available hardware parallelism. It shows a sophisticated form of *load balancing* in which we offload computation to many different computers on a network. Thanks to the notion of processor, we can rely on the concurrency mechanism to choose these computers automatically for us.

The example itself — computing the number of nodes in a binary tree — is of little practical value, but illustrates a general scheme that may be extremely useful for large, heavy computations such as those encountered in cryptography or advanced computer graphics, for which developers need all the resources they can get, but do not want to have to take care manually of the assignment of abstract computing units to actual computers.

Consider first a class extract that does not involve concurrency:

```
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 -- class BINARY_TREE
```

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

In a concurrent environment offering many processors, 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_TREE1 [G] feature
    left, right: BINARY_TREE1 [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_TREE1 [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_TREE1 [G]) is
                -- Adjust number of nodes from those in b.
        do
                if b /= Void then nodes := nodes + b.nodes end
        end
end -- class BINARY_TREE1
```

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.

On how to specify the mapping see "Mapping the processors: the Concurrency Control File", page 971.

An attractive property of the solution is that it ignores the practical problem of assigning the actual computers. The software just allocates processors as it needs to. (This is done in the creation instructions, not shown, which will appear in particular in the insertion procedure: to insert a new element into a binary tree you create a new node through !! *new_node*.*make* (*new_element*) which here, *new_node* being of the separate type *BINARY_TREE1*[*G*], will allocate a new processor to it.) The mapping of these virtual processors to the available physical resources is entirely automatic.

Locks

Assume you 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. This will provide us with a semaphore-like mechanism. 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

	invariant
		locked_iff_holder: locked = (holder /= Void)
	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; this problem was mentioned in the discussion of semaphores as one of the major limitations of this technique. But 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 behavior class:

```
	deferred class LOCKING_PROCESS 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 -- class LOCKING_PROCESS
```

Exercise E30.7, page 1036.

An effective descendant of *LOCKING_PROCESS* will effect *exclusive_actions* and *over*, and may redefine *setup* and *finalize*. Note that it is desirable to write *LOCKING_PROCESS* as a descendant of *PROCESS*.

Whether or not we go through *LOCKING_PROCESS*, a *grab* does not take away the corresponding lockable from all possible clients: it only excludes other lockers that observe the protocol. To exclude any client from accessing a resource, you must enclose the operations accessing the resource in a routine to which you pass it as an argument.

Routine *grab* of class *LOCKER* is an example of what has been called the business card scheme: passing to *resource* a reference to the *Current* locker, which the resource will keep as a separate reference.

Exercise E30.7, page 1036.

Based on the pattern provided by these classes, it is not difficult to write others implementing semaphores under their various forms. Object-oriented mechanisms help us help users of our classes avoid the classic danger of semaphores: executing a *reserve* on a resource and forgetting to execute the corresponding *free*. A developer using a behavior class such as *LOCKING_PROCESS* will fill in the deferred operations to cover the needs of his application, and can rely on the predefined general scheme to guarantee that each *reserve* will be properly followed by the corresponding *free*.

Coroutines

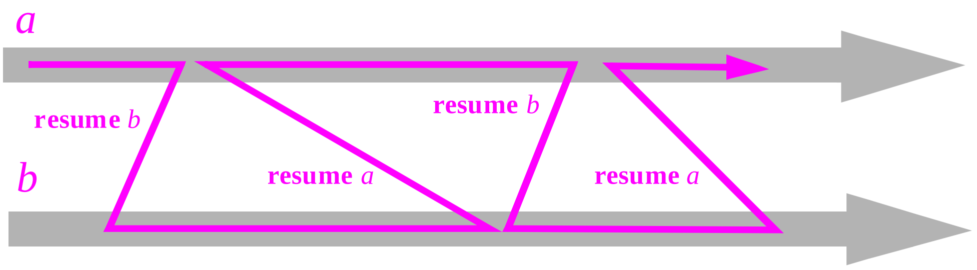
Although not truly concurrent, at least not in its basic form, our next example is essential as a way to test the general applicability of a concurrent mechanism.

The first (and probably the only) major programming language to include a coroutine construct was also the first object-oriented language, Simula 67; we will study its coroutine mechanism as part of the presentation of Simula. That discussion will also present some examples of the practical use of coroutines.

"Coroutine concepts", page 1118 (in the Simula chapter). The **resume** *instruction comes from Simula.*

Coroutines emulate concurrency on a sequential computer. They provide a form of program unit that, although similar to the traditional notion of routine, reflects a more symmetric form of communication:

- With a routine call, there is a master and a slave; the caller starts a routine, waits for its termination, and picks up where it left; the routine, however, always starts from the beginning. The caller *calls*; the routine *returns*.
