The C++// language - DRAFT

Denis Caromel, Fabrice Belloncle, and Yves Roudier
I3S - CNRS - University of Nice - INRIA Sophia Antipolis
650 Rte des Colles, B.P. 145 - 06903 Sophia Antipolis - France
{caromel,belloncl,roudier}@unice.fr
`http://www.inria.fr/sloop/c++ll` and
`http://wwwi3s.unice.fr/c++ll`

SLOOP

Introduction

The C++// language (pronounced "C++ parallel") was designed and implemented with the aim of importing reusability into parallel and concurrent programming, in the framework of a MIMD model. C++// defines a comprehensive and versatile set of libraries, using a small set of simple primitives, without extending the syntax of C++. The libraries are themselves extensible by end users, making C++// an open system.

Reusability has been one of the major contributions of object-oriented programming; bringing it to parallel programming is one of our main goals, and it would be a major step forward for software engineering of parallel systems. Part of the challenge is to combine the potential for extensive reuse with the high performance which is usually required of parallel and real-time systems.

Working mainly within the framework of physically-distributed architectures, we are concerned with both explicit and implicit parallelism in both the problem and solution domains. Our applications include parallel data structures, computer-supported cooperative work (CSCW), and fault-tolerance and reliability in safety-critical and real-time systems.

We are part of the *SLOOP* (Simulation and Parallel Object-Oriented Languages) project, a recently-formed research team with approximately 15 members at I3S-CNRS, the University of Nice, and INRIA Sophia Antipolis. This team is investigating three research areas: parallel and distributed discrete event simulation, parallel object-oriented languages, and communication and interconnection networks. These three domains are not independent: each level needs and uses the primitives and capabilities of the layer beneath it. As a member of this group, supporting distributed simulations [Mallet & Mussi 1993] is of first importance for our system.

To achieve this, we began the design and implementation of C++// in early 1994. While a recent development, C++// owes a significant part of its design and implementation techniques to previous research, which led to the definition of Eiffel// [Caromel 1989 , Caromel 1993a , Caromel 1993b], a parallel extension of Eiffel [Meyer 1988 , Meyer 1992]. In particular, C++// defines a reduced set of simple primitives: these can then be composed to create comprehensive and versatile libraries, which—most importantly—can then be extended by end users.

Another important characteristic of our system is the complete absence of any syntactical extension to C++. C++// users write standard C++ code, relying on specific classes to give programs a parallel semantics. These programs are then passed through a pre-processor, which generates new files. The original and new code are finally compiled and linked with a standard C++ compiler. When appropriate, all names related to the C++// system include the `ll` root in their name (for "para`ll`el").

This chapter begins by describing the basic features of our programming model, which is a distributed-memory MIMD model. Section 2 deals with the control programming of processes (i.e., the definition of concurrent process activity). A recommended method for parallel programming in C++// is outlined in Section 3. Those parts of the programming environment which handle compilation and mapping are described in Section 4. We then present our implementation of polygon overlay in Section 5. Finally, an overview of the implementation techniques which make the system open and user-extensible is given in Section 6, and we present some concluding remarks.

Information on our system is available from `http://www.inria.fr/sloop/c++ll` and `http://wwwi3s.unice.fr/c++ll`. Queries and comments can be sent to `c++ll@unice.fr` and `c++ll@sophia.inria.fr`.

1. Basic Model of Concurrency

This section describes four important characteristics of our parallel programming model: parallel processes, communication between them, synchronization, and data sharing. As mentioned above, we adopt a distributed-memory MIMD model, which means that there are no directly-shared objects in our system.

Along with simplicity and expressiveness, reusability is one of our major concerns. More specifically, we want to allow users to take an existing C++ system and transform it into a distributed one, so that they may derive parallel systems from sequential ones [Caromel 1990b].

1.1. Processes

One of the key features of the object-oriented paradigm is the unification of the notions of module and type to create the notion of class. When adding parallelism, another unification is to bring together the concepts of class and process, so that every process is an instance of a class, and the process's possible behavior is completely described by its class. However, not all objects are processes. At run-time, we distinguish two kinds of objects: *process objects* (or active objects), which are active by themselves, with their own thread of control, and *passive objects*, which are normal objects. This second category includes all non-active objects. An example of the arrangement of processes and objects at run-time is given in Figure 1.1.

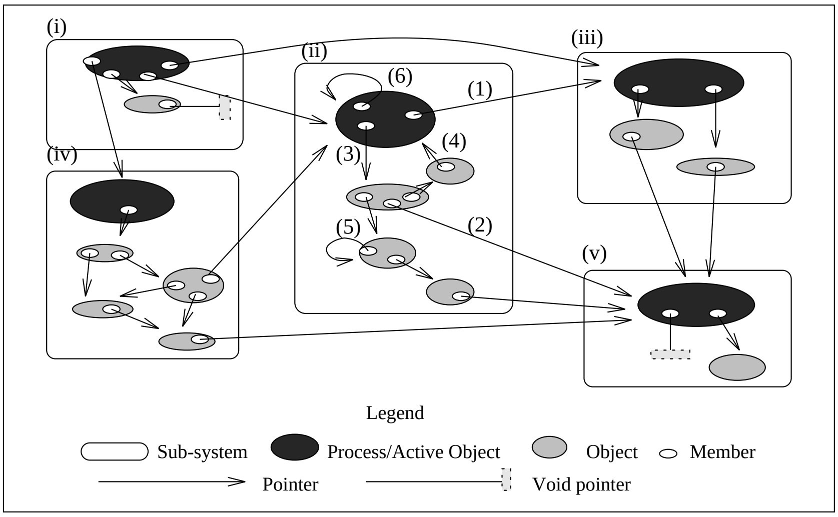

Figure 1.1
Processes and objects at run-time

At the language level, there are two ways to generate active objects. In the first, an active object is obtained by instantiating a standard sequential C++ class using `Process_alloc`:

```
A* p;           // A is a normal sequential class
p = (A *) new (typeid(A)) Process_alloc(···);
```

In this case, a standard sequential class `A` is instantiated to create an active object, whose method invocations are then serviced in the order in which they are received. The `Process_alloc` class is part of the C++// library, while `typeid` is the standard C++ run-time type identification (RTTI) operator. We will refer to this technique as the *allocation-based* process creation, and say that it produces an *allocation-based process*, or *allocation-based active object*. The allocation style is convenient, but limited because it only allows us to create processes which handle method invocations in a first-in, first-out (FIFO) manner.

The second technique, which we call *class-based*, is more general. All objects which are an instance of a class that publicly inherits from the class `Process` are processes. This `Process` class is part of the C++// library. To use class-based process creation, the programmer must derive a specific class, called a *process class*, from `Process`, as in:

```
class Parallel_A : public A, public Process {
  ⋮
};
⋮
Parallel_A* p;
p = new Parallel_A(···);
```

As with the allocation-based technique, instances of sub-classes of `Process` have a default FIFO behavior. However, as we will see in the following sections, it is possible to change this to create other behaviors. We say that the class-based technique generates *class-based processes*, or *class-based active objects*.

As shown on Figure 1.1, passive objects (i.e., objects which are not active) belong at run-time to a single process object. This organizes a parallel program into disjoint sub-systems, each of which consists of one active object encapsulating zero or more passive objects. Figure 1.2 presents the two styles of active object definition.

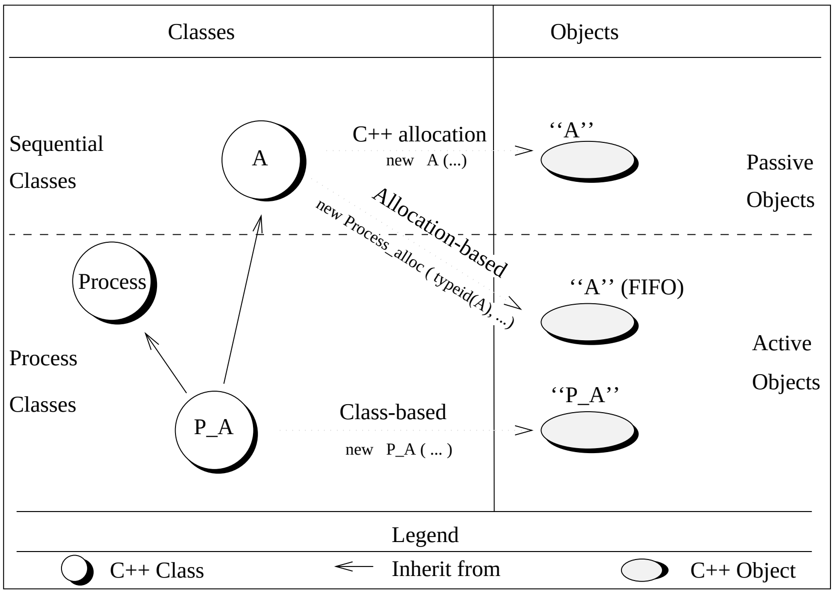

Figure 1.2
Allocation-Based and Class-Based Active Objects

1.2. Sequential or Parallel Processes

A major design decision for any concurrent programming system is whether processes are sequential (i.e., single-threaded) or able to support internal concurrency (i.e., multi-threaded). Because our system is oriented towards reuse and software engineering of parallel systems, rather than operating systems programming, we chose to make processes sequential. We believe that single-threaded processes are easier to reuse, and easier to write correctly.

