SUBMICRON SYSTEMS ARCHITECTURE PROJECT
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Introduction to C^{+-}

Jakov N. Seizovic

Caltech Computer Science Technical Report

Caltech-CS-TR-93-32

22 August 1993
(Subject to Revision)

The research described in this report was sponsored by
the Advanced Research Projects Agency of the Department of Defense,
and monitored by the Office of Naval Research.

Contents

Chapter 1

Introduction

This technical report is a draft version of the C+- programming manual. It consists of excerpts from Chapter 1, and the entire Chapters 2 and 3 of the author's Ph.D. thesis [33]. This report is to be superseded by a version that includes the C+- User's Manual.

1.1 Concurrent Programming

There are two, typically conflicting, driving forces shaping the developments in concurrent programming: increasing efficiency and increasing expressivity.

The efficiency-conscious programming systems are typically the products of design teams also involved with the design of concurrent machines, and often reflect the underlying architecture. Shared-memory programming and explicit-message-passing programming are representatives of this class.

The expressivity-conscious programming systems are often produced by the frustrated users of the products of the former groups, and are typically architecture-independent (Section 1.1.3).

1.1.1 Shared-Memory Programming

The first developments in concurrent programming were motivated by the advent of multiprogramming and multiuser operating systems. It should not, therefore, be surprising that the first concurrent-programming systems supported concurrent processes that communicated and synchronized through the memory of the machine on which they were executing. The development of the Parallel RAM (PRAM) model, a theoretical framework on which much of the work in concurrent algorithms is based, also promoted the popularity of this programming style, which is still the predominant form of concurrent programming.

From the early stages on, shared-memory programming has been plagued by various incarnations of the mutual-exclusion problem. This problem is due primarily to the discrepancy in access granularity between the data structures and the memory units used to represent these data structures. A number of remedies were introduced: atomic test-and-set and/or fetch-and-add instructions [18], and semaphores [13]. One of the most significant efforts was the work of Per Brinch Hansen on Concurrent Pascal, and the development of *monitors* [19]. Monitors encapsulate data with the (mutually-exclusive) operations defined on the data in programmer-defined, compiler-and-runtime-system-managed units. This work forms a foundation on which many of the recent developments in object-oriented concurrent programming are based, including the programming system described in this thesis.

1.1.2 Explicit Message Passing

Communication and synchronization through explicit message passing is a programming paradigm whose roots are as old as computers themselves, stemming from the need for inter-computer information exchange. This programming paradigm was adopted and adapted for programming multicomputers [22, 29]. Starting with the Cosmic Cube [28] and its commercial descendents [30, 26], the mainstream representatives of the multicomputer architecture employ off-the-shelf processor, memory, and compiler technology. Programming systems for these machines are based on a variety of sequential programming languages for specifying individual process behavior, wherein communication and synchronization between processes is achieved through a set of library routines.

There are two problems that are the curse of this programming style. First, although modular organization of data structures can be achieved within a process, this modularity does not extend readily to collections of processes. Second, the off-the-shelf technology often brought the off-the-shelf notion of process granularity; heavy, UNIX-style processes impose an unacceptably high software overhead to process communication and synchronization.

1.1.3 Architecture-Independent Programming

A number of programming models and notations have been devised to provide a uniform view to the programmer of concurrent computers, and to map computations onto either of the architectures described above. The advantages that these programming systems offer in reducing programming effort are remarkable; preserving the cost-effectiveness of concurrent computers running such programs, however, has yet to be demonstrated. The assembly programming of conventional, sequential computers has been all but eliminated by higher-level notations through large improvements in program-writing efficiency, with small degradations of program-execution efficiency. The same has yet to happen to tailor-made concurrent-programming notations.

Functional Programming and Dataflow

In its pure form [3], functional programming provides a method for defining functions in terms of other, more-primitive functions. The value of a function is determined only by the value of its arguments, and is not history-sensitive. Since there are no side effects, functional-programming notations are implicitly concurrent, and sub-expressions, including function arguments, can be evaluated independently of each other.

The introduction of side-effects into functional-programming notations enables them to model history-sensitive behavior, but it also opens them up to the full set of problems associated with imperative-programming notations. Extending pure functional programming with single-assignment variables and streams, as introduced by dataflow researchers, represents an important intermediate point. This extension relaxes the no-side-effects requirement into the *monotonicity* requirement: A variable starts up uninitialized, and an assignment bounds it to a value (multiple assignments are disallowed). A stream consists of a (possibly-infinite) sequence of variables that can only be read and appended. Using single-assignment variables for communication and synchronization is also used extensively in compositional programming [9, 7], and in concurrent logic programming, described next.

Concurrent Logic Programming

The programming model typically associated with sequential logic programming is that of proving an existentially quantified statement given a program that consists of a set of axioms [37]. Implementations of this model involve backtracking that could, in principle, be replaced by concurrent examination of all the alternatives. However, for efficiency reasons, and because of the need to better model input/output behavior [39, 34], concurrent logic programming makes a significant departure from this model: There is no backtracking; once a (non-deterministic) choice is made, no alternatives are examined.

A concurrent logic program consists of a set of guarded clauses, and each clause represents a recursive specification of process structures. To program in a concurrent logic programming notation is to specify tasks as unordered, concurrent sets of subtasks. Tasks communicate and synchronize with each other by binding single-assignment variables, and waiting for variables to become bound.

Restrictions on the expressivity of clause guards, to improve efficiency, lead to a family of *flat* concurrent-logic notations [39]. A minimalist approach to concurrent logic programming of Ian Foster and Stephen Taylor resulted in Strand, a streamlined and efficient concurrent-programming system [17], without giving up much of the expressive power.

UNITY

UNITY, developed by K. Mani Chandy and Jayadev Misra [8], is a computation model and a programming notation, with an associated proof methodology. A UNITY program consists of a set of guarded multiple assignments. These assignments are executed in arbitrary order: The focus of programming in UNITY is on *what*, *i.e.,* on data transformations, as opposed to *when*. A particular execution order can be enforced only through data dependencies. A computation terminates when it reaches a fixed point, *i.e.,* when no assignment in the program modifies any variables.

An interesting related research has been reported by Craig S. Steele [36]. In this work, a programming model and a corresponding notation are developed, in which program actions are associated with data objects through a programmer-specified triggering mechanism. An efficient multicomputer implementation of this UNITY-like programming system is demonstrated.

Actors

The Actors model of computation was first proposed by Carl Hewitt and Henry Baker [20, 21], and was later formalized by William D. Clinger [10] and Gul Agha [1]. In this model, the unit of concurrent computation is an *actor*, an independent computing agent that is activated in response to messages sent to it. Each actor has a unique address, an associated message queue, and a specified behavior. In a response to a message, an actor can: *send* messages, create *new* actors, and *become* a new actor by specifying its replacement behavior.

Because of its simplicity, potential efficiency, and straight-forward implementation on distributed architectures, the Actors model is the basis for numerous concurrent-programming systems. The reactive-process programming model, described next, and its associated notation, described in Chapter 2, are based in part on the Actors model of computation.

1.2 The Reactive-Process Programming Model

The reactive-process programming model is a variant of the Actors programming model. Computation in this model is performed by a set of *processes*, independent computing agents. A process is normally at rest, and starts executing in response to a *message* (including the initial, creation message). In the course of its execution, a process can send messages, create new processes, and modify its state, including self-termination. Message order is preserved for each pair of processes in direct communication. Each message is marked with a *tag* that specifies which of the process's compile-time-fixed set of *entry points* should be invoked. Each entry point runs to completion, and is therefore an atomic update of its process's state.

A process can affect the order of execution of its entry points by enabling and disabling them selectively, at run time; all entry points are initially enabled. A message tagged for a disabled entry point is delivered after that entry point is active again.

This model is extended to include the *remote procedure call* (RPC). An entry point of a process can be specified to return a value to the message sender. When a message is sent and tagged for such an entry point, the sender is suspended until the message with the returned value arrives.

Background

The reactive-process programming model is a result of the work in our research group over the last decade. Interestingly, a comparison with the early work of C. R. Lang on a concurrent version of Simula [25] reveals that our group's ideas seem to have come almost full circle. The ideas of C. R. Lang, and the preceeding work of Per Brinch Hansen, were far-sighted and out-of-sync with the multicomputer technology of their time. In retrospect, it is as if much of what our research group has been doing was tracking and driving the necessary communication, processor, memory, and compiler technology to approach this target.

Starting with the development of the Cosmic Cube, our group embraced the explicit message-passing programming style. The design of an experimental fine-grain multicomputer, Mosaic C, and the similarity of our approach to the Actor model of computation, provided additional motivation; this effort culminated with the work of W. J. Dally on Concurrent Smalltalk [12]; of W. C. Athas and N. J. Boden on Cantor, a minimalist Actor-based notation [2, 4]; and of W.-K. Su on Reactive-C and distributed event-driven simulation [38]. The work on the Cosmic Environment [38] and the Reactive Kernel [32] shifted our focus from organizing computations around processes to organizing computations around messages, and the reactivity became an essential part of the programming model.

Chapter 2

C^{+-}

2.1 Introduction

2.1.1 Object-Oriented Programming *vs.* Concurrency

Programming notations that support object-oriented programming techniques are the notations of choice for a rapidly growing number of complex applications. Indeed, not since the introduction of structured programming [11] has there been such a degree of unanimity in the programming community. This unanimity is even more remarkable considering that, just as was the case with structured programming [14], the power of object-oriented techniques is difficult to convey to readers through short, example programs in books or articles. When observed in isolation, none of these techniques is new or revolutionary. It is only when one approaches a large-scale programming task armed with the full set of techniques that their power becomes evident.

Structured-programming techniques advocate structuring of program control flow in a top-down, compositional fashion. Object-oriented programming techniques promote data organization in a bottom-up, standard-parts fashion. Both paradigms emphasize modularity, but, whereas the former is focusing principally on modularity of control structures, the latter does a better job of encapsulating data structures with the operations defined on these structures.

Object-oriented programming came about through attempts to make large, sequential programs more manageable. Techniques such as data encapsulation and access protection, inheritance, and guaranteed initialization, all emerge from the goal of helping programmers help themselves.

By our view, much of what the techniques of object-oriented programming are really helping to manage is *concurrency.* Events are concurrent if they are unordered, *i.e.,* if they can occur in any order, or in parallel. Mutual exclusion is an example of an issue most often associated with concurrent programming, but the problems that result from a disregard for mutual exclusion also occur regularly in large sequential programs. With uncontrolled access to global variables, it is impossible to keep track of all of the places in the code where a certain variable is accessed, and of all the invocations of such code. Non-deterministic execution is another issue most often associated with concurrent programming. For a fixed set of inputs, the execution of a sequential program will always result in the same ordering of state changes, yet, with side effects on global variables, it is often far from obvious what all the inputs to a program are.

Whereas sequential programming brings out the worst in us only in the large, concurrent programming will do that already in the small. It should not be surprising, then, that in the hope of reaping some of the benefits that object-oriented techniques brought to sequential programming, we are witnessing a proliferation of programming systems trying to amend a particular object-oriented notation with concurrent semantics.

2.1.2 Concurrent Object-Oriented Languages

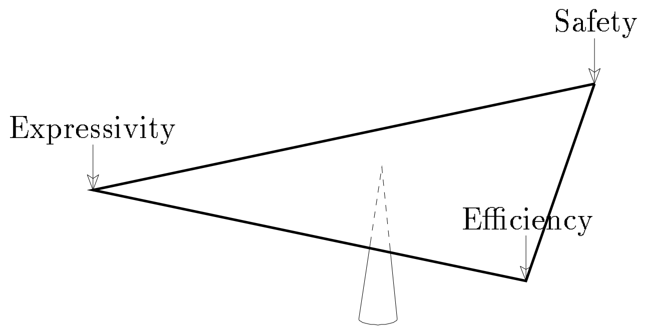

Figure 2.1: Design tradeoffs for concurrent programming systems

The three-way design tradeoffs illustrated in Figure 2.1 are typical of design of any programming system, not only those attempting to harness concurrency. However, all three requirements are more pronounced, and the balance more difficult to achieve, for a concurrent-programming system:

- *Efficiency* — One of the major reasons to employ concurrent solutions in the first place is to get more performance, and programming-system overheads are less likely to be tolerated by users.