- With coroutines, the relationship is between peers; coroutine *a* gets stuck in its work and calls coroutine *b* for help; *b* restarts where it last left, and continues until it is its turn to get stuck or it has proceeded as far as needed for the moment; then *a* picks up its computation. Instead of separate call and return mechanisms, there is a single operation **resume** *c*, meaning: restart coroutine *c* where it was last interrupted; I will wait until someone else **resume**s me.



Coroutine sequencing

This is all strictly sequential and meant to be executed on a single process (task) of a single computer. But the ideas are clearly drawn from concurrent computation; in fact an operating system running on a single CPU will internally use a coroutine-like mechanism to implement such schemes as time-sharing, multitasking and multithreading.

Coroutines may be viewed as a boundary case of concurrency: the poor man's substitute to concurrent computation when only one thread of control is available. It is always a good idea to check that a general-purpose mechanism degrades gracefully to boundary cases; so let us see how we can represent coroutines. The following two classes will 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} -- Implementation
    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 (created through a "once" function not shown here). 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.

Exercise E30.10, page 1036.

Although it looks like normal concurrent software, this solution ensures that (if all coroutines have different identifiers) at most one coroutine may proceed at any time, making it useless to allocate more than one physical CPU. (The controller could actually make use of its own CPU, but its actions are so simple as not to warrant it.)

The recourse to integer identifiers is necessary since giving *resume* an argument of type *COROUTINE*, a separate type, would cause deadlock. In practice, you should probably use **unique** declarations to avoid having to choose the 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 *resume* (*i*) will permit restarting 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 CPUs as a mechanism for controlling the maximum number of processes of a certain type which may be simultaneously active.

An elevator control system

The following example shows a case where object technology and the mechanism defined in this chapter can be used to achieve a pleasantly decentralized event-driven architecture for a real-time application.

The example describes software for an elevator control system, with several elevators serving many floors. The design below is somewhat fanatically object-oriented in that every significant type of component in the physical system — for example the

Milner quote: *page 956*.

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). This is getting close to Milner's wish, quoted at the beginning of this chapter, of making all objects parallel. The benefit is that the system is entirely event-driven; 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.

This implementation of the elevator example, adapted to control elevator displays on multiple screens and computers across the Internet (rather than actual elevators), has been used at several conferences to demonstrate concurrent and distributed O-O mechanisms.

Class *MOTOR* 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 elevator, it is free again, thanks to the wait by necessity 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*. Only for a very short time will an instance of *ELEVATOR* be reserved by a call from either a *MOTOR* or *BUTTON* object

```
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
```

Buttons are of two kinds: floor buttons, which passengers press to call the elevator to a certain floor, and cabin buttons, inside a cabin, which they press to make the cabin move to a certain floor. The two kinds send different requests: for a cabin button, the request is directed to a specific cabin; for a floor button, it can be handled by any elevator and so will be sent to a dispatcher object, which will poll the various elevators to select one that will handle the request. (The 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 associated 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 develop the algorithm that selects an elevator in procedure *accept*, you would need to let it 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 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 what elevator 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

Along with the previous one, the following example shows the mechanism's applicability to real-time problems. It also provides a good illustration of the concept of duel.

We want to enable an object 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 the behavior 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.

All routines with an argument t: REAL need the precondition t >= 0, omitted for brevity.

```
 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); unset_alarm (t); failed := False
        rescue
            if is_concurrency_interrupt then failed := True end
        end
        set_alarm (t: REAL) is
                    -- Set alarm to interrupt current object after t seconds.
                do
                            -- Create alarm if necessary:
                    if alarm = Void then !! alarm end
                    yield; actual_set (alarm, t); retain
                end
        unset_alarm (t: REAL) is
                    -- Remove the last alarm set.
                do
                    demand; actual_unset (alarm); wait_turn
                end
        action is
                    -- The action to be performed under watchdog control
                deferred
                end
feature {NONE} -- Actual access to watchdog
        actual_set (a: WATCHDOG; t: REAL) is
                    -- Start up a to interrupt current object after t seconds.
                do
                    a.set (t)
                end
        … Procedure actual_unset similar, left to the reader …
feature {WATCHDOG} -- The interrupting operation
        stop is
                    -- Empty action to let watchdog interrupt a call to timed_action
                do -- Nothing end
end -- class TIMED

separate class
        WATCHDOG
feature {TIMED}
        set (caller: separate TIMED; t: REAL) is
                    -- After t seconds, interrupt caller;
                    -- if interrupted before, terminate silently.
                require
                    caller_exists: caller /= Void
                local
                    interrupted: BOOLEAN
                do
                    if not interrupted then wait (t); demand; caller.stop; wait_turn end
                rescue
                    if is_concurrency_interrupt then interrupted := True; retry end
                end
```

See exercise E30.13, page 1036, about this procedure and the Business Card principle.