The model does not allow the user to program multi-threaded processes, but this does not prevent multi-threading at the operating system level. As we will see in Section 4.2, several sequential processes can be implemented with one multi-threaded operating system process for the sake of light-weightness.

1.3. Communication

Since a process is an object, it has member functions. When an object owns a reference to a process, it is able to communicate with it by calling one of its public members. This is C++//'s interprocess communication (IPC) mechanism: all communication towards active objects appear syntactically as member function calls:

```
p->f(parameters);
```

This idea, introduced by the Actors model [Hewitt 1977 , Agha 1986], means that what is sometimes called a process entry point is identical to a normal routine or member function.

While this idea is widely used in parallel object-oriented systems, there are many differences in the definition of the semantics of method-based IPC. In C++//, IPC calls are asynchronous by default, while function calls to passive objects retain the synchronous semantics of standard C++. This choice encourages the parallel execution of objects, and makes each process more independent from other parallel activities and more self-contained. As we will see, it is also very important for supporting reusability. Synchronous function calls are possible in C++//, but must be specified explicitly in either the function call or the process definition.

The choice of asynchronous IPC for the default structures a C++// system into independent asynchronous *sub-systems*: all of the communication between sub-systems is asynchronous. Figure 1.1 illustrates five such sub-systems.

1.4. Synchronization

Asynchronous communication can be difficult for programmers to manage. For example, since function calls to processes are asynchronous, one needs to explicitly synchronize before using result values, to make sure they have been returned by the processes. Commonly, such models lack the implicit synchronization usually provided by a synchronous communication semantics.

We use a simple rule, called *wait-by-necessity*, to address this problem. In C++//, a process is automatically blocked when it attempts to use the result of a parallel member function call that has not yet been returned. Thus, a caller does not wait for the result of an asynchronous function call until that value is explicitly used in some computation. Should a value not have been returned at that point, the caller is automatically blocked until the value becomes available. This mech-

```
v = p->f(parameters);      // asynchronous call
:
v->foo();                  // implicitly triggers a wait
                           // if v is awaited
:
if (Awaited(v)) {          // test the status of v
  :
}

Wait(v);                   // explicitly triggers a wait
                           // if v is awaited
:
obj->g(v);                 // no wait if pointer access
v2 = v;
```

Program 1.1
Wait-by-Necessity

anism implicitly synchronizes processes; the two primitives `Wait()` and `Awaited()` are provided for explicit synchronization.

Program 1.1 summarizes the semantics of wait-by-necessity. The result of a function call not yet returned is called an *awaited object*. Our semantics define that no wait is triggered by assigning a pointer to such an object to another variable, or by passing such a pointer as a parameter. A wait occurs only when the program accesses the awaited object itself (which is syntactically a pointer access to the object) or transmits (copies) the object to another process.

Wait-by-necessity is a form of future [Halstead 1985], and is related to concepts found in several other languages: the `Hurry` primitive of Act1 [Lieberman 1987], the `CBox` objects of ConcurrentSmalltalk [Yokote & Tokoro 1987], and the future type message passing of ABCL/1 [Yonezawa *et al.* 1987]. However, an important difference is that the mechanism presented here is systematic for all asynchronous function calls and automatic, which is reflected in the absence of any special syntactic construction. This has a strong impact on reusability.

In order to avoid the run-time overhead involved in the implementation of wait-by-necessity, programmers can use explicit synchronization primitives instead of implicit synchronization. This is a tradeoff between programming ease and reusability on the one hand, and efficiency and speedup on the other.

1.5. Sharing

If two processes refer to the same passive object, method calls to that object may overlap, which raises all of the problems usually associated with shared data. To address this issue, each non-process object in C++// is a private object, and is accessible to only one process. We say that a private object belongs to its process's sub-system.

The programming model ensures that sharing does not occur: the communication of passive objects between processes uses a copying semantics. Passive object parameters of a call are automatically transmitted by value from one process to another. A deep copy of these objects is achieved: when an object `X` is copied, all the (also passive) objects referred to by pointers in `X` are deep-copied as well. The implementation automatically and transparently handles the required marshalling of data and pointers, as well as circular object structures. Of course, process parameters are always passed by reference.

Figure 1.1 illustrates the absence of passive objects shared by processes in C++// programs. Each passive object is accessible to exactly one active object; each of the five sub-systems in this program consists of one active object and all the passive objects it can reach. The arcs labelled (1) and (2) are always activated as asynchronous communication (IPC), while the arcs labelled (3) to (6) are activated as normal function calls. As a consequence of the absence of shared objects, synchronization between sub-systems only occurs when one sub-system waits for a result value from another process.

Prohibiting shared data also has important methodological consequences. As we shall see in Section 3.3, the features of C++//, together with object-oriented techniques such as polymorphism and dynamic binding, permits the derivation of parallel systems from sequential ones. The absence of shared objects allows either an immediate reuse (the default automatic copy of parameters is the correct semantics), or the identification of new processes to program in order to implement semantically shared values. Finally, due to the absence of interleaving of operations inside sub-systems, it helps to ensure the correctness of the parallel applications derived from sequential ones.

The basic characteristics of our programming model are summarized in Figure 1.3.

- A process is an active object, sequential and single-threaded.
- Communication to active objects are syntactically programmed as member function calls, and are asynchronous.
- An object is automatically blocked when it attempts to use the result of a member function call that has not been returned yet (wait-by-necessity).
- There are no shared passive objects.
- Passive objects parameters are passed by value (copy).

Figure 1.3
Basic Features of the C++// Model

2. Control Programming

So far, we have only examined and defined the features of C++// which deal with the global aspects of the programming model, such as the nature of processes and their interactions. This section describes how the control flow of processes is specified, i.e., how behavior, communication, and synchronization of active objects is programmed. Here and later, we use *request* to mean a call issued to a process member function. Since communication is asynchronous, the process to which the request was issued will later *serve* the request: that is to say, it will execute the member function being invoked.

2.1. Centralized versus Decentralized Control

Decentralized control is distributed throughout a program. Each routine, public or otherwise, may contain a part of the control. An example of this is the `uses` clause of the Hybrid langauge [Nierstrasz 1987]:

```
type buffer of (···);
⋮
private {
  put: (item: item_type) ->;
    uses not full;
    { ··· }
```

```
  get:->item_type;
    uses not empty;
    { ··· }
```

Alternatively, control can be centralized (i.e., gathered into one place in the definition of a process), independent of the function code. An example of this is the `CONTROL` construct of Guide [Decouchant *et al.* 1989]:

```
CLASS buffer IS

METHOD put(···); BEGIN
  ⋮
END put;

METHOD get(···); BEGIN
  ⋮
END get;

CONTROL
  put: ((completed(put) - completed(get)) < size )
       and (current(put) = 0);
  get: (completed(put) < completed(get))
       and (current(get) = 0);
END buffer
```

Decentralized control makes the reuse of member functions difficult for two reasons. First, functions designed in a sequential framework cannot be reused in a parallel one just as they are, as elements of control must be added to them. Second, when a new process class is obtained through inheritance of another process class, the new class often needs to change the synchronization scheme used in the original class. If control is embedded in function bodies, this may not be feasible. This led us to decide to use centralized control in C++//, as it allows function reuse for both sequential and process classes.

Program 1.2 presents partial code for the library class `Process`. After creation and initialization, a process object executes its `Live()` routine. This routine describes the sequence of actions which that process executes during its lifetime. The process terminates when the `Live()` routine completes.

```
class Process {
 public:
  Process (···) {      // process creation
    ⋮
  }

  virtual void Live() {      // process body
    ⋮  // default FIFO behavior
  }
  ⋮
};
```

Program 1.2
The `Process` Class

2.2. Explicit vs. Implicit Control

Another design decision that must be made in concurrent object-oriented systems is whether process control is implicit or explicit. Control is explicit if its definition consists of an explicitly programmed thread of control (e.g., as in Hybrid). Otherwise, control is implicit, which in practice usually means that it is declarative (e.g., as in Guide).

Implicit control is a high-level concept. Thanks to its declarative nature, it is an effective way to express synchronization. It usually provides programmers with a consistent framework, in which they can forget implementation details and describe the synchronization constraints in a very synthetic manner. Implicit control has also the advantage of promoting synchronization reuse [Matsuoka *et al.* 1990 , Neusius 1991 , Decouchant *et al.* 1991 , Shibayama 1991 , Frolund 1992 , Lohr 1992].