- *Expressivity* — Moving from a single to many threads of control in itself places additional demands on expressivity, and also due to the requirement that threads communicate and synchronize their activities.

- *Safety* — In addition to mutual exclusion and possible non-determinism mentioned in the previous section, issues such as deadlock and livelock have to be dealt with. Simple semantics that aid correctness proofs are essential.

It is likely that some readers will find what we consider a balanced design to be biased in favor of efficiency, then expressivity, and then safety. Our argument about the increased importance of efficiency in a concurrent-programming environment is sometimes disputed on grounds that, because concurrent systems offer better performance/cost than their sequential counterparts, one can afford more inefficiencies at the operating/runtime system level. The consequence of this view on concurrent architectures is that machines with pathetic process-creation and communication overheads are being designed and built. The major goals of the work described in this thesis are to show that this pitfall can be avoided, and to demonstrate that fine-grain concurrency can be efficiently exploited.

Extensions of C++

C++ is an object-oriented notation that is in widespread use due to its efficiency, availability, and upward compatibility with C. C++ is the starting point for numerous programming systems that attempt to amend C++ with concurrent semantics, including the system described in this thesis.

C+-

C+- is the result of an experiment to express reactive-process, concurrent programs (Section 1.2) in an object-oriented programming notation. Although C+- is an extension[1] of C++, the objective of the C+- project has *not* been to be able to execute arbitrary C++ programs efficiently on the Mosaic. The emphasis of C+- is on providing efficient support for the simple abstractions fundamental to the reactive-process computational model: process creation and communication. C+- strives not to impose higher-level policies on synchronization, communication protocols, or process placement.

Although the C+- programming system is portable across a wide range of architectures, the Mosaic has been both the driving force and the reality test behind this effort. Design decisions have consistently been made to avoid compromising the performance of C+- programs on the Mosaic. Higher-level programming systems may be layered on top of C+-, but C+- is intended to serve as the Mosaic's lowest-level, *workhorse* programming system, suitable both for operating-system and application programming.

The remaining sections of this chapter are devoted to teaching the reader about C+-. Familiarity with the basic concepts of object-oriented programming and of C++ in particular is assumed: classes, inheritance, access rules, operator overloading. Keywords are underlined in programming examples. Although an effort has been made to steer clear of the idiosyncrasies of C++, some of them

[1]C+- is not a superset of C++ because it imposes restrictions on global variables, as discussed in Section 2.3.

were essential, and they are explained as they are encountered. The reader is cautioned, however, that C+- is by no measure a minimalist, toy-example-writing notation; some of the more advanced examples are likely to present difficulties to those not familiar with C++. Our hope is that this difficulty is the result of C+-'s completeness, rather than of poor design choices.

2.2 The Process Concept

The C++ *object* concept is carried over intact to C+-: `class` is a user-defined type; an object created according to a class definition is a collection of data items, a set of operations defined on them, and a set of access rules (Program 1). Class member functions have the usual, sequential semantics.

```
class   C
{
private:
        int     data;
public:
                C()             { data = 0; }           // initialization
        void    write(int i)    { data = i; }           // update
        int     read()          { return(data); }       // retrieve
};
```

Program 1: A Class Definition

The *process* concept is the only extension that C+- introduces to C++. The `processdef` keyword parallels the `class` keyword syntactically (Program 2). Access rules are associated with data members and functions of a process definition, and process definitions can be derived from other process definitions (Section 2.4.1).

```
processdef      P
{
private:
        int     data;
public:
atomic          P()             { data = 0; }           // initialization
atomic  void    write(int i)    { data = i; }           // update
atomic  int     read()          { return(data); }       // retrieve
};
```

Program 2: A Process Definition

However, a process created according to a process definition is more than a collection of data items:

Specification 1 A process is an independent computing agent, and a unit of potential concurrency. Its public interface consists of a set of atomic actions. At creation time, the process *constructor*[2] is executed if it is defined. After the constructor completes, the process is at rest. The invocation of an atomic action of a C+− process is decoupled from its execution. Conceptually, there is an infinite queue of incoming requests for each process; the invocation of an atomic action places a request into this queue. Process execution consists of servicing these requests, with each atomic action running to completion.

Creating a process is no different from creating an object (Program 3). In most cases, processes are created dynamically (`pp = new P;`), and persist until they are explicitly destroyed (`delete pp;`). One can also create a temporary process as a local variable, just as with any other type (`P p;`). This temporary process is destroyed implicitly when execution leaves its scope.

```
{
    int  i;             // declaring an integer
    P*  pp;             // declaring a process pointer

    pp = new P;         // creating a persistent process
    i = pp->read();     // retrieving a value
    pp->write(i+1);     // updating
    delete pp;          // explicitly destroying the persistent process

    {
        P  p;           // declaring a temporary process

        i = p.read();   // retrieving a value
        p.write(i+1);   // updating
    }                   // implicitly destroying the temporary process
}
```

Program 3: Programming with Processes

A C+− computation is initiated by a runtime system that, concurrently with initialization of global processes, creates an instance of `root` (Program 4), the constructor of which is defined by the user.

Specification 2 A process can affect the order of execution of its atomic actions by enabling and disabling them selectively, at run time. All atomic actions are initially enabled; execution of a disabled action is postponed until the action is enabled again.

[2] A process constructor is an atomic action with the same name as that of the process definition. The constructor may not return any value.

```
processdef      root
{
public:
atomic          root(int argc, char** argv);
};
```

Program 4: The `root` process

For example, let us assume that the rules for accessing a process of type `P` in Program 2 are such that it may be updated only once; every subsequent `write` request should be tagged as an error. Furthermore, all `read` requests occurring before the first `write` should be serviced only after the first update occurs. The process definition for this version of `P` is listed in Program 5.

Processes communicate and synchronize with each other through atomic actions. Thus far, we have discussed only the behavior of processes as servers — how they deal with incoming requests (invocations of their atomic actions). We shall now define the behavior of processes as clients — how they request services from other processes:

Specification 3 When invoking an atomic action that does not return a value (returns a `void`), or if the returned value is not used, the caller continues execution independently of the callee. The order of invocations is preserved for each pair of processes in direct communication. If the value returned by an atomic action is used, the caller may be suspended until the returned value is available.

Invoking an atomic action that returns a value does not, in itself, imply that the requesting process will be suspended until the requested value is available. It is only when this value is *used* that a thread of activity must be suspended. For example, the Program 6 uses a divide-and-conquer approach to compute the n^{th} Fibonacci number. Both sub-computations are initiated, and the process will suspend only if it attempts to add the two partial results before they are available.

It is sometimes desirable to enforce the sequential order of execution of sub-computations. In such cases, the C+- `await` construct should be used. For example, `return (await(f1.compute(n-1)) + f2.compute(n-2));` ensures that the first subcomputation is complete before the second one is initiated.

Programming systems differ considerably in what constitutes use of unresolved variables, also called *futures*. The most aggressive systems allow futures to be exchanged between processes, and suspend a thread only when a value is needed for a hardware-implemented expression evaluation. Support for futures is the central issue for numerous concurrent-programming systems [23, 35, 40]. C+- is not one of these systems, and is not very aggressive in trying to discover and utilize this type of concurrency. *In C+-, assigning an unresolved value to any programmer-defined variable constitutes use of that future, and will cause the*

```
processdef      P
{
private:
        int     initialized;
        int     data;
public:
atomic          P();
atomic  void    write(int);
atomic  int     read();
};

atomic          P::P()
{
    initialized = 0;
    passive read;
}

atomic  void    P::write(int i)
{
    if ( initialized )
    {
        report_error();
    }
    else
    {
        data = i;
        initialized = 1;
        active read;
    }
}

atomic  int     P::read()
{
    return(data);
}
```

Program 5: Enabling and Disabling Atomic Actions

thread to be suspended. C+- guarantees only that a thread will not be suspended unnecessarily within an expression evaluation. C+- semantics allow any additional compiler/runtime system optimization, but only within the body of a function or an atomic action. Unresolved variables must be resolved before they can be passed as arguments.

The reason for C+-'s non-aggressive utilization of futures is that we want to encourage a programming style in which the concurrent behavior is generated explicitly, as opposed to trying to utilize the concurrency that is implicit in sequential formulation. Synchronization on an unresolved future is inherently more

```
processdef      fib
{
public:
atomic  int     compute (int n)
                {
                    switch (n)
                    {
                        case    0:      return  0;
                        case    1:      return  1;
                        default:        fib  f1, f2;
                                        return  (f1.compute(n-1) + f2.compute(n-2));
                    }
                }
};
```

Program 6: Divide And Conquer

expensive than, for example, synchronization using the active/passive semantics, because the process state that must be saved when blocking on a future is much larger. For notations that have stack-based implementations of the regular function-call abstraction, such as C+-, this state includes the stack.

2.3 Managing Concurrency

All concurrency-related issues in the C+- programming system are encapsulated into the process concept. The following syntactic restrictions enforce this requirement:

- Only atomic actions can be `public` members of a process definition.[3]
- Only values, process pointers, and process references[4] can be arguments to atomic actions.
- Processes are the only global[5] variables allowed.
- Process definitions can have no `friends`.[6]

As specified in Section 2.2, a process is a unit of potential concurrency. Processes communicate and synchronize with each other through atomic actions.

[3]The C++ `static` member functions can be public members of a process definition, since their semantics do not allow them to access process members anyway.

[4]The difference between pointers and references is a subtle idiosyncrasy of C++, and, for the purposes of this thesis, the two can be considered equivalent.

[5]This includes both global and `static` C++ variables, *i.e.,* all variables with file scope.

[6]The `friend` construct in C++ allows non-member functions to have full access to `private` class members.

The remainder of this section will be devoted to examples illustrating how some of the well-known concurrent-programming paradigms can be implemented in terms of C+- processes.

2.3.1 Remote Procedure Call

The remote procedure call (RPC) is a common form of interaction between threads of activity. As illustrated in Program 7 and in Figure 2.2, a client requests a service from a server and suspends its execution until the request has been attended to. The semantics of the RPC are identical to those of an ordinary procedure call. The implementations of the two types of procedure calls, however, are typically different, because the client and the server may be operating in different address spaces. A better name for the RPC might be "interprocess procedure call."

```
processdef      server
{
public:
atomic  int     request (int);
};


processdef      client
{
public:
atomic          client (server* s)
                {
                    int  i = s->request(123);
                }
};
```

Program 7: Remote Procedure Call

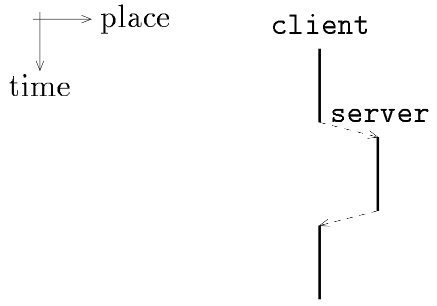

Figure 2.2: Remote Procedure Call

During a remote procedure call, the calling process is nominally suspended until the returned value is available, so no concurrency is introduced. However, as discussed in Section 2.2, with the use of futures, the semantics of the RPC can be extended so that several requests can be issued concurrently, and the calling process is suspended until all the requests have been serviced (Program 6 and Figure 2.3).