```
        unset is
                -- Remove alarm (empty action to let client interrupt set).
            do -- Nothing end
    feature {NONE}
        early_termination: BOOLEAN
    end -- class WATCHDOG
```

For clarity and to avoid mistakes every use of *retain* should, as here, include also the following *retain*, in the form *yield*; "Some call"; *retain*. Every use of *demand* (or *insist*) should similarly be of the form *demand*; "Some call"; *wait_turn*. You can use behavior classes to enforce this rule.

Accessing buffers

As a last example, let us wrap up the example of bounded buffers used several times in the presentation of the mechanism. We have seen that the class could be declared as just **separate class** *BOUNDED_BUFFER* [*G*] **inherit** *BOUNDED_QUEUE* [*G*] **end**, assuming the proper sequential *BOUNDED_QUEUE* class.

To use a call such as *q*.*remove* on an entity *q* of type *BOUNDED_BUFFER* [*T*], you must enclose it in a routine using *q* as formal argument. It may be useful for that purpose to provide a class *BUFFER_ACCESS* that fully encapsulates the notion of bounded buffer; application classes may inherit from *BUFFER_ACCESS*. There is nothing difficult about this behavior class, but it provides a good example of how we can encapsulate separate classes, directly derived from sequential ones such as *BOUNDED_QUEUE*, so as to facilitate their direct uses by concurrent applications.

```
    indexing
        description: "Encapsulation of access to bounded buffers"
    class BUFFER_ACCESS [G] is
        put (q: BOUNDED_BUFFER [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: BOUNDED_BUFFER [G]) is
                    -- Remove an element from q, waiting if necessary
                    -- until there is such an element.
            require
                    not q.empty
            do
                    q.remove
            ensure
                    not q.full
            end
        item (q: BOUNDED_BUFFER [G]): G is
                    -- Oldest element not yet consumed
            require
                    not q.empty
            do
                    Result := q.item
            ensure
                    not q.full
            end
    end
```

30.10 TOWARDS A PROOF RULE

On first reading you may move to "DISCUSSION", 30.12, page 1028.

(This section is for mathematically-inclined readers only. Although you may understand the basic ideas without having had a formal exposure to the theory of programming languages, full understanding requires that you be familiar with the basics of that theory, as given for example in [M 1990], whose notations will be used here.)

The basic mathematical property of sequential object-oriented computation was given semi-formally in the discussion of Design by Contract:

$\{INV$ **and** $pre\}$ $body$ $\{INV$ **and** $post\}$

where *pre*, *post* and *body* are the precondition, postcondition and body of a routine, and *INV* is the class invariant. With suitable axiomatization of the basic instructions this could serve as the basis of a fully formal axiomatic semantics for object-oriented software.

Without going that far, let us express the property more rigorously in the form of a proof rule for calls. Such a rule is fundamental for a mathematical study of O-O software since the heart of object-oriented computation — whether sequential as before, or concurrent as we are now able to achieve — is 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 proof rule for the sequential case may be informally stated as follows:

The basic sequential proof technique

> If we can prove that the body 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 non-qualified call in the assertions (of the form *some_boolean_property*) 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

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

More formally, we may express the basic proof rule as an adaptation to the object-oriented form of computation of Hoare's procedure proof rule:

$$\frac{\{INV \wedge \bigwedge_{p \in Pre\,(r)} p\}\ Body\,(r)\ \{INV \wedge \bigwedge_{q \in Post\,(r)} q\}}{\{\bigwedge_{p \in Pre\,(r)} p'\}\ Call\,(r)\ \{\bigwedge_{q \in Post\,(r)} q'\}}$$

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

$clause_1;\ \dots;\ clause_n$

The large "and" signs $\bigwedge$ indicate conjunction of all the 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 call's actual arguments 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 [M 1990].

The reason for considering the assertion clauses separately and then "anding" them is that this form prepares the rule's adaptation, described next, to separate calls in the concurrent case. Also of interest as preparation for the concurrent version is that you must take the invariant *INV* into account in the proof of the routine body (above the line), without any visible benefit for the proof of the call (below the line). More assertions with that property will appear in the concurrent rule.

What then changes with concurrency? 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. In a routine of the form

```
f (…, a: T, …) is
        require
                clause1; clause2; …
        do
                …
        end
```

any of the precondition clauses not involving 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 preceding sequential rule:

$$\frac{\{INV \wedge \bigwedge_{p \in Pre(r)} p\}\ Body(r)\ \{INV \wedge \bigwedge_{q \in Post(r)} q\}}{\{\bigwedge_{p \in Nonsep\_Pre(r)} p'\}\ Call(r)\ \{\bigwedge_{q \in Nonsep\_Post(r)} q'\}}$$

where *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 in detail earlier in this chapter, 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, following a discussion that highlighted the role of the preconditions.