However, many implicit control frameworks (i.e., abstractions for concurrency) exist, including path expressions [Campbell & Haberman 1974], synchronization counters [Robert & Verjus 1977 , McHale *et al.* 1990], behavior abstractions [Kafura & Lee 1990], and synchronizers [Agha *et al.* 1993]. Each has a different expressive power, and different properties regarding reusability of synchronization constraints. None is universal: within each, some problems are difficult to describe, and some problems cannot be described at all. Nevertheless, for its abstract nature and its reuse potential [Kafura & Lee 1989 , Frolund 1992], we believe that implicit control should be used as often as possible.

Synchronization expressed using explicit control is not defined declaratively, but is instead programmed. The languages CSP [Hoare 1978 , Hoare 1985], Ada [Burns 1985 , Gehani 1984 , Ichbiah *et al.* 1979], Occam [Inmos 1988], ABCL/1 [Yonezawa *et al.* 1987], POOL2 [America 1988], Concurrent C [Gehani & Roome 1989], and Eiffel// [Caromel 1993a] all use explicit control. Its first advantage is that, since the programmer actually programs a thread of control, all the expressive power associated with imperative language can be used. The detailed behavior of an active object can be specified, and fine tuning of policies is possible.

The second advantage of explicit control is that it allows programmers to describe the internal actions of processes, i.e., operations that are to execute independently of external requests. This kind of activity is difficult, or even impossible, to program with implicit control, since there is usually no place to mention the activation of personal actions. In any case, it is very hard to finely control the time spent executing such actions, and the time spent servicing externally-visible routines.

The third (and, to our group, decisive) advantage of explicit control is that under some conditions, it permits the design and construction of implicit control frameworks. Such abstractions can be put into libraries, from which programmers can chose the most appropriate for their needs [Caromel 1990c]. Then, within one language, one can always choose the type of control programming that best fits a given problem. In fact, explicit control may be viewed as a low-level mechanism, allowing either the definition of complex and precise synchronization, or the construction of a particular abstraction to be used for higher-level implicit control.

In summary, our argument is that:

1. programmers sometimes need explicit control;
2. implicit control permits the reuse of synchronization;
3. no universal implicit control abstraction exists; and
4. explicit control allows us to build implicit control abstractions.

As a consequence, the basic mechanism for programming process behavior in C++// is explicit control. Explicit control programming consists of defining the `Live()` routine of the `Process` class and its derived classes (Program 1.2) using the sequential control structures of C++.

All of the expressive power of C++ is available, without any limitation. For example, the process body of a bounded buffer can be defined as:

```
class Buffer1 : public Process {
  ⋮
  virtual void Live()
  {
    while (executing) {
      if (!full())
        explicit service of put
      if (!empty())
        explicit service of get
    }
  }
};
```

Besides explicit control, other features are needed in order to construct abstractions for concurrent programming. These features also provide C++// with a mechanism to explicitly service requests, the two instructions left unspecified in the example above.

First, defining a process's thread of control often consists of defining the synchronization of its public member functions. Since such an activity requires dynamic manipulation of C++ functions, we need to represent member functions as first-class objects. (In practice, only some limited features, such as the ability to use routines as parameters, and system-wide valid function identifiers, are needed.)

To fill that need, we provide the function `mid()` (for "method ID") to return function identifiers. Its usage is:

```
member_id f;
f = mid(put);
f = mid(A::put);
f = mid(A::put, A::get);
f = mid(A::put(int, P *));
```

In order to deal with overloading, this function returns either a single identifier, or a representation of all adequate functions. Related functions are defined in Section 6.3.

In the same way, because we need to explicitly program request servicing, we must be able to manipulate requests as objects (i.e., to pass them as parameters of other functions, to assign them to variables, and so on). In C++//, the class `Request` models requests. As shown below, every request is an instance of this class:

```
class Request {
 public:
  member_id m;                   // member to be called
  List<Any> eff_params;          // effective parameters
  Any object;                    // target object
  request_id id;                 // ID of the request
  :
};
```

A request object holds the identifier of the function to be invoked, the actual parameters of the invocation, a reference to the target object, an ID of the request, and information needed in order to implement the remote invocation. The class `Any` is used to manipulate any basic type of C++, and any pointer to a class or a structure. Since `Any` is implemented in standard C++, the basic type conversions are implicit, while conversions to class pointers must be explicit.

Finally, to be able to fully control request servicing, programmers must have access to the list of pending requests. This is given through the `Process` class, with a specific member named `request_list` that contains the list.

With these three facilities in place, it is possible to program the control of processes in diverse and flexible ways presented in the two next sections.

2.3. Library of Service Routines

Service primitives are needed to allow programmers to program control explicitly. Usually, programmers are given only a few such primitives, mainly because they are made part of the language itself as syntactical constructions (e.g., the `serve` instruction of Ada). With the primitives we define, it is possible to program a complete library of service routines [Caromel 1990a]. Some of these are shown in Program 1.3, where `f` and `g` are member identifiers obtained from the function `mid()` introduced in the previous section.

These functions are defined in the class `Process`, and can be used when programming the `Live()` routine. There are no limitations on the range of facilities that can be encapsulated in service routines. Timed services are an example of such expressiveness; selection based on the request parameters is another. Moreover, if a programmer does not find the particular selection function she needs, she is able to program it.

```
// Non-blocking services
serve_oldest();          // Serve the oldest request of all
serve_oldest(f);         // The oldest request on f
serve_oldest(f, g, ···);    // The oldest of f or g
...
serve_flush();           // Serve the oldest, wipe out the others
serve_flush(f);          // The oldest on f
...
// Blocking services: wait until there is actually a request to serve
bl_serve_oldest(f);      // blocking version of serve_oldest on f
bl_serve_flush();        // blocking version of serve_flush
...
// Timed blocking services: block for a limited time only
tm_serve_oldest(t);      // Serve the oldest, wait at most t
...
// Information retrieval
exist_request()          // true if a pending request exists
exist_request(f)         // True if a pending request exists on f
...
// Waiting primitives
wait_a_request();        // Wait until there is a request to serve
wait_a_request(f);       // Wait a request to serve on f
```

Program 1.3
A Library of Service Routines

Thus, libraries of service routines, specific to particular programmers or application domains, can be defined.

Another important point concerns efficiency: concurrency policies are determined within the context of each process, based on local information, rather than by using IPC. This avoids problems like polling bias [Gehani 1984]. This is an important advantage in distributed programming.

To illustrate the use of explicit control programming, Program 1.4 presents a C++// implementation of a bounded buffer. This definition implements a specific policy: when the buffer is neither `full()` nor `empty()`, the buffer alternates service on `put()` and `get()`. This policy is clearly not the only possible one. For example, some situations might require requests to be processed in the order of their arrival. Such a policy can be programmed as follows:

```
virtual void Live() {
  while (!stop) {
```

```
class Buffer: public Process, public List {
  ⋮
 protected:
  virtual void Live()
  {
    while (!stop) {
      if (!full())
        serve_oldest(mid(put));
      if (!empty())
        serve_oldest(mid(get));
    }
  }
};
```

Program 1.4
An Explicit Bounded Buffer Example

```
      if (full())
        serve_oldest(mid(get));
      else if (empty())
        serve_oldest(mid(put));
      else
        serve_oldest(mid(put), mid(get));
    }
  }
```

This is an example of explicitly fine-tuning the synchronization of processes. While this might be important in some contexts, in other contexts we might want to program within a more abstract framework, and ignore the implementation details. The next section shows how we allow such programming.

2.4. Library of Abstractions

Again, using the basic features defined in Section 2.2, it is possible to program the abstractions for concurrency control that are usually built into parallel languages [Caromel 1990c , Caromel 1991 , Caromel 1993c]. In order to define a new abstraction, one inherits from the `Process` class, and creates the desired synchronization behavior framework. In order to exploit the abstraction, users inherit from this class instead of from the `Process` class when defining an active object.

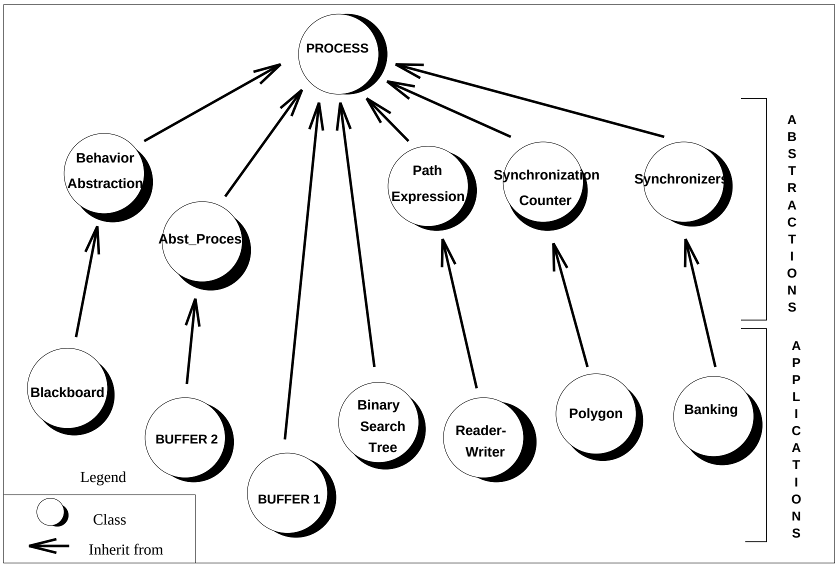

Figure 1.4
Library of Abstractions

For example, we can program a simple abstraction in which: a blocking condition (i.e., a function returning true or false) is associated with each public function, and a public function whose blocking condition is true is not served. The class `Abst_Process`, shown in Program 1.5, defines this framework. The function `associate()` permits users to specify blocking conditions for public functions; these associations are to be defined in the user class within the `synchronization()` function. The process body, which overrides the generic `Live()` routine, calls the `synchronization()` function once, and then loops to scan the pending request list, serving a request when its blocking condition is false. This `synchronization()` function is empty in the class `Abst_Process`, thus providing a default FIFO policy (no blocking condition).