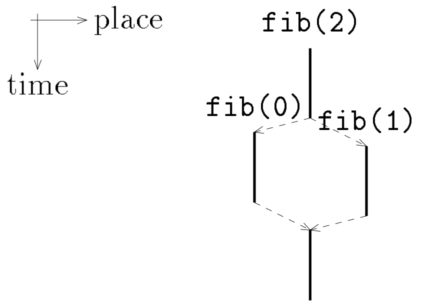

Figure 2.3: Divide And Conquer

2.3.2 Call Forwarding

Call forwarding is a paradigm associated with message-based object-oriented programming systems, and is similar to tail recursion. As an example, consider the sequential search of a singly-linked list of dictionary processes in Program 8.

```
processdef      dict
{
private:
        dict*   next;
        int     index;
        int     data;
public:
atomic  int     find (int i)
                {
                    if ( i == index )
                        return  data;
                    else
                        return  next->find(i);    // can be replaced by:
                //      forward  next->find(i);
                }
};
```

Program 8: A Sequential Search

When the value returned from an atomic action is itself obtained by an atomic action invocation, programmer may choose to use the `forward` statement instead. With the `return` statement, a request is issued, the process is suspended until the value is available, and then reply is sent to the calling process. The effect of call forwarding is to defer servicing of the request to another process. Two sequential search examples, one using the `return`, and another the `forward` statement, are illustrated in Figures 2.4 (a) and (b), respectively. In addition to reducing

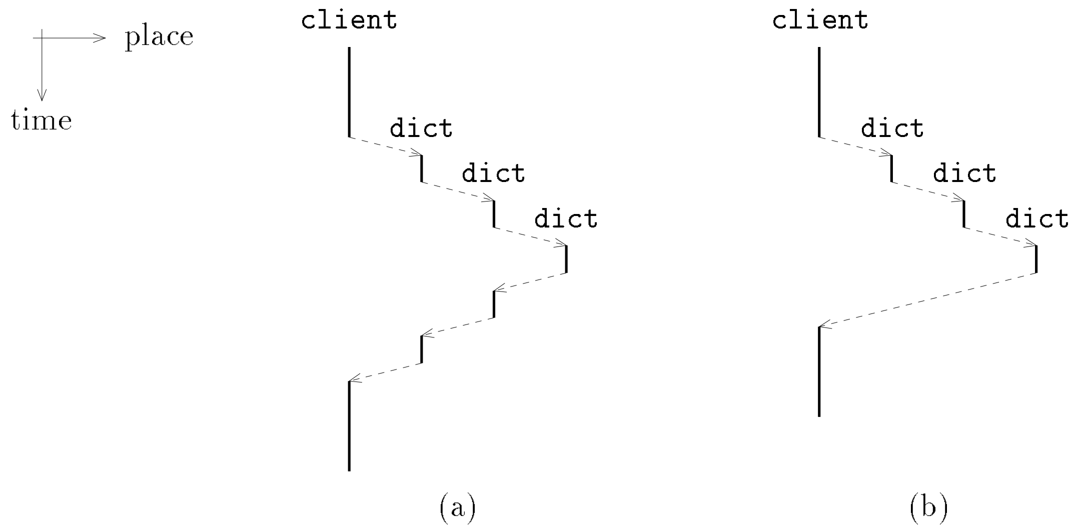

Figure 2.4: A Sequential Search with RPC (a), and with Call Forwarding (b)

the number of replies, call forwarding enables the list of processes that form a dictionary to process multiple requests in a pipeline fashion. At any point in time, each search request is being worked on by at most one dictionary process.

2.3.3 Fork-Join

The remote-procedure-call mechanism with limited support for futures, as provided by C+−, offers a convenient and easy-to-understand programming paradigm for an important class of problems. A more flexible, fork-join mechanism for process synchronization in C+− is offered through the combination of non-suspending, atomic-action invocation and active/passive semantics.

There are two paradigms that C+− programmers can use to generate concurrent activities:

- *Creating new processes,* whether persistent or temporary. The parent process continues execution independently[7] of the child.

- Upon *invoking an atomic action that does not return a value,* or when the returned value is not used, the caller continues executing without waiting for the callee.

[7]When a pointer to a newly created process is used in a subsequent computation, this may or may not require suspending the parent, depending on the implementation. However, the parent continues execution concurrently with child's constructor.

The synchronization barriers can be expressed using active/passive semantics. Suppose that an FFT computation is implemented as illustrated in Figure 2.5 [27]. The expressions along the edges of the graph are coefficients. Multiple inputs to a

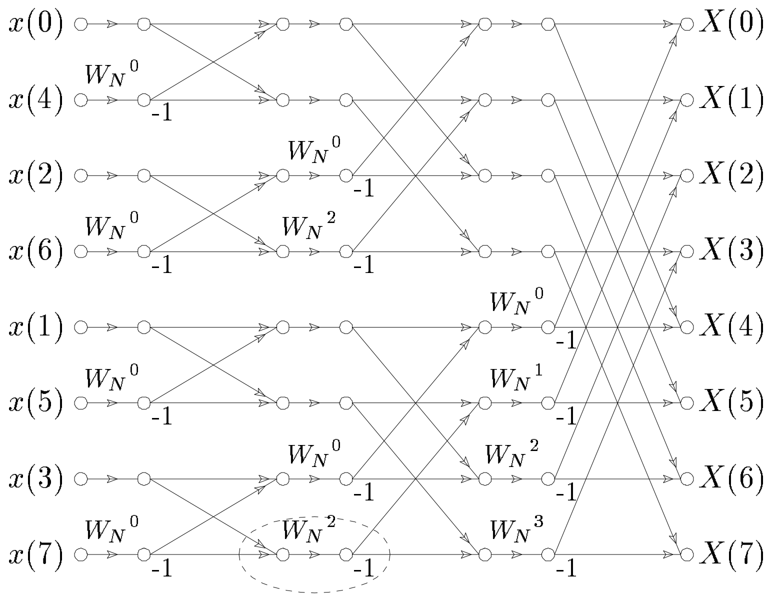

Figure 2.5: An 8-Point FFT Computation. ($W_N = e^{-i\frac{2\pi}{N}}, N = 8$)

node imply addition, and multiple outputs imply replication of the result.

A concurrent program for N-point FFT computation could employ N processes, and compute the result in $\mathcal{O}(\log N)$ steps. Each step would consist of: getting two requests along the input edges; adding the two input values; multiplying by the coefficient; and producing two output values.

A version of this program could similarly employ $N \log N$ processes in a pipeline regime, achieving the same $\mathcal{O}(\log N)$ latency, but a new result would be computed on every step.

In either approach, though, a process (circled in Figure 2.5) must get one data item along each of its input edges to be able to compute and emit one data item along each of its output edges. A process that might be used as part of the FFT-computation pipeline is listed in Program 9.

```
processdef      fft
{
private:
        Complex W, first;
        fft     *out_up, *out_dn;
        void    output(Complex in)
                {
                    Complex  result = (first + in) * W;
                    out_up->up(result);
                    out_dn->dn(-result);
                }
public:
atomic          fft(fft* u, fft* d, Complex r)
                {
                    W = r;
                    out_up = u;
                    out_dn = d;
                }
atomic  void    up(Complex in)
                {
                    if ( passive(dn) )          // upon receiving both requests
                    {                           // produce the output
                        active dn;
                        output(in);
                    }
                    else                        // if you only have one request
                    {                           // await the second one
                        passive up;
                        first = in;
                    }
                }
atomic  void    dn(Complex in)
                {
                    if ( passive(up) )          // upon receiving both requests
                    {                           // produce the output
                        active up;
                        output(in);
                    }
                    else                        // if you only have one request
                    {                           // await the second one
                        passive dn;
                        first = in;
                    }
                }
};
```

Program 9: An FFT-Computing Process

2.3.4 Semaphores

First introduced by E. W. Dijkstra [13], semaphores are low-level primitives for process synchronization. A semaphore is typically used to control access to a shared data structure, with an N-ary semaphore allowing access to at most $N-1$ processes at any point in time. Two operations are defined on semaphores: *acquire* and *release*. In general, an implementation of an N-ary semaphore must guarantee that the number of acquire operations minus the number of release operations is at most $N-1$, and at least 0. A C+- implementation of an N-ary semaphore is presented in Program 10.

```
processdef      semaphore
{
private:
        int     count;                      // number or processes inside
                                            //  the critical section
        int     max;                        // the maximum number allowed
public:
atomic          semaphore(int N)            // initially, there is no
                {                           //  processes inside the critical
                    max = N - 1;            /   section
                    count = 0;
                    passive release;
                }
atomic  int     acquire()
                {
                    count++;                // one more inside
                    active release;         // at least one can release
                    if ( count == max )     // if the maximum is reached,
                        passive acquire;    //  no one can get in
                    return 1;
                }
atomic  int     release()
                {
                    count--;                // one less inside
                    active acquire;         // at least one can acquire
                    if ( count == 0 )       // no one is in, so
                        passive release;    //  no one can exit
                    return 1;
                }
};
```

Program 10: N-ary Semaphore

An often-used special case for $N=2$, the binary semaphore, is illustrated in Program 11.

```
processdef      semaphore
{
public:
atomic          semaphore()
                {
                    passive release;
                }
atomic  int     acquire()
                {
                    active release;
                    passive acquire;
                    return 1;
                }
atomic  int     release()
                {
                    active acquire;
                    passive release;
                    return 1;
                }
};
```

Program 11: Binary Semaphore

2.3.5 Monitors

Of all of the concurrent-programming paradigms, semantics of C+− processes are closest to those of monitors [19]. Just as with monitors, C+− processes encapsulate a set of data items and offer mutually exclusive access to a set of routines operating on this data. C+− processes also share some of the problems associated with monitors, as both are non-reentrant. The invocation of an atomic action of a C+− process is, unlike an invocation of a monitor function, decoupled from its execution: conceptually, there is an infinite buffer of incoming requests for each process. This decoupling enables processes to be active computing agents, able to affect the order of execution of their atomic actions.

2.3.6 Recursion

In the examples shown so far, the requirement that all the `public` member functions of a process be atomic actions has been helpful in expressing interactions between concurrent threads of activity. From the point of view of C+− programmers, the most significant repercussion of the atomicity of interprocess activities is that, since at most one execution thread can be associated with a process, atomic actions that return values are not reentrant. For example, in Program 12, the `private` member function `fac` has ordinary, sequential, reentrant semantics. However, the `public` member function `FAC` must be an atomic action. An invocation of `FAC` will, therefore, result in deadlock.

```
processdef      bad
{
private:
        int     fac(int n)
                {
                    if ( n == 0 )
                        return  1;
                    else
                        return  n * fac(n-1);   // OK: functions are reentrant
                }
public:
atomic  int     FAC(int n)
                {
                    if ( n == 0 )
                        return  1;
                    else
                        return  n * FAC(n-1);   // ERROR: atomic actions are
                }                               // not reentrant
atomic  int     Fac(int n)
                {
                    return  fac(n);             // OK: atomic-action interface
                }                               // to a function
};
```

Program 12: Recursive Functions and Non-Recursive Atomic Actions

In the world of non-reentrant atomic actions, processes are the medium used to express recursive behavior (Program 13).

2.3.7 Message Passing

Invoking an atomic action of a process is equivalent to wrapping up the argument list and sending it in a message. According to Specification 3, the atomic-action invocation does not imply blocking (waiting for the reply does), so it is equivalent to a non-blocking message send.

Message receiving has two forms:

- *explicit*, associated with the behavior of processes as clients, which receive a value that is returned from a call to an atomic action; and
- *implicit*, associated with the behavior of processes as servers, which receive an argument list as part of a request to execute an atomic action.

The two forms of receive, explicit and implicit, cover the two extremes of the spectrum of possible mechanisms for message discretion: explicit receive

```
processdef      fac
{
private:
        int     output;
public:
atomic          fac(int input)
                {
                    if ( input == 0 )
                        output = 1;
                    else
                    {
                        fac  child(input-1);
                        output = input * child.result();
                    }
                }
atomic  int     result()
                {
                    return  output;
                }
};

// or

processdef      fac
{
private:
        int     input;
        fac*    parent;
public:
atomic          fac(int i, fac* p)
                {
                    if ( i == 0 )
                    {
                        p->result(1);
                        delete this;
                    }
                    else
                    {
                        input  = i;
                        parent = p;
                        new fac(i-1,this);
                    }
                }
atomic  void    result(int r)
                {
                    parent->result(input*r);
                    delete this;
                }
};
```

Program 13: Recursive Processes

accepts only a particular message from a particular process; implicit receive accepts any message from any process. The `active/passive` semantics provide a more general selective-receive mechanism: atomic actions of a process represent incoming communication channels, and the process can, at run time, select the communication channels over which it is ready to accept a message.