30.11 A SUMMARY OF THE MECHANISM

On first reading you may move to "DISCUSSION", 30.12, page 1028.

Here now is the precise description of the concurrency facilities presented in earlier sections. There is no new material in this section, which serves only as reference and may be skipped on first reading. The description consists of four parts: syntax; validity constraints; semantics; library mechanisms. It extends the sequential O-O mechanisms developed in the preceding chapters.

Syntax

The syntactic extension involves just one new keyword, **separate**.

A declaration of an entity or function, which normally appears as

x: *TYPE*

may now also be of the form

x: **separate** *TYPE*

In addition, a class declaration, which normally begins with one of **class** *C*, **deferred class** *C* and **expanded class** *C*, may now also be of a new form: **separate class** *C*. In this case *C* will be called a separate class. It follows from the syntax convention that a class may be at most one of: separated, expanded, deferred. As with expanded and deferred, the property of being separate is not inherited: a class is separate or not according to its own declaration, regardless of its parents' separateness status.

A type is said to be separate if it is either based on a separate class or of the form **separate** *T* for some *T* (in which case it is not an error, although redundant, for *T* to be separate — again the same convention as for expanded). An entity or function is separate if its type is separate. An expression is separate if it is either a separate entity or a call to a separate function. A call or creation instruction is separate if its target (an expression) is separate. A precondition clause is separate if it involves a separate call (whose target, because of rules that follow, can only be a formal argument).

Constraints

A Separateness Consistency rule in four parts governs the validity of separate calls:

- (1) If the source of an attachment (assignment instruction or assignment passing) is separate, its target entity must be separate too.
- (2) If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.
- (3) If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.
- (4) If an actual argument of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

There is also a simple consistency rule on types (not given earlier): in a type of the form **separate** *TYPE*, the base class of *TYPE* must be neither deferred nor expanded.

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

If an assertion contains a function call, any actual argument of that call must, if separate, be a formal argument of the enclosing routine, if any (separate argument rule).

Semantics

Each object is handled by a processor, its handler. If the target *t* of a creation instruction is non-separate, the newly created object will be handled by the same processor as the creating object. If *t* is separate, the new object will be allocated to a new processor.

Once it has been created, an object will at any time be in either of two states: free and reserved. It is free if no feature is being executed on it, and no separate client is currently executing a routine that uses as actual argument a separate reference attached to it.

A processor will be in one of three states: idle, busy and suspended. It is busy if it is executing a routine whose target is an object that it handles. It becomes suspended if it attempts an unsuccessful call (defined below) whose target is an object that it handles.

The semantics of calls is affected only if one of more of the elements involved — target and actual arguments — are separate. The discussion assumes a call of the general form *t*.*f* (…, *s*, …) where *f* is a routine. (If *f* is an attribute, we will assume for simplicity that it is called through an implicit function returning its value.)

The call is executed as part of the execution of a routine on a certain object C_OBJ, which may only be in a busy state at that stage. The basic notion is the following:

Definition: satisfiable call

In the absence of *CONCURRENCY* features (described next), a call to a routine *f*, executed on behalf of an object C_OBJ, is satisfiable if and only if every separate actual argument having a non-void value, and hence attached to a separate object A_OBJ, satisfies the following two conditions if the routine uses the corresponding formal as target of at least one call:

S1 • A_OBJ is free or reserved by C_OBJ.

S2 • Every separate clause of the precondition of *f* has value true when evaluated for A_OBJ and the actual arguments given.

If a processor executes a satisfiable call, the call is said to be successful and proceeds immediately; C_OBJ remains reserved, its processor remains in the busy state, every A_OBJ becomes reserved, the target remains reserved, the target's handler becomes busy, and it starts executing the routine of the call. When the call terminates, the target's handler returns to its previous state (idle or suspended) and each A_OBJ object returns to its previous state (free or reserved by C_OBJ).

If the call is not satisfiable, it is said to be unsuccessful; C_OBJ enters the suspended state. The call attempt has no immediate effect on its target and actual arguments. If one or more earlier unsuccessful calls are now satisfiable, the processor selects one of them to become successful as just described. The default policy if more than one is satisfiable is to select the one that has been waiting longest.