This abstraction can be used to program a bounded buffer in an implicit style, as shown by the class `Buffer` in Program 1.6. Instead of inheriting from `Process`, this class uses the `Abst_Process` abstraction and, as previously, also inherits from the data structure `List`. The routine `synchronization()` defines the class's concurrency control; the

```
class Abst_Process: public Process, ··· {
 public:
  // associate a blocking condition bl with a function f
  void associate(member_id f, member_id bl) {
    ···    // synchronization to be defined
  }

  virtual void synchronization() {
    // empty: FIFO policy
  }

  virtual void Live() {
    synchronization();
    while (!stop)
      for (each pending request)
        if (blocking function is false) service request
  }
};
```

Program 1.5
An Example of Abstraction

functions `full()` and `empty()` are the blocking conditions of the public functions `put()` and `get()` respectively.

The definition of the buffer given in Program 1.6 is much more abstract than the one given in Program 1.4. In the former, control is specified declaratively, and is also non-deterministic, in that we have not specified whether the order of service respects the order of request arrival. In languages with explicit control, a specific instruction is usually needed in order to obtain non-determinism, such as the `select` instruction in Ada. This important capability is sometimes very much desirable for its abstraction, sometimes not. For example, it can be useful for a simple and concise specification of the next service over a set of member functions. In our framework, non-determinism is obtained, when needed, by normal programming, and made available through the library of abstractions (Figure 1.4), which can be specific to particular programmers or to application domains, and are extensible by final users. Figure 1.5 summarizes the basic features of the C++// model for control programming.

```
class Buffer2: public Abst_Process, public List {
  ⋮
 protected:
  virtual void synchronization() {
    associate(mid(put), mid(full));
    associate(mid(get), mid(empty));
  }
};
```

Program 1.6
An Implicit Bounded Buffer

- Processes have centralized and explicit control programming.
- Member functions and requests are first class objects.
- The list of pending requests is accessible.
- A library of service routines provides for explicit control programming.
- A library of abstractions allows for implicit and declarative control.

Figure 1.5
Control Programming in C++//

3. A Programming Method

This section presents how to use C++// to program parallel and distributed applications. Because it is rather difficult to evaluate performances of distributed systems before they actually run, we believe that definition of processes has to be postponed as much as possible, and should be flexible and adaptable.

3.1. Sequential Design and Programming (step 1)

The first step is a standard, sequential, object-oriented design [Booch 1986 , Booch 1987 , Halbert & O'Brien 1987 , Meyer 1988]. The only thing that matters is, before defining parallel activities, to have a fully sequential implementation: we are then able to conduct tests to ensure the correctness of the sequential algorithms and implementation. The next three steps deal with parallel design and are specific to C++//.

3.2. Process Identification (step 2)

Processes are a subset of classes. Object-oriented design usually gives a finer-grained decomposition than structured design, so there is no need for restructuring. This step is therefore defined by two successive phases.

(3.2.a) Initial Activities In a concurrent system, there are points where the activity starts. Our method uses these sources of *initial activity* to structure a system into processes. The objects where an activity takes place will be the initial processes.

There are various cases, heavily depending on the application domain: *active objects* for parallelization, *control objects* that continuously ensure their function of control, *event dependent objects* that need to be activated at dedicated speed or asynchronously, all the periodic or asynchronous I/O, etc.

(3.2.b) Shared Objects At this point, a set of classes are defined to be processes and we know the system topology. The concurrent model focuses on each object referred to by at least two processes. If the object can be passed by copy, there is nothing to change, the model automatically ensures this behavior. If it appears that the object really needs to be shared, a process class has to be programmed. This rule leads to identification of new processes.

3.3. Process programming (step 3)

In this section we program the processes identified during the previous step. The classes that remain passive are commonly used without any changes.

(3.3.a) Define Process Classes The method to directly program the processes relies on the fact that each class is a potential process.

First, it has to be decided which technique is the most appropriate for programming the concurrency control of the process: explicitly (using the basic `Process` class), or implicitly (choosing and using one abstraction). Then, the new process class is a new class that derives from the corresponding passive class and either the class `Process`, or the selected abstraction.

Because IPCs are unified with member function calls, we are able to use the methods defined in the ancestor. However, not every original public operation will be used. Since we program the process class, new interfaces reflecting the concurrent activities can be defined.

(3.3.b) Define the Activity A process class is given a default FIFO body. There are mainly two, possibly simultaneous, cases where we need to change the process body by redefining the control method: the *synchronization* is not FIFO, or the process carries on *internal activities* besides the request services. This is not a problem since the programmer has all the control to tune up fine policies regarding the relative time to spend among the different activities.

(3.3.c) Use the process classes In order to use the process classes defined previously, all the objects identified in step 2. as processes need to be assigned with an instance of the corresponding process. An entity `a` declared as: `A* a;` has to be assigned with a process object of type `P_A` (derived class of `A`): `a = new P_A ( ···)`. A function call on `a` is now executed on an asynchronous basis (the caller does not wait for its completion). This automatic transformation of synchronous calls into asynchronous ones is crucial to avoid method redefinition. An inherited method may use the result of a function call issued to the process:

```
res = a->fct(parameters);
...
res->g(parameters);
```

In this case, the *wait-by-necessity* handles the situation. Without this automatic *data-driven synchronization* one would have to redefine the routine in order to add explicit synchronization.

3.4. Adaptations (step 4)

At this stage, the system starts running and efficiency tests are realized. This last step is a system refinement in order to match parallel specifications.

(3.4.a) Refine the Topology If a value is kept in another process context, it can be obtained in two different ways: (1) by requesting the object that holds the data, (2) by receiving the data asynchronously through a public method. Generally a refinement in the topology will transform a situation (1) into a situation (2). The aim is usually to globally minimize the number of interprocess communications. Such a modification leads to design new classes by inheritance. A process will receive the value it needs through a new public method.

(3.4.b) Define New Processes There are various reasons leading to define new processes: *buffering processes* can be useful (even if the model is asynchronous), *secretary* process can trim the workload of another process, an object detected to be *computationally intensive* may need to be transformed into a process to map it onto another processor, etc. Since this phase identifies new processes, we again apply rule 3.2.b about shared objects. Finally, we apply step 3 in order to program the new process classes. Step 4 (*Adaptation to constraints*) is realized repeatedly.

The method description is complete.

4. Environment

This section describes the facilities that support the development of C++// programs, including compilation of source code, executable generation, and a mechanism for mapping active objects onto machines.

4.1. Compilation

Compilation is achieved by pre-processing a program's source files using the command `c++ll`. The pre-processor does not modify the user classes, but instead generates extra code in separate files. For each user file, a corresponding C++ file is generated by the C++// system. These generated files and the original user files are then compiled with a standard C++ compiler. All files are finally linked together with the C++// library (Figure 1.6).

For each source file, `file.cc`, code generation is achieved in two phases, which are transparent to the user. The first phase analyses the source code and generates an information file named `file-ll` in a directory called `.c++ll` below the directory of the original file. The second phase generates a C++ file (`file-ll.cc`).

The file `.c++ll-config` contains general information regarding the user's personal settings of the C++// installation. Information specific to each C++// system is specified in the file `.c++ll-system` in order to produce an executable from a set of files.

4.2. Mapping

Mapping assigns each active object created during the execution of a C++// program to an operating system process on an actual machine or processor. In order to avoid confusion, we call the sub-system consisting of one active object and all its passive objects a *language process*, and use the term *OS process* for the usual notion of an operating system process.