2.3.8 Single-Assignment Variables

Single-assignment variables are a safe form of futures (Section 2.2). Requesting a read access on an uninitialized, single-assignment variable causes the requesting process to be suspended until the variable is assigned to. Since there can be at most one assignment to a single-assignment variable, these variables can be effectively cached. Processes of type `P` in Program 5 are an example of a possible C+- implementation of single-assignment variables.

2.3.9 Process Aggregates

Thus far, we have described processes as independent entities, and have emphasized the code-execution aspects of processes. In this section, we shall show how processes can be treated as instances of a restricted data form, one that can be accessed only through a set of mutually exclusive, atomic actions.

As illustrated in Program 14, C+- programmers can treat processes as variables of any other type. Whether a process is a local variable, member of an object or of another process, element of an array, or used in any other way in which a variable can be used in C++, the process semantics are the same. According to the syntactic restrictions described in Section 2.3, the only operations allowed on a process are to take its address and to access its public members (all of which are atomic actions).[8] The various process usages determine only when a process is created and when it is destroyed. For non-process data types, variable usage also implies what the memory layout is. When accessing processes, one cannot assume, for example, that a process declared as a local variable resides on the stack; nor can one assume that a process that is a member of a class is placed in memory next to the other data members. In Section 3.1.1, we shall discuss how programmers can affect process-placement strategy.

The semantics of C+- are defined such that efficient implementations exist for both mainstream variants of MIMD computers: multiprocessors, which have one global address space, and multicomputers, which have multiple local address spaces. In C+-, regardless of the underlying architecture, a pointer to a process

[8]Process assignment is an atomic action invocation, equivalent to issuing a request to the source process to send a copy of itself to the destination process (Section 3.1.5). Passing processes as arguments is a form of assignment.

```
processdef      P
{
    // ...
};


class   C               // an object of class C contains:
{
public:
        P       p;      // a process
        P*      pp;     // and a process pointer
};


{
    P  p1, p2;          // declare two processes

    p1 = p2;            // process assignment

    P  p[10];           // declare a process array
}
```

Program 14: Treating Processes As Data

can be dereferenced globally, since it contains sufficient information to uniquely identify the process it points to.

An important advantage that multiprocessors have over multicomputers is that they can employ most of the data-layout strategies developed for sequential computers. There are additional performance considerations guiding the design decisions on the data layout, as discussed in [24]. If, for the time being, we neglect such performance considerations, a vector of C+- processes could, on a multiprocessor, be laid out in memory in the same way as a vector of elements of any simple data type. Elements with successive indices would reside at memory addresses that differ by a stride equal to the size of the process. This approach would allow the programmer to compute the address of any process in the vector given the address of any other process in the same vector, and the two corresponding indices.

On a multicomputer, using the above layout strategy for vectors of processes is unacceptable for two reasons: first, the address space of a multicomputer is contiguous only within each multicomputer node, so the maximum size of a process vector would be limited by the size of node memory; and second, although the computation model allows elements of a process vector to operate concurrently, that concurrency could not be used to a performance advantage, because the elements would all reside on the same node.

This example is but an instance of a more general problem of naming constituent elements of distributed objects [12, 6]. There are two issues that are central to the solution of this problem. The first issue is that there should exist a single name (address) of a distributed object, and a way of addressing constituents given this name. The second issue is that the programmer should be able to compute on references, not just store them at process-creation time and fetch them when they need to be used.

A simple solution that takes only the first issue into the account could employ an address-manager process. The manager's address would represent the address of the distributed process as a whole. All the requests would be directed to this process, and then forwarded to appropriate constituent processes. This solution obviously introduces an access bottleneck, but may be acceptable for element processes that exhibit a large ratio of computation/communication.

We consider this problem to be too important to be left to *ad hoc* approaches, particularly for such often-used paradigms as arrays of processes. Accordingly, C+- offers a runtime-system-supported mechanism for address management that preserves the C++ address-computation semantics.

The example in Program 15 shows that the creation of a process array

```
{
    processdef  P       { };

    P*  p = new P[123];          // is equivalent to:

    {
        P*  p = unique__CPM(123,sizeof(P));
        for (int i=0; i<123; i++)
            new @(p+i) P;
    }
}
```

Program 15: Creating A Vector of Processes

consist of two stages. First, a set of unique references is allocated by invoking the `unique__CPM` function, with arguments specifying how many references are required, and what the stride between the adjacent references should be. This function returns a pointer of the generic process-pointer type, `pointer_t`, analogous to `void*` in C++. Next, the actual process creation is requested, specifying that each new element process be placed in such a manner that it can be located through the given pointer. A description of various flavors of process creation is presented in Section 3.1.1. A set of algorithms that provide efficient support for process placement and lookup is described in [5].

2.3.10 Summary

The programming examples in Section 2.3 illustrate that a small set of mechanisms supported by C+− is sufficient to express a variety of concurrent-programming paradigms. This set consists of: process creation, asynchronous request, synchronous request (remote procedure call), and selective servicing of requests (active/passive mechanism). In Chapter 3, we shall present an implementation framework for this set of mechanisms.

2.4 Managing Program Complexity

In the introductory section of this chapter, we discussed how object-oriented programming techniques came about through efforts to aid programmers in managing program complexity. All of the object-oriented techniques supported by C++ are extended to managing processes in C+−. The interested reader may consult the wealth of available literature on C++, including, but not limited to [16].

In the remainder of this section, for completeness, we shall mention briefly two of those techniques: inheritance and virtual functions. We shall then discuss the techniques that are specific to C+− and concurrent programming: process layering, process libraries, and customizing of the data exchange.

2.4.1 Class Inheritance

Class inheritance is the C++ mechanism that enables user-defined types to be *derived* from more basic types, inheriting data members and functions from the base type, possibly adding new ones and/or overriding old ones. Access rights are associated with each class member. For example, in Program 16, `private` members of the base class `shape` can be accessed only by member functions of `shape`; `protected` members of `shape` can, in addition, be accessed by member functions of any class derived from `shape` (for example, `circle`); and `public` members of `shape` can be accessed by any piece of code anywhere in the program. The `class circle` is derived from `class shape` by adding a data member (`radius`) and a member function (`modify_radius()`), and by overriding the member function `draw()`.

A typical memory layout for the two classes is shown in Figure 2.6. The point to

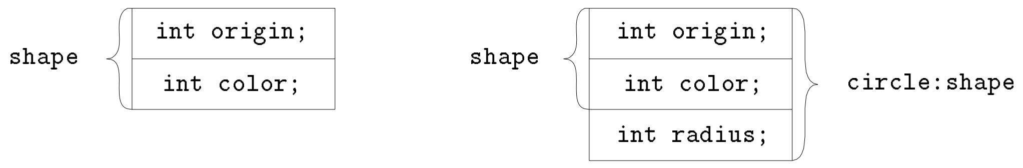

Figure 2.6: Class Inheritance *vs.* Memory Layout

be remembered is that C++ class inheritance is a compile-time rather than a runtime

```
class   shape
{
private:
        int     origin;
        void    modify_origin();
protected:
        int     color;
        void    modify_color();
public:
        void    draw();
};


class   circle : shape
{
private:
        int     radius;
public:
        void    modify_radius();
        void    draw();
};
```

Program 16: Class Inheritance

mechanism.[9] Every instance of `class circle` contains a part corresponding to an instance of `class shape`; it is the definition of `class shape` that is shared, not any particular instance of it.

The C++ class-inheritance mechanism is mimicked by process definitions in C+–; they too can be specified through their similarities with and differences from previously-defined process definitions.

2.4.2 Virtual Functions

The virtual-function mechanism supported by C++ is a mechanism that enables programmers to separate the design of member-function interfaces from the design of member functions themselves.

For example, in Program 16, given a `shape* sp`, and a `circle* cp`, the invocation of `sp->draw()` and `cp->draw()` will result in calling `shape::draw()` and `circle::draw()`, respectively. The compiler decides which call to generate based on the type of pointer through which the function has been called.

Had the two `draw()` functions been `virtual`, the invocation of `sp->draw()` could have invoked either of the two functions, depending on what the pointer

[9]Neglecting, for the time being, such C++ features as multiple inheritance and virtual functions.

sp pointed to. In this case, the compiler generates an indirect call through the class-specific table.

2.4.3 Process Layering

The standard C++ inheritance mechanism allows one to describe process definitions hierarchically. However, once a process is created, it is an independent entity. The hierarchy is reflected in its structure, *not* in its relationship with other processes.

There are important applications where, in addition to *hierarchy in structure*, it is useful to have runtime-exercised *hierarchy in control.* For example, in operating or runtime systems [5], user processes are created and managed by system processes. In simulators [38], processes that model the behavior of physical elements are managed by time- or event-driven schedulers.

The mechanism that C+- uses to support such applications is *process layering*, also called *dynamic process inheritance.* As illustrated in Program 17 and Figure 2.7, every instance of `processdef gate` is managed by an instance of

```
processdef      scheduler
{
private:
        int     time;
};


processdef      gate : dynamic scheduler
{
protected:
        gate*   output;
};


processdef      two_input_gate : gate
{
private:
        int     state;
atomic  void    input1(int);
atomic  void    input2(int);
};
```

Program 17: Process Layering

`processdef scheduler`. The details of process layering will be discussed in Section 3.1, which describes the C+- runtime-system interface. The relationship between the manager process and the managed process is established at the creation time of the managed process. The manager provides a set of services to all processes that it manages, with the same access protection that is offered

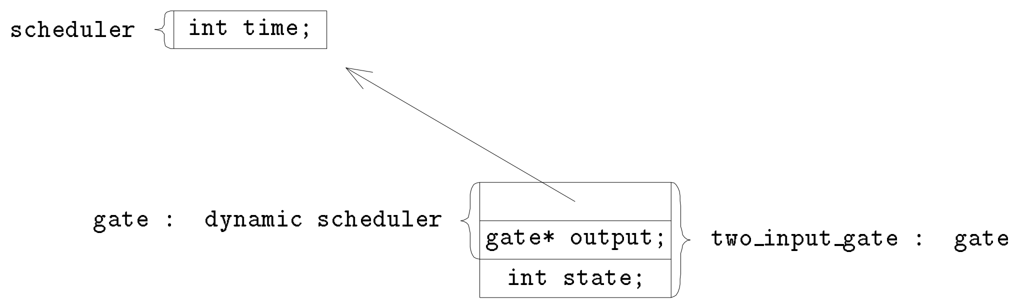

Figure 2.7: Process Layering *vs.* Memory Layout

through the class-inheritance mechanism. The manager decides when an atomic action of any of the processes managed by it is executed (as opposed to invoked), while conforming to the definitions of process behavior as specified in Section 2.2.

2.4.4 Process Libraries

Libraries of C+- processes can be organized in the same way as libraries of data structures in C++. In most cases, the remote procedure calls to atomic actions of processes form a suitable interface, and these calls replace the class member-function interfaces. In these cases, it is sufficient that programs include header files that contain interface-process definitions.

There are cases, however, in which imposing the RPC interface would overly serialize computations that are otherwise concurrent. For example, a process library might initialize a set of processes for FFT computation, as illustrated in Section 2.3.3, employing several input and several output data streams. A stream of input values can be represented by a sequence of non-blocking atomic-action invocations. If a stream of output values were represented as a sequence of replies obtained through the RPC mechanism, just as in the sequential-search example of Section 2.3.2, the computation could not be pipelined. However, unlike in this search example, this problem could not be resolved with call forwarding.