The final semantic change is the definition of wait by necessity: if a client has started one of more calls on a certain separate object, and it executes on that object a call to a query, that call will only proceed after all the earlier ones have been completed, and any further client operations will wait for the query call to terminate. (We have seen that an optimizing implementation might apply this rule only to queries returning an *expanded result*.) When waiting for these calls to terminate, the client remains in the "reserved" state.

Library mechanisms

Features of class *CONCURRENCY* enable us in some cases to consider that condition S1 of the satisfiable call definition holds even if A_OBJ has been reserved by another object (the "holder"), assuming C_OBJ (the "challenger') has called *demand* or *insist*; if as a result the call is considered satisfiable, the holder will get an exception. This will only occur if the holder is in a "yielding" state, which it can achieve by calling *yield*.

To go back to the default non-yielding state, the holder can execute *retain*; the boolean query *yielding* indicates the current state. The challenger's state is given by the integer query *Challenging* which may have the value *Normal*, *Demanding* or *Insisting*.

To return to the default *Normal* state the challenger can execute *wait_turn*. The difference between *demand* and *insist* affects what happens if the holder is not *yielding*: with *demand* the challenger will get an exception; with *insist* it simply waits as with *wait_turn*.

When these mechanisms cause an exception in the holder or challenger, the boolean query *is_concurrency_exception* from class *EXCEPTIONS* has value true.

30.12 DISCUSSION

As a conclusion to this presentation, let us review the essential criteria that should guide the development of a concurrent O-O mechanism. These criteria served as a basis for the approach presented here; in a few cases, as will be seen, some more work remains to be done to achieve their full satisfaction. The goals include:

- Minimality of mechanism.
- Full use of inheritance and other object-oriented techniques.
- Compatibility with Design by Contract.
- Provability.
- Support for command-query distinction.
- Applicability to many forms of concurrency.
- Support for coroutine programming.
- Adaptability through libraries.
- Support for reuse of non-concurrent software.
- Support for deadlock avoidance.

We will also take a final look at the question of interleaving accesses to an object.

Minimality of mechanism

Object-oriented software construction is a rich and powerful paradigm, which, as noted at the beginning of this chapter, intuitively seems ready to support 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 will conflict with them, making the developer'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.

The extension presented in the preceding sections is indeed minimal syntactically, since it is not possible to add less than one new keyword.

Full use of inheritance and other object-oriented techniques

It would be unacceptable to have a concurrent object-oriented mechanism that does not take advantage of all O-O techniques, in particular inheritance. We have noted that the "inheritance anomaly" and other potential conflicts are not inherent to concurrent O-O development but follow from specific choices of concurrency mechanisms, in particular active objects, state-based models and path-expression-like synchronization; the appropriate conclusion is to reject these choices and retain inheritance.

We have repeatedly seen how inheritance can be used to produce high-level behavior class (such as *PROCESS*) describing general patterns to be inherited by descendants. Most of the examples would be impossible without multiple inheritance.

Among other O-O techniques, information hiding also plays a central role.

Compatibility with Design by Contract

It is essential to retain the systematic, logic-based approach to software construction and documentation expressed by the principles of Design by Contract. The results of this chapter were indeed based on the study of assertions and how they fare in a concurrent context.

In that study we encountered a striking property, the concurrent precondition paradox, which forced us to provide a different semantics for assertions in the concurrent case. This gives an even more fundamental place to assertions in the resulting mechanism.

Support for command-query separation

"Objects as machines", page 751.

A principle of object-oriented software construction was developed in preceding chapters: Command-Query Separation. The principle enjoins us not to mix commands (procedures), which change objects, and queries (functions and attributes), which return information about objects but do not change them. This precludes side-effect-producing functions.

It is commonly believed that the principle cannot hold in a concurrent context, as for example you cannot write

```
next_element := buffer.item
buffer.remove
```

and have the guarantee that the element removed by the second call is the same that the first instruction assigned to *next_item*. Between the two instructions, another client can mess up with the shared buffer. Such examples are often used to claim that one must have a side-effect-producing function *get*, which will both return an element and remove it.

This argument is plainly wrong. It is confusing two notions: exclusive access and routine specification. With the notation of this chapter, it is easy to obtain exclusive access without sacrificing the Command-Query Separation principle: simply enclose the two instructions above, with *buffer* replaced by *b*, in a procedure of formal argument *b*, and call that procedure with the attribute *buffer* as argument. Or, if you do *not* require the two operations to apply to the same element, and want to minimize the amount of time a shared resource is held, write *two* separate routines. This kind of flexibility is important for the developer. It can be provided, thanks to a simple exclusive access mechanism, whether or not functions may have side effects.

Applicability to many forms of concurrency

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

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. This is achieved by using the abstract notion of processor, and relying on a distinct facility (Concurrency Control File, libraries…) to adapt the solution to any particular hardware architecture that you may have available.