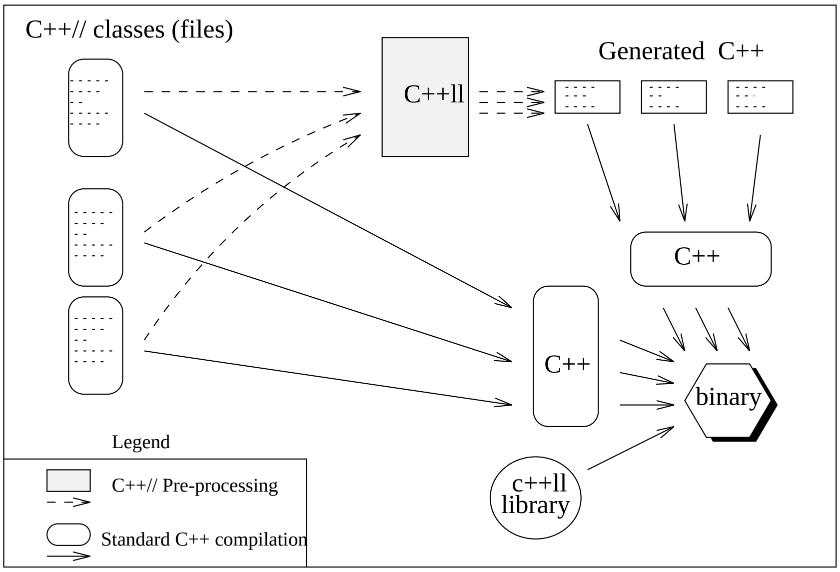

Figure 1.6
Compilation of a C++// System

The mapping of a language process to an OS process on a particular processor is controlled by the programmer through the association of the machine where the process is to be created, and its light-weight or heavy-weight nature. The machine itself can be specified in two ways. The first method is to specify a virtual machine name, which is simply a string. This name is related to an actual machine name by a translation file called `.c++ll-mapping`. The C++// system looks for this file first in the directory in which the process is running, and, if it is not found there, in the user's home directory. An example of such file is:

```
// .c++ll-mapping
// virtual name   actual name
   Mach_A         Nice
   Mach_B         cannes.unice.fr
   Mach_C         alto.unice.fr
   Names          monaco
   Server         Inria.Sophia.fr
   S1             wilpena.unice.fr
   S2             192.134.39.96
   S3             // current machine
   P1             I3S-1
```

```
 :              :
 P6             INRIA-1
```

The other technique used to specify a machine is to use a language process that already exists. In this case, the new process is created on the machine where that language process is running. With this technique, processes can be linked together to ensure locality. We say that a process is *anchored* to another one, because its mapping will automatically follow that specified for the process it is anchored to. An anchor can transitively reference another anchored process.

The *light-weight* switch permits creation of several language processes inside a single OS process. In the *heavy-weight* case, only one language process is mapped to each OS process. Figure 1.7 presents the semantics of the association of the two criteria. The user accesses these switches through a class called `Mapping`:

```
class Mapping {
 public:
  virtual void on_machine(const String& m); // virtual machine name
  virtual void with_process(Process* p);
    // set the machine to be the same as for the existing process p

  virtual void set_light();      // set to light-weight process
  virtual void set_heavy();      // set to heavy-weight (OS) process
};
```

Process nature		Machine: Virtual machine M	Machine: Machine where the language process p is running
	Light	On machine actual_name (M) On an arbitrary existing OS process (a new one if none)	Same machine and OS process as p
	Heavy	On machine actual_name (M) On a new OS process	Same machine as p On a new OS process

Figure 1.7
Combination of the Mapping Criteria

```
A *p1;     // A is a normal sequential class: allocation-based style
P_A *p2, *p3;          // P_A is a process class: class-based style
Mapping *map1, *map2; // mapping objects
⋮
map1->set_heavy();
map1->on_machine("Server");
p1 = (A *) new (typeid(A), map1) Process_alloc(···);
                       // p1 on a new OS process,
                       // on machine with actual name "Server"
⋮
map2->set_heavy();
map2->with_process(p1);
p2 = new (map2)  P_A(···);
                       // p2 on a new heavy-weight process,
                       // same machine as p1
map2->set_light();
p3 = new (map2) P_A(···);
                       // p3 on a light-weight process,
                       // same machine as p1,
                       // inside the same OS process as p1
map2->on_machine("P1");
for (int i=0; i < 100 ; i++)
   t[i] = new (map2) P_A(···);
                       // t[i] on a light-weight process,
                       // on machine with actual name "P1",
                       // all within the same OS process
```

Program 1.7
Mapping Processes to Machines

When a program creates a language process, an object of type `Mapping` can be passed to the allocator (`new()`) in order to specify the desired mapping of the new process. Program 1.7 presents the syntax used for this. With the `.c++ll-mapping` file taken from above, Program 1.7 produces the mapping presented in Figure 1.8. Note that a language process, even created as heavy-weight, becomes light-weight if a new language process is mapped onto its OS process; this is the case for the `p1` process.

Besides this local control of mapping, there is a global variable that is valid within each C++// sub-system:

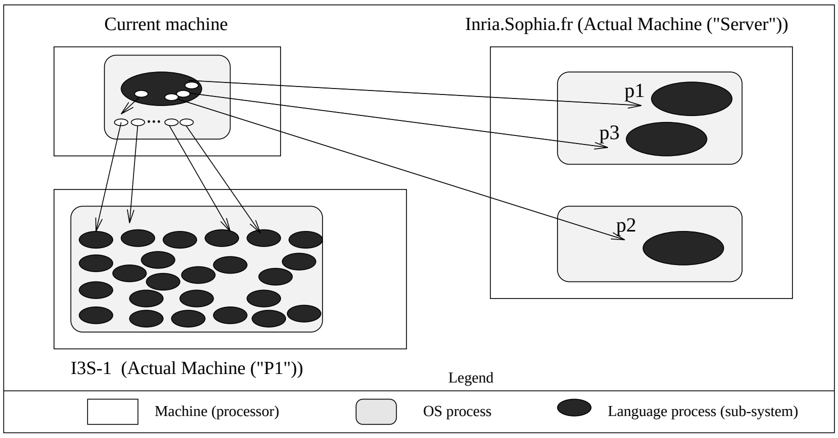

Figure 1.8
Example of Mapping

```
Mapping* mapping;
```

which permits the establishment of a global mapping strategy. This variable is accessible to all the objects of the sub-system, and is used by default during process creation when no mapping object is passed. The scope of `mapping` is the process sub-system; each sub-system has its own copy of this variable. Using this functionality, a global mapping strategy accessible throughout a program can be implemented by constructing a centralized server (a C++// process) which on request returns a mapping object to be used for subsequent process creation.

Our intention is to develop more sophisticated mapping strategies by deriving new classes from the `Mapping` class. For example, we are currently defining `Cluster` classes to allow processes to be grouped and managed in a more abstract manner. In the longer term, we aim to develop automatic or semi-automatic load-balancing classes that will rely on modelling and evaluation of machine and network load.

5. Polygon Overlay

5.1. Sequential Design and Programming (step 1)

We implement the overlay of two polygon maps using a simple pipeline. We define the class `polygon` (Program 1.8), whose objects can test their

```
class polygon {
public:
  polygon (int x1, int y1, int x2, int y2, polygon* n=0) :
    xl(x1), yl(y1), xh(x2), yh(y2), next(n)
    { area = (x2-x1) * (y2-y1); }

  virtual polygon* get_next () { return next; }

  // set_next call will be polymorphic (polygon) in the parallel classes
  virtual void set_next (polygon * n) { next = n; }
  ⋮
  virtual void overlay (polygon * map2_polygon) {
    if (map2_polygon->get_area())
      if (!area) next->overlay(map2_polygon);
      else if (!intersection(map2_polygon))
             next->overlay(map2_polygon);
           else {
             produce_intersection(map2_polygon);
             if (map2_polygon->get_area())
               next->overlay(map2_polygon);
           }
  }
 protected:
  int xl, yl, xh, yh, area;
  polygon * next;
};
```

Program 1.8
Polygon Class

intersection with other polygons. The pipeline is a list of objects of class `polygon`. It is initialized with the contents of the first map, and is fed with polygons from the second map.

This algorithm is enhanced by the addition of an `area` field to the class `polygon`. This permits us to stop propagating a polygon from the second map through the pipeline whenever it has been completely accounted for by the polygons of the first map. Symmetrically, it allows us to remove a polygon from the first map when its area has been totally consumed.

```
#include "polygon.h"

polygon * tmpPoly;
polygon * map1 = 0;

void read_map1(void) {
  for(int i=0; i<nbpoly1; i++) { ··· // read from file
    map1 = new polygon(xl, yl, xh, yh, map1);
  }
}

void read_and_overlay_map2(void) {
  for(int i=0; i<nbpoly2; i++) { ··· // read from file
    tmpPoly = new polygon(xl, yl, xh, yh, 0);
    map1->overlay(tmpPoly);   // possible stack overflow!
  }
}

// the first process
int main(int argc, char * argv[]) {
  read_map1();
  read_and_overlay_map2();
}
```

Program 1.9
Pipeline Construction

The construction of the pipeline is straightforward: polygons from the first map are read and linked together, and then polygons from the second map are passed to the pipeline (Program 1.9).