The mechanism typically used for C+- libraries with multiple input and output streams is as follows: an input stream is represented by a sequence of non-blocking atomic-actions invocations of an input-interface process; an output stream is, similarly, a sequence of non-blocking atomic-actions invocations of a process provided by the library user. In this arrangement, the library-user process must be derived from the output-interface process of the library it uses (Section 2.5). When a process uses multiple libraries, multiple inheritance is employed to derive such a process from all of the output-interface processes from which it requires results.

2.4.5 Data Exchange

The designers of C++ made a commendable effort to provide an overloading mechanism that enables programmers to pass arguments by value, even when these arguments are arbitrarily-complicated, linked, data structures. This mechanism is not sufficient for concurrent-programming systems, which must take into account some additional considerations. On multicomputers, object pointers have local meaning. Also, concurrent computers may be heterogeneous ensembles comprised of machines with different data layout, alignment, size, or representation.

C+− addresses all of these potential problems at the inter-process-communication level (invocations of atomic actions) with mechanisms that are described in the remainder of this section. The communication specifications are *declarative*, as opposed to *imperative:* the programmer specifies what special actions should be taken when a data item of certain type is communicated; the compiler guarantees that actions thus specified will be invoked on every occurrence of communication.

Communicating Arbitrarily-Complex Data Structures by Value

One of the premises of fine-grain concurrent programming is that large data structures are implemented in terms of many small, cooperating processes, so it is tempting to claim that process pointers that can be globally dereferenced are all that programmers might possibly want. However, an important use for pointers in C++ is for data structures that are only partially specified at compile time: linked data structures and arrays of variable size. If proper support and clean semantics for this feature were not offered, users would have resorted to *ad hoc* solutions.

The mechanism supported by C+− enables the programmer to specify what *extra* actions should be taken when communicating an object of some class by value. In its most common form, it amounts to flattening the linked data structure before sending, and relinking it upon receiving. As will be illustrated in Section 3.1, variants of this mechanism can also be used to express more intricate (but sometimes much more efficient) communication protocols.

Suppose that the data type of choice is a singly-linked list of elements of type `list`, each of which contains a pointer to the next element in the list, a pointer to a vector of integers, and a field specifying the size of the integer vector. Figure 2.8 illustrates what is required to pass a data item of type `list` by value. Part (a) shows a data item scattered around in memory. Part (b) shows the flattened data structure, with the dashed parts corresponding to other arguments that may be sent in the same communication. If the concurrent computer at hand is a shared-memory multiprocessor, and if the flattened argument list is in the shared address space, the task is completed. Now suppose that passing arguments moves them from one address space to another, as typically happens on a multicomputer. When the message that encapsulates the argument list is received, all the pointers are off by a constant (c), and have to be re-linked, as in (d).

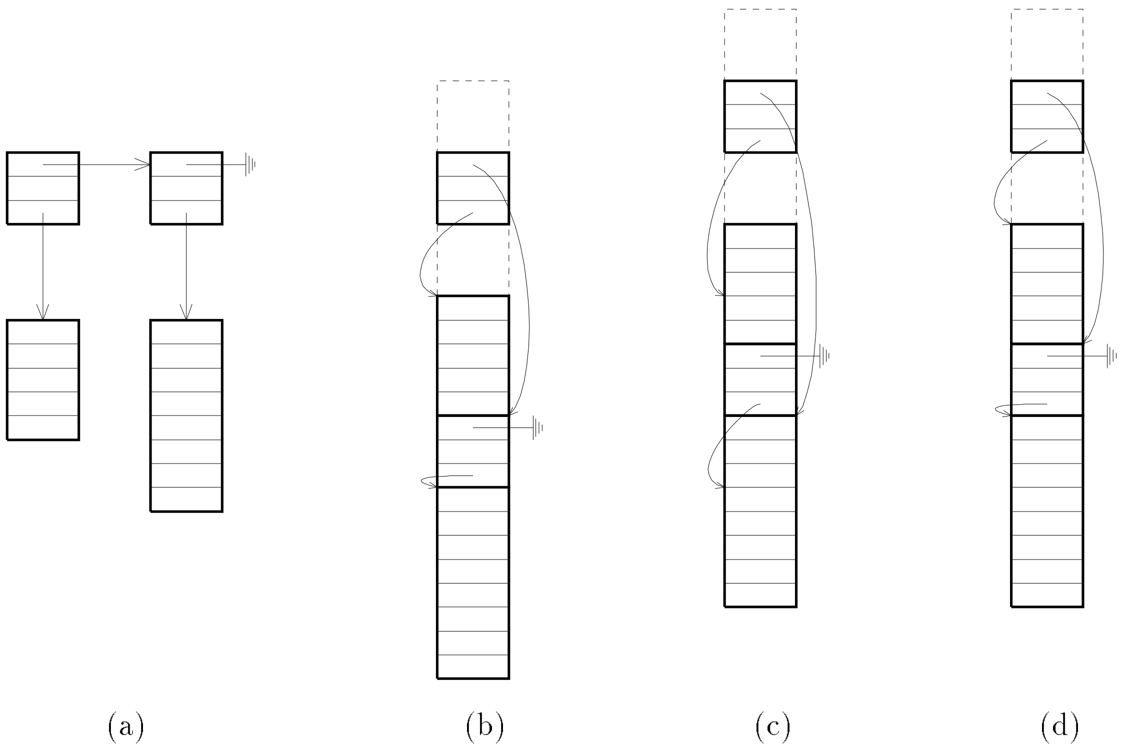

Figure 2.8: Flattening Linked Data Structures

Program 18 is the specification of the flattening and re-linking tasks: The `operator space` computes how much extra space is needed in the argument list when an instance of `list` is passed as an argument to an atomic action. The `operator send` specifies that, in addition to this instance of `list`, a vector of integers and the remaining part of the list should be passed along. The `operator recv` requests that the vector of integers (`data`) and the rest of the list `next` be re-linked in place on the receive side.

This special handling will be invoked not only for instances of `list`, but also for all objects derived from `list`, and for all objects that contain instances of `list` as members. C+- data-structure libraries can, accordingly, be built in a way that allows library users to be indifferent about the details of the implementation.

This example illustrates how arbitrarily complex, linked, data structures can be passed by value. However, to avoid copying, and when sharing of data structures between processes is needed, structures must consist of linked processes, not of linked objects.

```
class   list
{
private:
        int     size;           // number of integers "data" points to
        int*    data;
        list*   next;           // a pointer to the next of kin

public:
        size_t  operator space ()
                {
                    size_t  s = space(data,size);  // space for size integers
                    if (next)  s += space(next);   // space for the rest
                    return  s;                     //  of the list
                }

        void*   operator send (void* v)
                {
                    v = send(v,data,size);         // send size integers
                    if (next)  v = send(v,next);   // send the rest
                    return  v;                     //  of the list
                }

        void    operator recv ()
                {
                    recv(data);                    // re-link int*
                    if (next)  recv(next);         // re-link the rest
                }                                  //  of the list
};
```

Program 18: Passing Linked Data Structures By Value

Communicating Across Heterogeneous Machine Boundaries

The C+- compiler assembles all messages (argument lists to atomic actions), and initiates all instances of communication (invocations of atomic actions). This information enables the compiler to handle the size and alignment of the basic data types (integers, floating-point numbers, *etc.*) for a programmer-specified set of machines that may be involved in direct communication.

The example in Program 19 specifies that, in addition to the local-machine type, communication may be established with machines of types `I286` and `Sparc` (arbitrary, user-specified names). The entries within each machine description correspond to the data size and alignment (measured in units of size equal to the minimum-addressable memory unit on the machine running this program), and any special treatment that may be required for a particular basic data type.[10]

[10]The following is the complete list of C+- basic data types: `char, short, int, long, float, double, long double, signed char, unsigned char, unsigned short, unsigned int, unsigned long, void*, entry_t,` and `pointer_t`.

```
machine I286
{
    char,       1,      1;
    short,      2,      1;
    int,        2,      1;
    long,       4,      1;
};

machine Sparc
{
    char,       1,      1;
    short,      2,      2,      send_lib,       recv_lib;
    int,        4,      4;
    long,       4,      4;
};
```

Program 19: Machine Descriptions

For example, for a machine of type `Sparc`, short integers are of size 2 and have to be positioned on addresses divisible by 2. When sending a short integer to a process residing on a machine of type `Sparc`, the data item has to be converted using the user-supplied and user-named function `send_lib`; when receiving a short integer from such a process, the data item has to be converted using the function `recv_lib`.

The compiler implicitly generates type `machine_t`, defined as:

```
enum  machine_t  { local__CPM, I286, Sparc };
```

and the user is obliged to define the function

```
machine_t  machine__CPM (pointer_t);
```

that maps process pointers into machine types.

2.5 Putting It All Together

The examples of C+- programs shown so far were chosen to illustrate programming techniques. We have deliberately chosen clarity over completeness, and, indeed, some of these examples require the addition of forward declarations to be accepted by the compiler.

In this section, we shall show an example of a complete program that computes the N-point FFT, as illustrated in Figure 2.5. Our concurrent program will closely match this data-dependency graph, with one addition: We shall introduce a column of nodes whose purpose is to rearrange the input values from the standard, linear ordering of indices to the bit-reversed ordering required at the input of the

FFT-computing graph. Figure 2.9 shows the modified graph, with circled parts corresponding to sub-computations performed by individual processes.

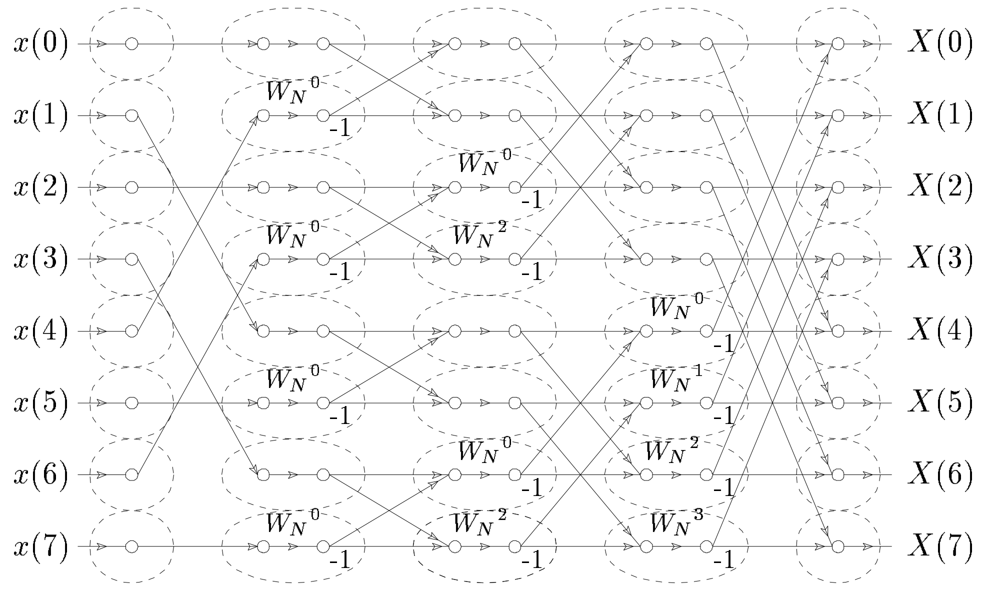

Figure 2.9: An 8-Point FFT Computation, with the Processes Circled.

Typically, writing C+− programs consists of four stages:

- Choosing a concurrent algorithm;
- Designing an input/output interface;
- Designing the process hierarchy; and,
- Describing process behavior.

We shall organize the program as a library package. Figure 2.10 illustrates the user-level view of this library. Input values are to be sent to processes of type `fft`, and output values will be delivered to processes of the same type. For an N-point FFT computation, there are N input and N output processes, all of which have to be derived from `fft`. The set of pointers to N input processes could be represented in a variety of ways, but it is often most intuitive to represent these processes as members of a process vector, as described in Section 2.3.9. The same is true for the set of pointers to N output processes.