Adaptability through libraries

Many concurrency mechanisms have been proposed over the years; some of the best known were reviewed at the beginning of this chapter. 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 that mechanism.

Here the facilities of the object-oriented method should again be put to good use. One of the most important aspects 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. Examples of such encapsulating mechanisms (such as the *PROCESS* class and the behavior class for locks) have been presented in this chapter, and the exercises suggest a few more.

One may expect that a number of libraries will be produced, relying on the basic tools and complementing them, to support concurrency models catering to specific needs and tastes.

We have also seen the use of library classes such as *CONCURRENCY* to provide various refinements to the basic scheme defined by the language mechanism.

Support for coroutine programming

As a special case, coroutines provide a form of quasi-concurrency, interesting both in itself (in particular for simulation activities), and as a smoke test of the applicability of the mechanisms, since a general solution should adapt itself gracefully to boundary cases. We have seen how it is possible, once again using the library construction mechanisms of object technology, to express coroutines based on the general concurrent mechanism.

Support for reuse of non-concurrent software

It is necessary to support the reuse of existing, non-concurrent software, especially libraries of reusable software components.

Class BUFFER_ ACCESS was on page 1021.

We have seen how smooth the transition is between sequential classes such as *BOUNDED_QUEUE* and their concurrent counterparts such as *BOUNDED_BUFFER* (just write **separate class** *BOUNDED_BUFFER* [*G*] **inherit** *BOUNDED_QUEUE* [*G*] **end**). This result is somewhat tempered by the frequent desirability of encapsulation classes such as our *BUFFER_ACCESS*. Such encapsulation seems useful, however, and may be an inescapable consequence of the semantic differences between sequential and concurrent computation. Also note that such wrapper classes are easy to write.

Support for deadlock avoidance

One area in which more work remains necessary is how to guarantee deadlock avoidance.

Deadlock potential is a fact of concurrent life. For example any mechanism that can be used to program semaphores (and a mechanism that is *not* powerful enough to emulate semaphores would be viewed with suspicion) can cause deadlock, since semaphores are trivially open to that possibility.

"Locks", page 1009, and exercise E30.8, page 1036.

The solution lies partly in the use of high-level encapsulation mechanisms. For example a set of classes encapsulating semaphores, as was presented for locks, should come with behavior classes that automatically provide a *free* for every *reserve*, thereby guaranteeing deadlock avoidance for applications that follow the recommended practice by inheriting from the behavior class. This is, in my experience, the best recipe for deadlock avoidance.

This approach may not be sufficient, however, and it may be possible to devise simple non-deadlock rules, automatically checkable by a static tool. Such a rule (expressed as a methodological principle rather than a language validity rule, for fear it may be too restrictive) was given earlier: the *Business Card* principle. But more is needed.

Permitting concurrent access?

A final note on one of the principal properties of the approach: the requirement that at most one client may access any supplier object at any given time, preventing interleaving of routines and requiring any VIP treatment to use the duel mechanism.

"Concurrent accesses to an object", page 982.

The rationale was clear: if any challenger client can interrupt the execution of a routine at any time, we lose the ability to reason on our classes (through properties of the form {*INV* **and** *pre*} *body* {*INV* **and** *post*}) since the challenger can leave the object in an arbitrary state.

This objection would disappear if we only permitted challengers to execute a routine of a very special kind: an *applicative* routine (in the sense defined for functions in earlier chapters) which does not modify the object or, if it modifies it, cancels all its modifications before it leaves. This would assume a language mechanism to state that a routine is applicative, and compilers enforcing that property.

30.13 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Concurrency and distribution are playing an increasing role in most application areas of computers.
- Concurrency has many variants, including multiprocessing and multiprogramming. The Internet, the Web and object request brokers bring even more possibilities.
- It is possible to use the fundamental schemes of object technology — classes, encapsulation, multiple inheritance, deferred classes, assertions and so on — for the greatest benefit of developers of concurrent and distributed applications.
- No active-passive object distinction is necessary or desirable. Objects are by nature able to perform many operations; making them active would restrict them to just one.
- A simple extension of the sequential object-oriented notation, using a single keywords (**separate**), covers all the major application areas of concurrency.
- Each object is handled by a processor. Processors are an abstract notion describing threads of control; a system can use as many processors as it wants regardless of the number of available computing devices (CPUs). It must be possible to define the mapping from processors to CPUs outside of the software proper.
- An object handled by a different processor is said to be separate.
- Calls on separate targets have a different semantics, asynchronous rather than synchronous. For that reason, any entity representing separate objects must be declared as such, using the keyword **separate**.