5.2. Parallel Programming

5.2.1. Light-Weight Processes Once the sequential polygon pipeline has been written and debugged (method step 1), it can be parallelized. The pipeline polygons can be identified as active (step 2). The simplest parallel version can be obtained by deriving a new class (Program 1.10) from the sequential class `polygon` and the system class `Process` (step 3.1). In this version, each method call to a pipeline polygon object will be transformed into a request to a polygon process; its service policy needs only be FIFO (step 3.2).

```
#include "Process.h"
#include "polygon.h"

class ppolygon : public polygon, public Process {
 public:
  ppolygon (int x1, int y1, int x2, int y2, ppolygon2* n=0) :
    Process(), polygon(x1, y1, x2, y2, n)   { }
};
```

Program 1.10
Ppolygon: a Light-Weight Process Polygon Class

Of course, this parallelization option is practical for large data sets only if the environment is multi-threaded.

We must also redefine the construction of the polygons of the first map (i.e., create process polygons). Program 1.11, maps light-weight processes to three different virtual machines.

5.2.2. Heavy-Weight Processes If the underlying system only provides heavy-weight processes, parallelization is still possible, but will be best done by combining active parallel polygons (Program 1.12) and passive sequential polygons to program the pipeline (steps 2 and 3).

In Figure 1.9, we show the system of active and passive objects created by this program and the passing of a polygon from the second map to the pipeline. This polygon will be forwarded through the pipeline until its area has been consumed by all its intersections. In the figure, for example, the polygon from the second map is not forwarded beyond the second language process.

Note that the same `Process` class should serve for both the light-weight and heavy-weight versions, since it only denotes a language process. However, the mapping description is different in the two main programs. Mapping should be the only place where we have to indicate whether the language process should be mapped onto the same OS process.

Note also that the sequential version is programmed with terminal recursion. With large input maps, this can produce a stack overflow during the execution. Therefore, we have also programmed a recursionless version. It is a little less natural, but is able to process large maps. The parallel version, although normally suffering from the same problem, is able to handle big maps: indeed, when the pipeline is split between

```
Mapping* light_mapping = new Mapping;
char *machines[3] = {"Mach_A", "Mach_B", "Mach_C"};
int current_machine=0;
⋮
void read_map1(void) {
  light_mapping->set_light();
  for(int i=0; i<nbpoly1; i++) { ···
    light_mapping->set_machine(machines[current_machine]);
    if (current_machine < 2) current_machine++;
      else current_machine = 0;
    // polymorphic assignment of result
    map1 = new(light_mapping) ppolygon(xl, yl, xh, yh, map1);
  }
}
```

Program 1.11
Modifications to the Main Routine

```
class ppolygon2 : public polygon, public Process {
 public:
  ppolygon2(int x1, int y1, int x2, int y2, ppolygon* n,
   char *fname, long int pos, int nb) : polygon(x1, y1, x2, y2, n) {
    polygon *p;
    int xa1, ya1, xa2, ya2;

    // Read portion of map1 from file fname

    if (nb) {    // read last passive polygon (if any):
      p = read_polygon_from_file();
      p->set_next(next); next = p; // point to the next process
    }

    // read other passive polygons (if any):
    for(int i=1; (i<nb) && !feof(f); i++) {
      p = read_polygon_from_file();
      p->set_next(next); next = p;
    }
    area = (xa2-xa1) * (ya2-ya1);
  }
};
```

Program 1.12
Ppolygon2: - a Heavy-Weight Process Polygon Class

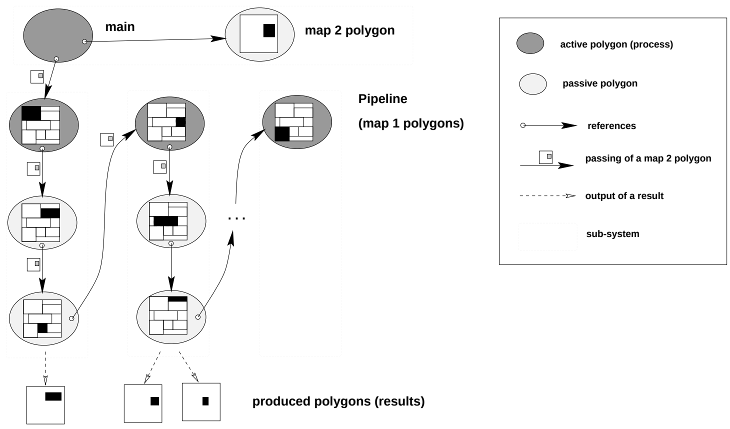

Figure 1.9
Heavy-Weight Polygon Process Pipeline

several processes, the stack is partially split between pending requests lists. For the same reason, there are no restrictions about stack size on the size of maps in the light-weight process version.

For performance figures, we refer the reader to our web pages (`http://www.inria.fr/sloop/c++ll` or `http://wwwi3s.unice.fr/c++ll`).

6. Implementation

C++// currently runs on the following platforms: DEC AlphaStation 200 4/166, Sun SparcStation 4C (IPC), Sun Sparc 4D (multi-processor), and PC ix86. Operating systems supported include DEC Unix 3.0 (formerly DEC OSF/1), SunOs 4.1.3, Solaris 2, and Linux 1.2.8. We are using the GNU compiler and library version 2.7.0; PVM 3.3.7 and PVM 3.3.8 for interprocess communication [Furmento & Baude 1995], and are running on a 10Mb/s Ethernet. The system will be ported to other platforms in the near future, since its only requirements are a C++ compiler supporting RTTI and a PVM library (though the language is not tied to PVM, and could use a simpler communication library).

Below, we describe the construction of the C++// environment. This presentation goes beyond implementation details since the technique we use—reification—also supports the customization and extension of our system, as laid out in Section 6.4.

6.1. A Reflection-Based System

The C++// system is based on a Meta-Object Protocol (MOP) [Kiczales *et al.* 1991]. There are various MOPs for different languages and systems, with various goals, compilation and run-time costs, and various levels of expressiveness. Within our context, we use a reflection mechanism based on reification. Reification is simply the action of transforming a call issued to an object into an object itself; we say that the call is "reified". From this transformation, the call can be manipulated as a first class entity (i.e., stored in a data structure, passed as parameter, sent to another process, etc).

A *meta-object* (Figure 1.10) captures each call directed towards a normal *base-level* object; a meta-object is an instance of a *meta-class*. In some ways, a local object that provides a remote object with local access, i.e. a *proxy* [Shapiro 1986 , Edelson 1992 , Makpangou *et al.* 1994 , Dave *et al.* 1992 , Birrell *et al.* 1995], is a kind of meta-object.

MOP techniques have been used in many contexts to provide an elegant model of various concepts, and an extensible design and implementation of various language features or extensions (such as remote objects). An important work has been the CLOS MOP [Bobrow *et al.* 1988] . In that case, even the semantics of inheritance is extensible through meta-object programming (something which we do not support). The Eiffel// language [Caromel 1990a , Caromel 1993b] also used a meta-level for reification. MOPs have a wide scope of applications, and are an active field of research for parallel and distributed programming [Chiba & Masuda 1992 , Watanabe & Yonezawa 1988 , Yokote & Tokoro 1986 , Madany *et al.* 1992 , Buschmann *et al.* 1992 , McAffer 1995 , Chiba 1995]. Work using more traditional methods, such as *proxy generators*, are closely related [Birrell *et al.* 1995].

6.2. A MOP for C++: Basic Classes

The first principle of our MOP for C++ is embodied in a special class, called `Reflect`. All classes inheriting publicly from `Reflect`, either directly or indirectly, are called *reified classes*. A reified class has *reified instances*; all calls issued to a reified object are reified. This last requirement is important for reusability, as it permits users to take a normal class, and then globally modify its behavior, in order to transform it into a process.

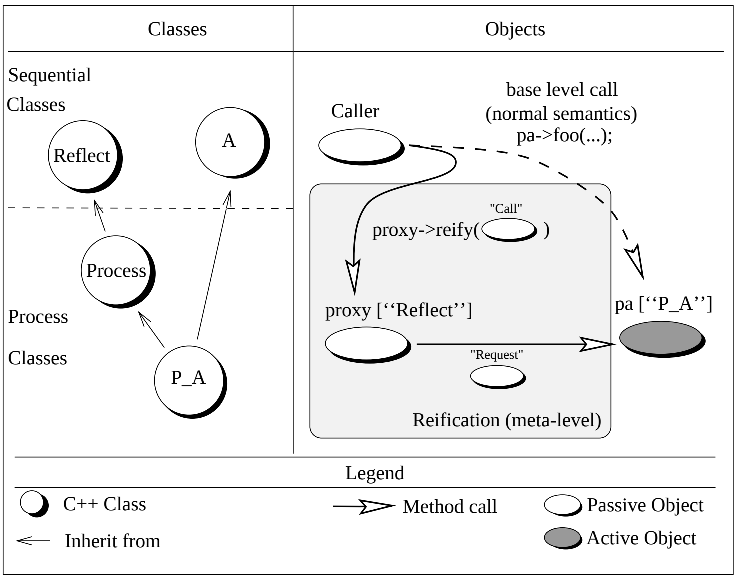

Figure 1.10
Reification of Calls

The `Reflect` class implements the reflection mechanism with reification:

```
class Reflect {
 public:
  virtual void reify(Call* c) {
    c->execute();   // a call reification
  }
  void* operator new (size_t s, type_info& t) {
    ⋮
  }
  Reflect(···) {
    ⋮
  }
};
```

The creation of an instance of a `Reflect` class returns a meta-object (in our case, a proxy) for the type being passed in as a parameter of the allocator. `type_info` is the standard RTTI class of C++.