Program 20 is the header file that the user must include to access the library. A user program might look like Program 21: Since the library sends the output values to the vector of `fft` processes, the `consumer` processes are derived from `fft`, and have to be created using the distributed-process mechanism. The `producer` processes, on the other hand, don't have to be elements of any vector unless some other part of the user code needs to treat them so.

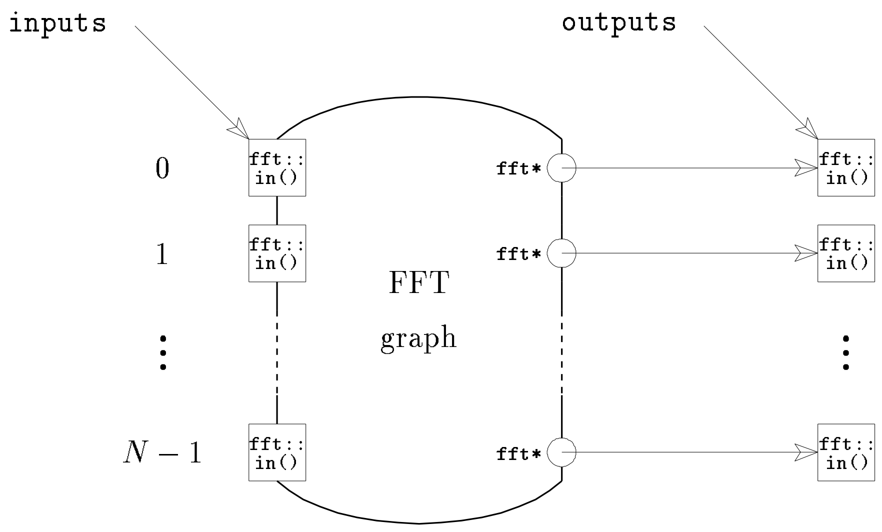

Figure 2.10: User View of the FFT-Library

```
// fft.h


#include       <c+-.h>                    // The runtime-system header file
#include       <Complex.h>                // The complex-arithmetic package


processdef     fft : public CPM           // The runtime system requires that
{                                         // every process be derived from CPM
public:
atomic  virtual void    connect(fft*) = 0;  // The '= 0' syntax in C++ denotes
atomic  virtual void    in(Complex) = 0;    // that this is the specification
};                                        // of an interface, leaving it to
                                          // the derived processes to specify
                                          // how the requests are serviced

processdef     fft_graph : public CPM     // This process represents the
{                                         // whole graph
private:
        fft*    inputs;                   // The pointer to the first input
        int     order;                    // Size of the FFT graph
public:
atomic           fft_graph(int, fft*);    // Creating the fft process graph
atomic          ~fft_graph();             // Deleting the fft process graph
atomic  fft*    input(int);               // Finding out the address of a
                                          // particular input
};


Complex W(int N, int i);                  // A function that computes
                                          // complex roots of 1

int     bit_reverse(int N, int i);        // A bit-reversing function
```

Program 20: The FFT-Library Header File

```
#include        <fft.h>


processdef      consumer : public fft
{
public:
atomic  virtual void    in(Complex);        // Do something with the result
};


processdef      producer : public CPM
{
public:
atomic                  producer(fft*);     // Produce input values
};


const   int  N = 32;


root::root (int argc, char** argv)
{
    fft*  outputs = new consumer[N];        // Create the vector of consumers

    fft_graph*  g = new fft_graph(N,outputs);
                                            // Create the computation graph

    fft*  inputs = g->input(0);             // Get the reference to the inputs

    for (int i=0; i<N; i++)                 // Create N producers
        new producer(inputs+i);
}
```

Program 21: An Example of FFT-Library Usage

Figure 2.11 shows the process-specification hierarchy that we chose to implement, and Programs 22 and 23 specify this hierarchy.

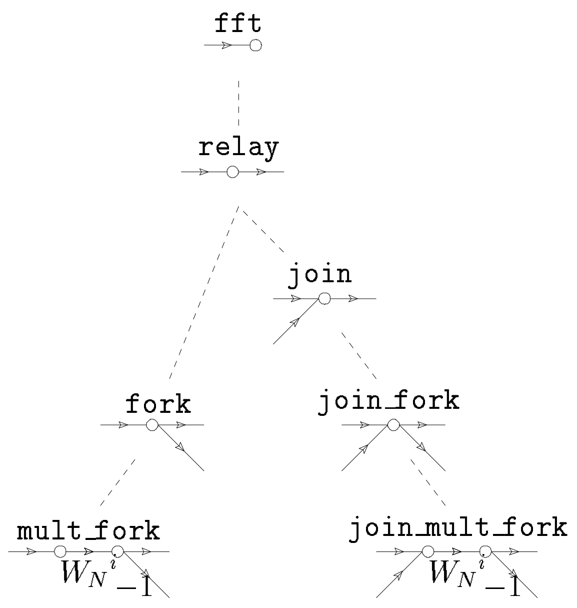

Figure 2.11: Process-Specification Hierarchy

The `fft` process definition is just an interface specification, and does not describe any computation. The remaining process definitions specify that the process activity consists of four distinct stages:

- Establishing a connection, *ie,* obtaining output references;
- Getting one or two input values;
- Computing the result, which may involve an addition and a multiplication; and,
- Outputting one or two output values.

The common parts of the code are shared between different process definitions through the process-inheritance mechanism. Using multiple inheritance (whereby process definitions can be derived from more than one process definition) would have resulted in better code reuse. Nevertheless, we felt that, in the examples in this thesis, multiple inheritance would not have contributed to reader's understanding of C+-.

```
// fft1.h


#include        "fft.h"


processdef      relay : public fft
{
protected:
                fft*    out;                // Output reference
                Complex result;             // The result
        virtual void    compute(Complex);   // How to compute the result
        virtual void    output();           // How to generate the output
public:
atomic  virtual void    in(Complex);
atomic  virtual void    connect(fft*);
atomic                  relay()
                        { passive(in); }
};


processdef      join;


processdef      fork : public relay
{
protected:
                join*   out1;               // Fork adds an output reference,
        virtual void    output();           // and produces two output values
public:
atomic  virtual void    connect(fft*, join*);
};


processdef      mult_fork : public fork     // Mult_fork also needs to multiply
{
protected:
                Complex W;                  // so here is the multiplicand
        virtual void    compute(Complex);   // and how to compute
        virtual void    output();           // It must generate the +- output
public:
atomic  virtual void    connect(fft*, join*, Complex);
};
```

Program 22: Process Hierarchy for FFT Computation, Part 1

```
// fft2.h


#include        "fft1.h"


processdef      join : public relay         // Join has two distinct inputs
{
protected:
        virtual void    compute(Complex);   // How to compute the result
public:
atomic  virtual void    in (Complex);
atomic  virtual void    in1(Complex);
atomic                  join()
                        { passive(in); passive(in1); }
};


processdef      join_fork : public join             // The same modifications
{                                                   // as from relay to fork
protected:
        join*   out1;
        virtual void    output();
public:
atomic  virtual void    connect(fft*, join*);
};


processdef      join_mult_fork : public join_fork   // The same modifications
{                                                   // as from fork to mult_fork
protected:
                Complex W;
        virtual void    compute(Complex);
        virtual void    output();
public:
atomic  virtual void    connect(fft*, join*, Complex);
};
```

Program 23: Process Hierarchy for FFT Computation, Part 2

The behavior of various process types is specified in Programs 24, 25 and 26.

```
// fft0.cpm

#include        "fft2.h"

atomic
void
relay::connect (fft* f)
{
    out = f;
    active;             // make all atomic function active
}

atomic
void
fork::connect (fft* f, join* j)
{
    out  = f;
    out1 = j;
    active;
}

atomic
void
mult_fork::connect (fft* f, join* j, Complex c)
{
    out  = f;
    out1 = j;
    W    = c;
    active;
}

atomic
void
join_fork::connect (fft* f, join* j)
{
    out  = f;
    out1 = j;
    active;
}

atomic
void
join_mult_fork::connect (fft* f, join* j, Complex c)
{
    out  = f;
    out1 = j;
    W    = c;
    active;
}
```

Program 24: The FFT Computation, Part 1

```
// fft1.cpm

#include        "fft2.h"

atomic
void
relay::in (Complex c)
{
    compute(c);
    output();
}

void
relay::compute (Complex c)
{
    result = c;
}

void
mult_fork::compute (Complex c)
{
    result = W * c;
}

void
relay::output ()
{
    out->in(result);
}

void
fork::output ()
{
    out->in(result);  out1->in1(result);
}

void
mult_fork::output ()
{
    out->in(-result);  out1->in1(result);
}
```

Program 25: The FFT Computation, Part 2

```
// fft2.cpm

#include        "fft2.h"

atomic
void
join::in (Complex c)
{
    if ( passive(in1) )
        {  compute(c);  output();  active(in1);  }
    else
        {  result = c;             passive(in);  }
}

atomic
void
join::in1 (Complex c)
{
    if ( passive(in) )
        {  compute(c);  output();  active(in);    }
    else
        {  result = c;             passive(in1);  }
}

void
join::compute (Complex c)
{
    result += c;
}

void
join_mult_fork::compute (Complex c)
{
    result = (result + c) * W;
}

void
join_fork::output ()
{
    out->in(result);  out1->in1(result);
}

void
join_mult_fork::output ()
{
    out->in(-result);  out1->in1(result);
}
```

Program 26: The FFT Computation, Part 3

Finally, Programs 27, 28 and 29 contain the code used to build the N-point FFT process graph. Depending on how time-critical this creation task is, solutions range from entirely sequential, taking $\mathcal{O}(N \log N)$ steps, to maximally concurrent, taking just $\mathcal{O}(\log N)$ steps. Our solution follows an intermediate approach, in which the process creation is concurrent and takes $\mathcal{O}(\log N)$ steps, whereas passing references around is sequential for each process column, and takes $\mathcal{O}(N)$ steps.

```
// fft3.h


#include        "fft2.h"


processdef      build_top_fft : public CPM
{
public:
atomic          build_top_fft(int, join*, int, int, fft*);
};


processdef      build_btm_fft : public CPM
{
public:
atomic          build_btm_fft(int, join*, int, int, fft*);
};
```

Program 27: Building the FFT Graph, Part 1

```
// fft3.cpm


#include        "fft3.h"


fft_graph::fft_graph (int N, fft* outs)
{
    order   = N;
    inputs  = new relay[N];

    if ( N > 1 )
    {
        join*  j = new join[N];
        new build_top_fft(N, j, 0, N/2-1, inputs);
        new build_btm_fft(N, j, N/2, N-1, inputs);
        for (int i=0; i<N; i++)
            (j+i)->connect(outs+i);
    }
    else
    {
        inputs->connect(outs);
    }
}
```

Program 28: Building the FFT Graph, Part 2

```
// fft4.cpm


#include        "fft3.h"


build_top_fft::build_top_fft (int N, join* outs, int from, int to, fft* inputs)
{
    int  n = to - from + 1;

    if ( n > 1 )
    {
        join_fork*  f = new join_fork[n];
        new build_top_fft(N, f, 0, n/2-1, inputs);
        new build_btm_fft(N, f, n/2, n-1, inputs);
        for (int i=0; i<n; i++)
            f[i].connect(outs+i,outs+n+i);
    }
    else
    {
        fork*  f = new fork;
        f->connect(outs,outs+1);
        (inputs+bit_reverse(N,from))->connect(f);
    }
}


build_btm_fft::build_btm_fft (int N, join* outs, int from, int to, fft* inputs)
{
    int  n = to - from + 1;

    if ( n > 1 )
    {
        join_mult_fork*  f = new join_mult_fork[n];
        new build_top_fft(N, f, 0, n/2-1, inputs);
        new build_btm_fft(N, f, n/2, n-1, inputs);
        for (int i=0; i<n; i++)
            f[i].connect(outs+n+i,outs+i,W(N,from+i));
    }
    else
    {
        mult_fork*  f = new mult_fork;
        f->connect(outs+1,outs,W(N,from));
        (inputs+bit_reverse(N,from))->connect(f);
    }
}
```

Program 29: Building the FFT Graph, Part 3

Chapter 3

Implementation Issues

There are two major components to the C+- programming system: the translator from C+- to C++, and the C+- runtime system. This programming system is currently supported on the Mosaic, and on all systems that support the Cosmic Environment/Reactive Kernel (CE/RK) [31] message-passing primitives, which includes sequential computers, networks of workstations, and a variety of commercial multicomputers and multiprocessors.