- Consistency rules, implying in particular that a separate entity may not be assigned to a non-separate one, ensure that there are no “traitors” — that no non-separate entity becomes attached to a separate object.
- To achieve exclusive access to a separate object, it suffices to use the corresponding reference as an argument to a separate call (a call with a separate target).
- The target of a separate call must itself be a separate formal argument of the enclosing routine.
- Preconditions on separate targets cannot keep their usual semantics as correctness conditions (this is the “concurrent precondition paradox”). They serve as wait conditions.
- The mechanism developed in this chapter covers multitasking, time-sharing, multi-threading, client-server computing, distributed processing on networks such as the Internet, coroutines and real-time applications.

30.14 BIBLIOGRAPHICAL NOTES

The approach to concurrency described in this chapter evolved from a presentation at TOOLS EUROPE [M 1990a] and was revised in [M 1993b], from which some of the material in this chapter (examples in particular) was derived. It is now known as SCOOP for "Simple Concurrent Object-Oriented Programming". John Potter and Ghinwa Jalloul have developed a variant that includes an explicit **hold** instruction [Jalloul 1991] [Jalloul 1994]. Wait by necessity was introduced by Denis Caromel [Caromel 1989] [Caromel 1993].

The first implementation of the model described here was developed by Terry Tang and Xavier Le Vourch. Both contributed new insights.

A good textbook on the traditional approaches to concurrency is [Ben Ari 1990]. Original references include: on semaphores, [Dijkstra 1968a], which also introduced the "dining philosophers" problem; on monitors, [Hoare 1974]; on path expressions, [Campbell 1974]. The original CSP model was described in [Hoare 1978]; the book [Hoare 1985] presents a revised model with special emphasis on its mathematical properties. Occam2 is described in [Inmos 1988]. A CSP and Occam archive is available at Oxford University: *http://www.comlab.ox.ac.uk/archive/csp.html* (I am grateful to Bill Roscoe from Oxford for help with details of CSP). CCS (Communicating Concurrent Systems) [Milner 1989] is another influential mathematically-based model. Although cited only in passing in this chapter, Carriero's and Gelernter's Linda method and tool [Carriero 1990] is a must know for anyone interested in concurrency.

A special issue of the *Communications of the ACM* [M 1993a] presents a number of important approaches to concurrent object-oriented programming, originally drawn from concurrency papers at various TOOLS conferences.

Another collection of papers that appeared at about the same time is [Agha 1993]. An earlier collective book edited by Yonezawa and Tokoro [Yonezawa 1987] served as catalyst for much of the work in the field and is still good reading. Other surveys include a thesis [[Papathomas 1992] and an article [Wyatt 1992]. Yet another compilation of contributions by many authors [Wilson 1996] covers C++ concurrency extensions.

Hewitt's and Agha's *actors* model, which predates the object-oriented renaissance and comes from a somewhat different background, has influenced many concurrent O-O approaches; it is described in an article [Agha 1990] and a book [Agha 1986]. Actors are computational agents similar to active objects, each with a mail address and a behavior. An actor communicates with others through messages sent to their mail addresses; to achieve asynchronous communication, the messages are buffered. An actor processes messages through functions and by providing "replacement behaviors" to be used in lieu of the actor's earlier behavior after a certain message has been processed.

One of the earliest and most thoroughly explored parallel object-oriented languages is POOL [America 1989]; POOL uses a notion of active object, which was found to raise problems when combined with inheritance. For that reason inheritance was introduced into the language only after a detailed study which led to the separation of inheritance and subtyping mechanisms. The design of POOL is also notable for having shown, from the start, a strong concern for formal language specification.

Much of the important work in concurrent O-O languages has come from Japan. [Yonezawa 1987], already cited, contains the description of several influential Japanese developments, such as ABCL/1 [Yonezawa 1987a]. MUSE, an object-oriented operating system developed at the Sony Computer Science Laboratory, was presented by Tokoro and his colleagues at TOOLS EUROPE 1989 [Yokote 1989]. The term "inheritance anomaly" was introduced by Matsuoka and Yonezawa [Matsuoka 1993], and further papers by Matsuoka and collaborators which propose various remedies.

Work on distributed systems has been particularly active in France, with the CHORUS operating system, of which [Lea 1993] describes an object-oriented extension; the GUIDE language and system of Krakowiak *et al*. [Balter 1991]; and the SOS system of Shapiro *et al*. [Shapiro 1989]. In the area of programming massively parallel architectures, primarily for scientific applications, Jean-Marc Jézéquel has developed the ÉPÉE system [Jézéquel 1992], [Jézéquel 1996] (chapter 9)], [Guidec 1996].