All calls issued to instances of the `Reflect` class and its derived classes will trigger the execution of the member function `reify()` with the appropriate object of type `Call` as a parameter:

```
class Call {
 public:
  virtual void execute();
  List<Any> eff_params;          // effective parameters
  member_id m;                   // member to be called
  Any object;                    // target object
  Any result_place;              // result address
};
```

Instances of the `Call` class are reified calls (i.e., objects which represent the reification of calls). Figure 1.10 illustrates reification.

From this mechanism, we implement the basic classes of our programming model described in Section 1.1. For example, the class `Process_alloc` inherits from `Reflect`, and redefines the `reify()` function as:

```
class Process_alloc: public Reflect {
 public:
  virtual void reify(Call* c) {
    ⋮            // send the request to the remote process
  }
  void* operator new (size_t s, type_info& t) {
    ⋮            // process creation
  }
  Process_alloc(···) : Reflect (···) {
    ⋮
  }
};
```

It is at this point, within such routines, that a mapping between the primitives needed to implement a specific model of parallel programming (such as point-to-point communication or broadcasting) and the actual platform primitives will take place. This permits our system to use the most efficient primitives available on a given architecture.

6.3. Class and Member Identification

Within the framework of multiple address spaces, the need for class and member identifiers is inescapable. While the classes presented in the previous section can have simple and low-cost representations in a shared address space, more sophisticated policies are needed in a distributed environment.

Another issue is the capability to automatically marshal and unmarshal objects between processes and address spaces. The `Class_info` and `Member_info` classes respond to these specific issues:

```
class Class_info {
 public:
  class_id id;               // system-wide valid identification
  structure s;               // information on the class structure
                             // for (un)marshalling of objects
  List<Any> flat(Any obj);  // flattening an object
  Any build(List<Any> l);   // building an object
  ⋮
};

class Member_info {
 public:
  member_id id;             // system-wide valid identification
  Class_info* return_type;  // information on the member
  ⋮
};
```

These classes are meta-classes representing information on C++ classes. They conform to the design principle of the standard class `type_info`, which provides some basic information, and which was designed with the intention of being extended according to specific needs [Stroustrup 1994].

Two functions are used to obtain the class and member IDs:

```
class_id  cid(A);      // class Id
member_id mid(put);    // member Id
```

The `mid()` primitive was presented in Section 2.2 for member manipulation, and used in Sections 2.3 and 2.4 for programming the control of processes. By construction, `class_id` and `member_id` have system-wide validity (i.e., can be passed as parameters between processes), while the `Class_info*` and `Member_info*` pointers have a specific value within each address space. Two functions:

```
Class_info*  cid_class(class_id c);      // from cid to class
Member_info* mid_member(member_id m);    // from mid to member
```

provide a mapping in a given address space between class and member identifications and local `Class_info*` and `Member_info*` addresses. For convenience, two other functions provide direct access to the objects representing a class and a member function, respectively:

```
Class_info*  typeid_ll(A);        // class representation
Member_info* memberid_ll(put);    // member representation
```

These are similar to the RTTI `typeid()` operator of standard C++. Finally, the following function allows direction translation from RTTI class representation to C++// class identifiers:

```
class_id  rtti_ll(type_info& t); // RTTI to C++// class id
```

In our distributed framework, the class `Request` (introduced in Section 2.2) is implemented through inheritance from the MOP class `Call`, and defines only the new members corresponding to the specific information which is needed:

```
class Request: public Call {
 public:
  virtual void execute();     // new definition
  request_id id;              // id of the request
  ⋮
};
```

An instance of this class is then sent to the remote process, which executes the call by calling the `execute()` function on it.

6.4. Customization and Extension of C++//

The MOP presented above is independent of any parallel programming model. The MIMD model we described in this paper is programmed on top of the MOP, without any compiler modification. All the classes described in the model, such as `Process` and `Process_alloc`, are defined as library classes using the basic `Reflect` and `Call` classes and the class and member identification primitives. An important consequence of this is that other parallel programming models, such as shared-memory MIMD or SPMD, can be defined on top of the MOP.

As well as supporting libraries of concurrent programming models, this technique also permits us to address extension issues. We might provide a debugging environment and tracing, or use efficient platform-specific primitives for communication (e.g. broadcast and multicast communication), since specific behaviors can be added at the reification stage. It is also worth noting that this mechanism allows us to handle issues

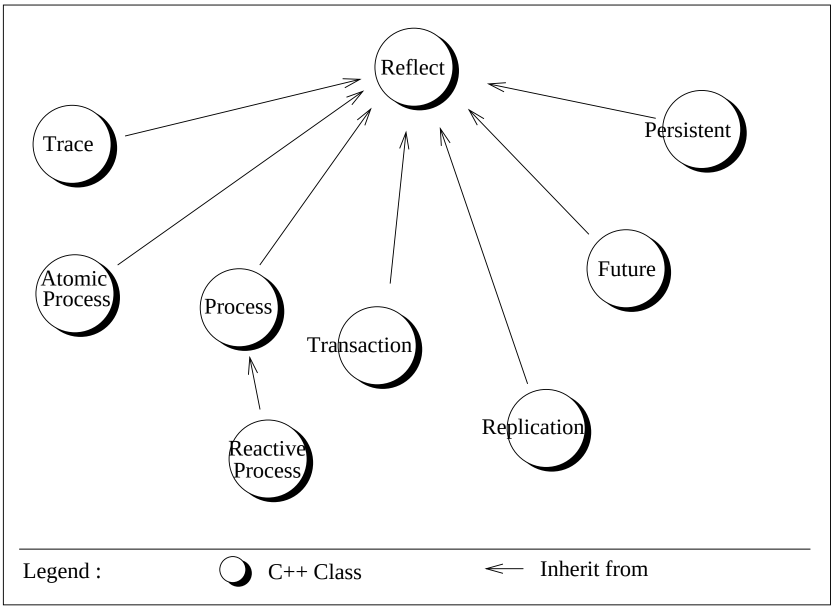

Figure 1.11
Customization and Extension of C++//

such as persistent objects, atomicity, replication, and fault tolerance (Figure 1.11). Wait-by-necessity, for example, is implemented through a class `Future`, which uses reification by inheriting from `Reflect`. Such an open system, or open implementation [Kiczales *et al.* 1991], is extensible by the end user, and adaptable to various needs and situations.

Conclusion

Our system currently has several limitations. First, basic types are not subject to wait-by-necessity; this can only be achieved by encapsulating them in classes. While this can be a problem regarding reusability, the runtime cost of adding synchronization to basic types was felt to be too high.

The fact that data members cannot be accessed transparently on a process object is a partial limitation of our model. However, since C++

does not provide uniform access to data or function members, this constraint is unlikely to be removed in the future.

Light-weight language processes and reflective template classes are not yet implemented. We are currently working on these important features.

Finally, to support the use of polymorphism between standard passive objects and process objects, all public functions have to be virtual; otherwise, non-virtual function calls will not be transformed into IPCs. This limitation comes directly from C++, which does not provide dynamic binding by default on all members. This drawback can be alleviated if the C++ compiler provides an "all-virtual" option. The choice here is between reusability, and paying the price of having all functions virtual. Of course, making such a change requires recompiling all of files involved, but this is a small price to pay compared to the benefits of code reuse.

Our work focussed on reuse, flexibility, and extensibility. At different levels—service routines, abstractions for control programming, and libraries defining specific programming models—the system we propose tries to both conform to information hiding principles, and to be open for customization and extension. We feel this approach is at least a partial solution to the complexity and diversity of parallel programming.

Granularity is another crucial point of parallel programming. Achieving an appropriate match between the granularity of program activities, and the capability of the underlying parallel architecture, is a challenging part of high-performance programming. We believe that the reusability and flexibility of object-oriented languages allows us to address this problem.

Finally, another important aspect of distributed programming is the correctness issues it raises. These were not addressed in this chapter, but this is another area of investigation for our group [Attali *et al.* 1993 , Attali & Caromel 1995]. We hope formal techniques, together with parallel object-oriented programming, will permit advances to be made.

Acknowledgments

The authors gratefully acknowledge the help and support of the SLOOP project members. The work of Françoise Baude and Ph.D. student Nathalie Furmento on interprocess communication was decisive to the current system. Discussions, and joint work with collaborators Jean-Claude Bermond, Bruno Gaujal, Philippe Mussi, and Michel Syska, and Ph.D. students Olivier Dalle, Günther Siegel, Olivier Delmas, and Stéphane Perennes have been of first importance.

Bibliography

[Agha 1986] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Agha *et al.* 1993] G. Agha, S. Frølund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology*, May 1993.

[America 1988] P. America. Definition of POOL2, a Parallel Object-Oriented Language. Technical Report 364, Philips Research Laboratories, April 1988. ESPRIT Project 415.