The translator is written in C++, and is both compile-machine- and target-machine-independent. Most of the runtime-system code is portable as well, with the exception of a small set of C+- library functions that are illustrated in Section 3.1.

3.1 The Runtime-System Framework

The relationship between the C+- programming notation and the C+- runtime systems is symbiotic: Programs written in C+- require runtime-system support; C+- runtime systems are typically written in C+-.

Although most of the runtime-system code is portable, the resource-allocation requirements on various machines are quite different. Given a sufficiently large node memory, the amount of runtime-system support that C+- programs require is minimal. The runtime systems for C+- implementations on computers with workstation-size nodes typically consist of less than a thousand lines of C+- code. The Mosaic fine-grain multicomputer consists of nodes with severely restricted memory resources; hence, the runtime system for the Mosaic employs much more sophisticated runtime mechanisms. Various configurations of MADRE, the MosAic Distributed Runtime systEm, range from two to ten thousand lines of C+- code. MADRE was written by Nanette J. Boden, and its design and the distributed algorithms it employs are described in detail in her Ph.D. thesis [5]. This work demonstrates that the complexity of runtime systems for fine-grain multicomputers need not result in large penalties in speed, nor does it imply large chunks of noderesident code that reduce the available node memory even further. MADRE is itself a concurrent program that employs distributed solutions to manage distributed resources [5].

The mutual dependence of the C+- programming notation and the C+- runtime systems is only apparent. In fact, the runtime system is just a pre-written part of any user program — a part that includes an interface to the resource-allocation and communication capabilities of the machine it is running on. The C+- programming model and programming notation supply only the framework for implementing process management and data communication, striving not to restrict the spectrum of possible runtime-system implementations. The remainder of this section describes this framework. Since the primary target for executing C+- programs is the Mosaic, the names and default semantics of functions that we use correspond to message-passing communication primitives. This does not, however, imply that these primitives are the only ones that can be used; shared-memory communication primitives, for example, are equally suitable for implementing the necessary low-level routines.

3.1.1 Process Creation

An example of how process creation may be implemented in C+- is given in Program 30. In general, process creation consists of the following three stages:

- *Choosing a manager,* by invoking the `manager_CPM` function[1] corresponding to the type of the process being created. This function must return a pointer to the process that will be asked to instantiate the new process. It is possible to define multiple versions of this function, some of which may take arguments. For example, different versions may correspond to different process-placement strategies.

- *Requesting the creation* from the chosen manager by invoking the manager's `create_CPM` atomic action. The two arguments[2] correspond to the size of the process and the address of the constructor to be invoked. If the constructor takes arguments, those are passed as well. Various flavors of process creation can coexist in the system, with one of them selected at creation time.

- *Instantiating the process* is done by a manager process, not necessarily the one originally chosen: The creation can be delegated to other potential manager processes, and is eventually done in the consenting manager's address space [5].

[1]This function must be declared `static`, which is a C++ feature that makes a member function generic, associated with a certain class definition, not with any particular instance of that class.

[2]The `size_t` is a C++-defined integer type that can represent the size of the largest possible object (or process). The `entry_t` type is introduced by C+-, and will be described in Section 3.1.4.

```
processdef      Manager
{
public:
atomic  P*      create__CPM (size_t, entry_t, ...);
};

processdef      P : dynamic Manager
{
public:
static  Manager*  manager__CPM();
atomic            P();
atomic            P(int);
};

{
    new P;              // is equivalent to:
    P::manager__CPM()->create__CPM(sizeof(P),&P::P());

    new P(123);         // is equivalent to:
    P::manager__CPM()->create__CPM(sizeof(P),&P::P(int),123);
}
```

Program 30: Process Creation

3.1.2 Runtime Services

All of the `protected` and `public` members of a manager can be accessed by the processes it manages. This access is handled transparently by the compiler. The programmer need not be concerned whether some service is provided through regular inheritance or through dynamic inheritance, with the latter requiring one or more levels of indirection (Program 31).

3.1.3 Process Dispatch

A problem that emerges in the design of all operating and runtime systems is that of specifying an interface for invoking user programs. This task is typically done in an *ad hoc* way. For example, user programs written in C and run under UNIX must have a function called `main`, which is the user-code entry point. However, this approach does not enable the operating system code to merely call this function, since the address of `main` is not known at the operating-system linking time. The typical solution is to require that `main` always be at the same address, or to find its address at loading time.

Every C+- process has a fixed number of entry points, corresponding to its atomic actions, each of which could take different numbers and types of arguments, and return values of different types. If the runtime system itself is to be expressed in C+-, there must be a way of dispatching to any atomic action of any process, or

```
processdef      Manager                 // runtime-system code
{
protected:
        int     i;
        void    f();
};


processdef      P : dynamic Manager     // user code
{
private:
        int     j;
        void    g();
public:
atomic          P()
                {
                    j = 0;      // accessing local data
                    i = 0;      // accessing manager's data
                    g();        // calling local function
                    f();        // calling manager's function
                }
};
```

Program 31: Accessing Runtime Services

of any process in some predefined set. In the remainder of this section, we describe the C+- atomic-action dispatch mechanism.

As illustrated in Figure 3.1, every process `P` is a node of a process tree, with its path toward the root of a tree leading through its manager `M`, its manager's manager `MM`, *etc.* Several such trees may coexist on each physical node. Every `processdef M` that could be used as a `dynamic` base for some process definition, which means that an instance of `M` could be a manager of some process, must have a special atomic action defined, `atomic @M(entry_t)`, called the *dispatcher*. A generic dispatcher, `atomic @(entry_t)`, also has to be defined; its job is to dispatch to root processes of process trees.

The `entry_t` is a type introduced by the compiler, corresponding to any and all types of entry points of processes that *could* be defined with `M` as their dynamic base. A variable of this type can be used like a regular C++ member-function pointer, with one important distinction: one need not know the interfacing details of all atomic actions that a variable of type `entry_t` may be used to invoke. How arguments are passed to anonymous atomic actions is discussed at the end of this section. How values are returned from atomic action is presented in Section 3.1.9.

Specification 4 An execution of an atomic action of a process can be requested only from the body of its manager's dispatcher atomic action.

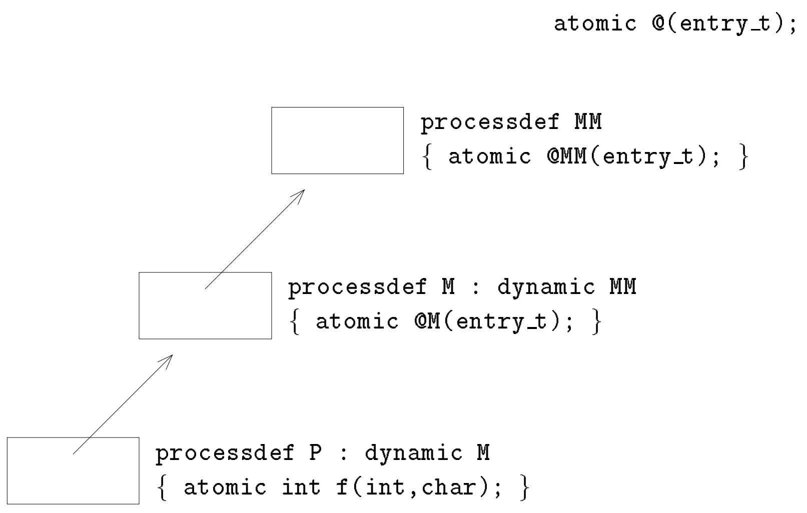

Figure 3.1: Process Dispatch

For the process hierarchy in Figure 3.1, this specification means that the execution of an atomic action of `processdef P`, say `P::f`, consists of executing the generic dispatcher `@`, which calls `MM::@MM`, which calls `M::@M`, which calls `P::f`. It is this layered execution that enables "managers" to manage other processes: *The semantics of atomic-action executions can be changed by modifying the runtime-system code.* As stated in The Annotated C++ Reference Manual: "...this opens vast opportunities for generalization and language extension in the general area of: What is a function and how can I call it?" [16]. This feature could strike the reader as intolerably under-specified and inviting of hacking and abuse. However, the safety properties of this mechanism are not as weak as they may appear to be. The runtime-system-specified mechanisms cannot be changed by users — the manager *always* gets to run before dispatching to the managed process. We have come to believe that the support for some mechanism of this kind is essential for a notation that is intended for expressing operating and/or runtime systems.

Another way of thinking about this layered dispatch mechanism is that every process provides a set of services (its atomic actions), and an *escape* mechanism to which it can defer the execution if it cannot handle the requested service itself.

Arguments to Atomic Actions

The memory layout of the arguments to atomic actions is the same as that for regular functions in C++, with additional arguments being passed to the dispatcher actions of the manager processes (Figure 3.2).

These additional arguments are, by default, generated by the compiler, but, as discussed in Section 3.1.7, this default behavior can be replaced by one defined by the programmer.

An additional feature is that the arguments are assumed to be members of the

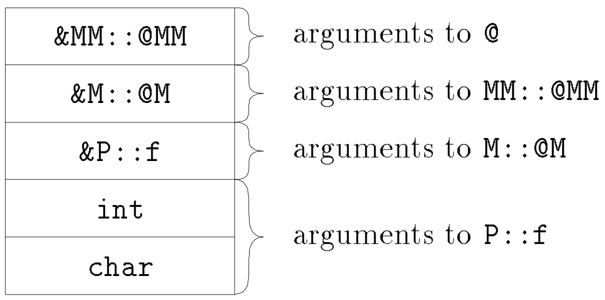

Figure 3.2: Atomic-Actions Arguments Layout

compiler-introduced structure `args_t`, and can be accessed as a unit through a pointer variable `args_t* args;` (similar to the `this` variable in C++).

3.1.4 The `pointer_t` and the `entry_t` Types

In the programming examples we made use of `pointer_t` and `entry_t` types, always referring to them as "introduced by the compiler." These two types are actually defined by the runtime-system in a file that has to be included by every C+- program (`<c+-.h>`). The C+- translator makes the structure of every process pointer the same as that of `pointer_t`, and the structure of every pointer to a member of a process the same as that of `entry_t`.

3.1.5 Process State

As discussed in the previous sections, the state of a C+- process consists of its:

- data members,
- active/passive set, and
- a pointer to the manager process.

What are the semantics of process assignment in the context of processes with the state defined above? The default C+- semantics for process assignment are bit-wise copying of data members and of the representation of the active-passive set; the pointer to the manager process is left untouched. The example in Program 32 shows process assignment as equivalent to sending a request to the source process to send a copy of itself to the specified destination process.

3.1.6 Process Migration

No notion of process migration is supported directly in C+-. A process pointer typically contains an absolute address of a piece of memory representing the

```
processdef      P
{
private:
        int     i;
public:
atomic  int     copy__CPM (P* pp)
                {
                    forward  pp->copy__CPM(*this);
                }
atomic  int     copy__CPM (P p)
                {
                    *this = p;
                    return  1;
                }
};


{
    P  p1, p2;

    p1 = p2;    // is equivalent to:

    await ( p2.copy__CPM(&p1) );
}
```

Program 32: Process Assignment

state of a process. However, the example in Program 32 shows how simple it is to copy the state of a process. Furthermore, with the ability of the runtime system to define the structure of process pointers (Section 3.1.4), the runtime-system framework described in this chapter was sufficient to implement distributed processes (Section 2.3.9). The support for distributed processes requires the same indirection mechanism that might be used for process migration. The work reported in [5] is a first step towards a thorough examination of the issues involved in process migration. The results presented in this work establish conditions under which, for example, process state can be shipped to where the atomic-action code is located just as readily as code can be cached where the process state is located.