Also influential has been the work done by Nierstrasz and his colleagues at the University of Genève around the Hybrid language [Nierstrasz 1992] [Papathomas 1992], which does not have two categories of objects (active and passive) but relies instead on the notion of thread of control, called *activity*. The basic communication mechanism is remote procedure call, either synchronous or asynchronous.

Other important projects include DRAGOON [Atkinson 1991], which, like the mechanism of this chapter, uses preconditions and postconditions to express synchronization, and pSather [Feldman 1993], based on the notion of thread and a predefined *MONITOR* class.

Many other developments would need to be added to this list. For more complete surveys, see the references cited at the beginning of this section. The proceedings of workshops regularly held at the ECOOP and OOPSLA conferences, such as [Agha 1988], [Agha 1991], [Tokoro 1992], describe a variety of ongoing research projects and are precious to anyone who wants to find out what problems researchers consider most pressing.

The work reported in this chapter has benefited at various stages from the comments and criticism of many people. In addition to colleagues cited in the first two paragraphs of this section they include Mordechai Ben-Ari, Richard Bielak, John Bruno, Paul Dubois, Carlo Ghezzi, Peter Löhr, Dino Mandrioli, Jean-Marc Nerson, Robert Switzer and Kim Waldén.

EXERCISES

E30.1 Printers

Page 962.

Complete the *PRINTER* class, implementing the job queue as a bounded buffer and making sure queue manipulation routines as well as *print* do not need to process the special “stop request” print job (*print* may have **not** *j*.*is_stop_request* as a precondition).

E30.2 Why import must be deep

Assume that a shallow import mechanism (rather than *deep_import*) were available. Construct an example that will produce an inconsistent structure — one in which a separate object is attached to a non-separate entity.

E30.3 The “inheritance anomaly”

“Synchronization for concurrent O-O computation”, page 980.

In the *BUFFER* example used to illustrate the “inheritance anomaly”, assume that each routine specifies the exit state in each case using a **yield** instruction, as in

```
put (x: G) is
        do
                "Add x to the data structure representing the buffer"
                if "All positions now occupied" then
                        yield full
                else
                        yield partial
                end
        end
```

Write the corresponding scheme for *remove*. Then write the class *NEW_BUFFER* with the added procedure *remove_two* and show that the class must redefine both of the inherited features (along with the specification of which features are applicable in which states).

E30.4 Deadlock avoidance (research problem)

Page 990.

Starting from the Business Card principle, investigate whether it is feasible to eliminate some of the possible deadlocks by introducing a validity rule on the use of non-separate actual arguments to separate calls. The rule should be reasonable (that is to say, it should not preclude commonly useful schemes), enforceable by a compiler (in particular an incremental compiler), and easily explainable to developers.

E30.5 Priorities

“Duels and their semantics”, page 999.

Examine how to add a priority scheme to the duel mechanism of class *CONCURRENCY*, retaining upward compatibility with the semantics defined in the presentation of procedures *yield*, *insist* and related ones.

E30.6 Files and the precondition paradox

Consider the following simple extract from a routine manipulating a file:

```
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
```

Discuss how, in spite of the absence of obvious concurrency in this example, the precondition paradox can apply to it. (**Hint**: a file is a separate persistent structure, so an interactive user or some other software system can access the file in between the various operations performed by the extract.) Discuss what can happen as a consequence of this problem, and possible solutions.

E30.7 Locking

Rewrite the class *LOCKING_PROCESS* as an heir of class *PROCESS*.

Page 1010. Class PROCESS was on page 961.

E30.8 Binary semaphores

Write one or more classes implementing the notion of binary semaphore. (**Hint**: start from the classes implementing locks.) As suggested at the end of the discussion of locks, be sure to include high-level behavior classes, meant to be used through inheritance, which guarantee a correct pattern of *reserve* and *free* operations.

"Locks", page 1009.

E30.9 Integer semaphores

Write one or more classes implementing the notion of integer semaphore.

E30.10 Coroutine controller

Complete the implementation of coroutines by spelling out how the controller is created.

"Coroutines", page 1012.

E30.11 Coroutine examples

The discussion of Simula presents several examples of coroutines. Use the coroutine classes of the present chapter to implement these examples.

"Coroutine concepts", page 1118 (in the Simula chapter).

E30.12 Elevators

Complete the elevator example by adding all the creation procedures as well as the missing algorithms, in particular for selecting floor requests.

"An elevator control system", page 1014

E30.13 Watchods and the Business Card principle

Show that the procedure *set* of class *WATCHDOG* violates the Business Card principle. Explain why this is all right.

Page 1020.

E30.14 Once routines and concurrency

What is the appropriate semantics for once routines in a concurrent context: executed once per system execution, or once per processor?