[Attali & Caromel 1995] I. Attali, D. Caromel, and A. Wendelborn. A Formal Semantics and an Interactive Environment for Sisal. In A. Zaky, editor, *Tools and Environments for Parallel and Distributed Systems*, pages 231–258. Kluwer, 1995. to appear.

[Attali *et al.* 1993] I. Attali, D. Caromel, and M. Oudshoorn. A Formal Definition of the Dynamic Semantics of the Eiffel Language. In G. Gupta, G. Mohay, and R. Topor, editors, *Sixteenth Australian Computer Science Conference (ACSC-16)*, pages 109–120. Griffith University, February 1993.

[Birrell *et al.* 1995] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. Technical Report SRC-RR-115, DEC Systems Research Center, 1995.

[Bobrow *et al.* 1988] D.G. Bobrow, L.G. DiMichiel, R.P. Gabriel, S.E. Keen, G. Kiczales, and D.A. Moon. Common Lisp Object System Specification: X3J13 document 88-002R. *SIGPAN Notices*, 23, September 1988.

[Booch 1986] G. Booch. Object-Oriented Development. *IEEE Transaction on Software Engineering*, February 1986.

[Booch 1987] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings, Second edition, 1987.

[Burns 1985] A. Burns. *Concurrent Programming in ADA*. Cambridge University Press, 1985.

[Buschmann *et al.* 1992] F. Buschmann, K. Kiefer, F. Paulish, and M. Stal. The Meta-Information-Protocol: Run-Time Type Information for C++. In A. Yonezawa and B.C. Smith, editors, *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 82–87, 1992.

[Campbell & Haberman 1974] R.H. Campbell and A.N. Haberman. The Specification of Process Synchronization by Path Expression. In *Colloque sur les Aspects Théoriques et Pratiques des Systèmes d'Exploitation, Paris*, 1974.

[Caromel 1989] D. Caromel. Service, Asynchrony and Wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1989.

[Caromel 1990a] D. Caromel. Concurrency: an Object Oriented Approach. In J. Bezivin, B. Meyer, and J.-M. Nerson, editors, *Technology of Object-Oriented Languages and Systems (TOOLS'90)*, pages 183–197. Angkor, June 1990.

[Caromel 1990b] D. Caromel. Concurrency and Reusability: From Sequential to Parallel. *Journal of Object-Oriented Programming*, 3(3):34–42, September 1990.

[Caromel 1990c] D. Caromel. Programming Abstractions for Concurrent Programming. In J. Bezivin, B. Meyer, J. Potter, and M. Tokoro, editors, *Technology of Object-Oriented Languages and Systems (TOOLS Pacific'90)*, pages 245–253. TOOLS Pacific, November 1990.

[Caromel 1991] D. Caromel. A Solution to the Explicit/Implicit Control Dilemma. *Object-Oriented Programming Systems Messenger*, 2(2), April 1991.

[Caromel 1993a] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[Caromel 1993b] D. Caromel and M. Rebuffel. Object Based Concurrency: Ten Language Features to Achieve Reuse. In R. Ege, M. Singh, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems (TOOLS USA'93)*, pages 205–214. Prentice Hall, August 1993.

[Caromel 1993c] D. Caromel. Abstract Control Types for Concurrency (Position Statement for the panel : *How could object-oriented concepts and parallelism cohabit ?*). In L. O'Conner, editor, *International Conference on Computer Languages (IEEE ICCL'94)*, pages 205–214. IEEE Computer Society Press, August 1993.

[Chiba & Masuda 1992] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with Meta-Level Architecture. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*, pages 482–501, Kaiserslautern, July 1993. Springer-Verlag.

[Chiba 1995] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, volume 30 of *ACM Sigplan Notices*, pages 285–299, Austin, Texas, October 1995. ACM Press.

[Dave *et al.* 1992] A. Dave, M. Sefika, and R.H. Campbell. Proxies, Application Interfaces and Distributed Systems. In *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems (OOOS)*. IEEE Computer Society Press, September 1992.

[Decouchant *et al.* 1989] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X. Rousset de Pina. A Synchronization Mechanism for Typed Objects. *SIGPLAN Notices*, 24(4):105–107, April 1989.

[Decouchant *et al.* 1991] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, and X. Rousset de Pina. A Synchronization Mechanism for an Object-Oriented Distributed System. In *IEEE Eleventh International Conference on Distributed Computing Systems*, 1991.

[Edelson 1992] D. Edelson. Smart Pointers: They're Smart, but They're Not Pointers. In *Proceedings of the Usenix C++ Conference*, August 1992.

[Frolund 1992] S. Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming. In *ECOOP '92 proceedings*, June 1992.

[Furmento & Baude 1995] N. Furmento and F. Baude. Design and Implementation of Communications for the C++// System. Technical Report RR 95-50 I3S, CNRS I3S - Univ. de Nice - INRIA Sophia Antipolis, 1995.

[Gehani & Roome 1989] N. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989.

[Gehani 1984] N. Gehani. *ADA Concurrent Programming*. Prentice-Hall, 1984.

[Gehani 1984] N. Gehani. Concurrent Programming in the ADA Language: the Polling Bias. *Software–Practice and Experience*, 14(5), 1984.

[Halbert & O'Brien 1987] D.C. Halbert and P.D. O'Brien. Using Types and Inheritance in Object-Oriented Languages. In *European Conference on Object-Oriented Programming*, June 1987.

[Halstead 1985] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[Hewitt 1977] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Hoare 1978] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), August 1978.

[Hoare 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Ichbiah *et al.* 1979] J.D. *et al* Ichbiah. ADA Reference Manual and Rationale for the Design of the ADA Programming Language. *SIGPLAN Notices*, 14(6), 1979.

[Inmos 1988] Inmos Ltd. *Occam 2 Reference Manual.* Prentice Hall, 1988.

[Kafura & Lee 1989] D.G. Kafura and K.H. Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. *The Computer Journal*, 32(4), 1989.

[Kafura & Lee 1990] D.G. Kafura and K.H. Lee. ACT++: Building a Concurrent C++ with Actors. *Journal of Object-Oriented Programming*, 3(1), May 1990.

[Kiczales *et al.* 1991] G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Lieberman 1987] H. Lieberman. Concurrent Object-Oriented Programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

[Lohr 1992] K.-P. Lohr. Concurrency Annotations Improve Reusability. In *TOOLS USA'92*, August 1992.

[Madany *et al.* 1992] P. Madany, N. Islam, P. Kougiouris, and R.H. Campbell. Practical Examples of Reification and Reflection in C++. In A. Yonezawa and B.C. Smith, editors, *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 76–81, 1992.

[Makpangou *et al.* 1994] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. Fragmented Objects for Distributed Abstractions. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems.* IEEE Computer Society Press, 1994.

[Mallet & Mussi 1993] L. Mallet and P. Mussi. Object Oriented Parallel Discrete Event Simulation: The PROSIT Approach. In A. Pave, editor, *Modelling and Simulation FSM 93.* Society for Computer Simulation, June 1993.

[Matsuoka *et al.* 1990] S. Matsuoka, K. Wakita, and A. Yonezawa. Synchronization Constraints With Inheritance: What is Not Possible—So What Is? Technical Report Technical Report 10, The University of Tokyo, Department of Information Science, 1990.

[McAffer 1995] J. McAffer. Meta-level Programming with CodA. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, (Aarhus, Denmark), August 1995. also available from `ftp://camille.is.s.u-tokyo.ac.jp/pub/members/jeff/docs/ecoop95.a4.ps.gz`.

[McHale *et al.* 1990] C. McHale, B. Walsh, S. Baker, A. Donnelly, and N. Harria. Extending Synchronisation Counters. Technical Report TCD-Pub-0011, University of Dublin, Trinity College, July 1990. ESPRIT Project Comandos, number 834 and 2071.

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

[Meyer 1992] B. Meyer. *Eiffel: The Language (Version 3).* Prentice–Hall, 1992.

[Neusius 1991] C. Neusius. Synchronizing Actions. In *ECOOP '91 Proceedings*, June 1991.

[Nierstrasz 1987] O.M. Nierstrasz. Active Objects In Hybrid. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1987.

[Robert & Verjus 1977] P. Robert and J.-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In B. Gilchrist, editor, *Proc. IFIP Congress*, pages 981–986, North-Holland, 1977.

[Shapiro 1986] M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Mass. (USA)*, pages 198–204. IEEE, May 1986.

[Shibayama 1991] E. Shibayama. Reuse of Concurrent Object Descriptions. In A. Yonesawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture.* Springler Verlag, 1991.

[Stroustrup 1994] B. Stroustrup. *The Design and Evolution of C++.* Addison–Wesley, 1994.

[Watanabe & Yonezawa 1988] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 23, September 1988.

[Yokote & Tokoro 1986] Y. Yokote and M. Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 21, pages 331–340, 1986.

[Yokote & Tokoro 1987] Y. Yokote and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

[Yonezawa *et al.* 1987] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming.* The MIT Press, 1987.