3.1.7 Invoking Atomic Actions

As illustrated in Program 34, an atomic-action invocation consists of three stages:

- *Introductory Stage* — Upon calling `operator space` to determine the size of the argument list, the `operator head` is invoked to build the dispatcher list. Given a data type `TYPE` and a process type `PROCESS`, the default operator semantics are as follows:

```
size_t  operator space(TYPE t)
        {
            return  sizeof(t);
        }

static
void*   PROCESS::operator head(void* v, pointer_t p, entry_t e, size_t s)
        {
            return  operator send(v,e);
        }
```

- *Main Stage* — For each element in the argument list, the `operator send` is invoked. The default operator semantics are bit-wise copy:

```
void*   operator send(void* v, TYPE t)
        {
            TYPE*  tp = v;
            *tp = t;
            return  tp+1;
        }
```

- *Final Stage* — The `operator tail` is invoked, with no-op default semantics:

```
static
void    PROCESS::operator tail(void*, void*)
        { }
```

At the time of atomic-action execution, `operator recv` is invoked for each element in the argument list. The default semantics for this operator are a no-op (Program 33).

The set of operators described above provides runtime-system programmers with a powerful tool that they can use to define how process communication is actually implemented in terms of lower-level routines. The same set of operators is available to users. An example of an application that might benefit significantly from the ability to exercise total control is a program that implements communication-network protocols. The general usability of the above mechanism, however, is highly questionable: Once the compiler relinquishes control over data layout to a naive user, obscure problems abound. For a great majority of applications, the efficiency of the data-exchange mechanisms described in Section 2.4.5 is sufficient.

```
void    operator recv(TYPE t)
        { }
```

Program 33: Default `operator recv`

```
processdef      MM
{
public:
atomic          @MM(entry_t);
};

processdef      M : dynamic MM
{
public:
atomic          @M(entry_t);
};

processdef      P : dynamic M
{
public:
atomic  void    f (int, char);
};

{
    P*   p;
    int  i;
    char c;

    p->f(i,c);          // atomic action invocation is equivalent to

    {
        size_t  size = operator space(&MM::@MM)         // assuming there are
                     + operator space(&M::@M)           // no alignment problems
                     + operator space(&P::f)
                     + operator space(i)
                     + operator space(c);
        void  *b, *v;
        pointer_t  pp = p;

        b = v =     operator head (   pp, &MM::@MM, size);
            v = MM::operator head (v, pp, &M::@M,   size);
            v =  M::operator head (v, pp, &P::f,    size);

        v = operator send(v,i);
        v = operator send(v,c);

        v =  M::operator tail (   v, pp, &P::f,    size);
        v = MM::operator tail (   v, pp, &M::@M,   size);
                operator tail (b, v, pp, &MM::@MM, size);
    }
}
```

Program 34: Atomic-Action Invocation

3.1.8 Active/Passive

The active/passive mechanism, because of its simplicity and efficiency, is the C+- synchronization mechanism of choice. The runtime-system interface for this mechanism is presented in Program 35. If a different synchronization mechanism is required, it can be implemented following the same approach.

```
processdef      P
{
public:
atomic  void    f();
atomic  int     g();
};


atomic
void
P::f ()
{
    active  f;          // is equivalent to:
    P::active__CPM(&P::f);

    passive  g;         // is equivalent to:
    P::passive__CPM(&P::g);
}
```

Program 35: Active/Passive Implementation

3.1.9 Remote Procedure Call

When invoking an atomic action that returns a value, the sequence of events is identical to that described in Section 3.1.7, except that an extra argument is passed. This extra argument is the pointer to the currently-running process — the process that expects the reply. This pointer is obtained by calling the runtime-system-defined function `current_CPM()`.[3] The NULL extra argument implies that the returned value is not required.

Values Returned From Atomic Actions

Inside an atomic action, the extra argument is called `reply_CPM`. As illustrated in Program 36, returning a value from an atomic action is equivalent to invoking the `return_CPM(...)` atomic action of the process pointed to by the `reply_CPM` pointer.

[3]Note that it was not possible to use the `this` variable, because a process might be suspended while executing a non-member function.

```
processdef      P
{
public:
atomic  int     f();
};


atomic
int
P::f ()
{
    return  123;        // is equivalent to

    {
        if (reply__CPM)
            reply__CPM->return__CPM(123);
        return;
    }
}
```

Program 36: Atomic Actions Returning Values

Suspending A Process

Whenever a returned value is expected from an atomic action, the compiler introduces a placeholder for that value, and the runtime system is passed a pointer to this placeholder through the `wait__CPM(void*)` function. Multiple placeholders can be active at any time, as discussed in Section 2.2. When the process attempts to access the placeholder and finds it uninitialized, it suspends itself by invoking the `suspend__CPM()` function.

3.2 From C+- to C++

There are a number of reasons for translating from C+- to C++ instead of compiling from C+- directly to Mosaic code. First, this was a faster way to build a running system. Second, the wide availability of C++ compilers guaranteed machine-independence. Third, we had good experience in re-targeting the Gnu C++ compiler to produce excellent code for the Mosaic processor. And fourth, since C+- is syntactically so similar to C++, C++ debugging tools and other programming-support tools can be used with few or no modifications. One disadvantage of the translation approach is that the compile time increases, because programs must be parsed twice. A possible disadvantage is that some optimization opportunities may be lost when using C++ as an intermediate target notation. However, we have identified no such lost opportunities so far.

3.2.1 Parsing

The translator is a C++ program built within the framework of a Bison-produced parser [15]. Practically every person who has ever worked on a project that involved parsing of C++ has already expressed their distaste that C++ syntax cannot be described by an LALR(1) grammar. Nevertheless, we feel that our own distaste should be on record, too. We acknowledge that it is not the compiler writer, but the language user, who should be the ultimate judge of the value and style of a programming notation. However, if syntactic issues are subtle enough to be difficult for a compiler, what hope does a user have of not making obscure mistakes writing programs using that syntax? Fortunately, beginners tend to use a small set of basic language constructs, whereas experienced users tend to develop their own programming style from a subset of the rich C++ offering. In our experience, the complexity of handling the few special cases in parsing C++ is comparable to the complexity of all of the remaining issues of translating C+- into C++. Suffice it to say that we are looking forward to the ANSI standard for C++ syntax.

In our implementation of the translator, each grammar rule corresponds to a class definition. For example, given the grammar rule in Program 37, three

```
expression      :       assignment_expression
                |       expression  ,  assignment_expression
                ;
```

Program 37: An Example of a Grammar Rule

class definitions have to be written, as shown in Program 38. Parsing a C+- program generates a parse tree that consists of nodes that are instances of classes such as these illustrated in Program 38. We developed a program that, given an input grammar such as the one illustrated in Program 37, generates the default class definitions (similar to those described in Program 38), the code that builds the parse tree, and the default definitions of `output()` functions. The resulting program code is a parsing specification for Bison, which can be used to produce a default parser. When a source program is fed to this default parser, the parser builds the parse tree. It then invokes the `output()` function at the topmost level of the tree, thereby causing the entire source program to be produced as the output. This default behavior can be modified by defining additional elements of class definitions, by specifying extra actions to be taken while building the parse tree, and by providing customized versions of the `output()` routine for any class definition. This simple tool for developing programs for source-to-source transformation, a program of less than two thousand lines of C++ code, has been crucial to our ability to experiment with numerous versions of C+- syntax. This tool generates about two-thirds of the approximately 60,000 lines of C++ code of a complete C+- translator.

```
class   expression
{
        void    output() = 0;
};

class   expression0 : public expression
{
        assignment_expression*  member0;
public:
        void    output()
                {
                    member0->output();
                }
};

class   expression1 : public expression
{
        expression*             member0;
        assignment_expression*  member1;
public:
        void    output()
                {
                    member0->output();
                    member1->output();
                }
};
```

Program 38: A Part of the Definition of the Parse Tree

3.2.2 Code Generation

Once the hurdle of parsing C+- is overcome, the translation from C+- to C++ is a fairly simple task. The description of the runtime-system framework in Section 3.1 also specifies this translation task. Since the process concept is the only extension that C+- introduces to C++, the focus of the translator is on keeping track of processes and various other process-related types. The translator considers each segment of a source program to be a type transformation. For example, a process-pointer type, when dereferenced, is transformed into a process type, and a function call transforms a list of argument types into the type of the returned value. Since the translator keeps track of all of the type transformations in a program text, operations on processes are detected, and the replacement code, as illustrated in Section 3.1, is generated.

3.2.3 Code Splitting

In addition to the transformations described in Section 3.1, there is one more requirement on the translator. Since the Mosaic, a machine with limited node-memory resources, is the most important target machine for executing C+- programs, the C+- translator must provide support for code splitting. Pieces of code are cached in each node by the runtime system, and invoked through the indirect-function-call mechanism. A design decision had to be made on what the code-splitting target should be.

The default object-code unit provided by the regular C++ compilers is a piece of code produced by the compilation of one source file. We considered this default setup to be unacceptable. Programmers would have to organize their code according to the code-splitting policy rather than according to the programming-abstraction requirements of the application. This setup would unavoidably lead to loss of portability, whereby the source code would have to be rearranged and split into smaller pieces when moving to a machine with less node memory.

Given that the default code-splitting policy was deemed unusable, we identified three well-defined code-splitting targets. These three targets, with increasing granularity, are to split the code so that each piece corresponds to:

- an *atomic action* of a process,
- a *function* and/or an atomic action of a process, or
- a *block of code* within a function, with strictly sequential execution (no conditional execution).

The next-higher-granularity target would be equivalent to turning the runtime system into a pseudo-code interpreter.

If the block of code with strictly sequential execution is the code-splitting target, only code that is certain to be executed is ever brought to the code cache. However, this implies more frequent code-cache updates.

If the code corresponding to a function or an atomic action is the code-splitting target, there is no unnecessary code duplication, as every named piece of code is a stand-alone unit. In this case, an indirect-call overhead has to be paid for each function call.

Even though each of these options could be supported by the C+- translator, we decided to split the code into pieces that correspond to atomic actions of processes. This was the least-complicated and the best-understood approach, and it still allowed us to provide an experimental testbed that can be used to determine the effect of code-splitting granularity on the machine performance. Code of a function is linked with every atomic action that invokes it. Some of the runtime-system services, such as sending messages and creating new processes, are accessed by virtually every user process, and replicating that code would be equivalent to including a large fraction of the runtime system in the code of each user-process atomic action. Access to these services is through the indirect-function-call mechanism, but its specification is left entirely to the runtime-system implementation [5]. We consider this an acceptable compromise, particularly because any efficient code-caching policy must distinguish such often-used code anyway.

Bibliography

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

[2] William C. Athas. Fine-Grain Concurrent Computation. Caltech Computer Science Technical Report 5242:TR:87 (Ph.D. thesis), 1987.

[3] J. Backus. Can programming be liberated from the Von Neumann style?, A functional style and its algebra of processes. *CACM,* 21(8): 613–641, August 1978.

[4] Nanette J. Boden. A Study of Fine-Grain Programming Using Cantor. Caltech-CS-TR-88-11, 1988.

[5] Nanette J. Boden. Runtime Systems for Fine-Grain Multicomputers. Caltech-CS-TR-92-10, 1993.

[6] Andrew A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[7] K. Mani Chandy, Carl Kesselman. Compositional C++: Compositional Parallel Programming. Caltech-CS-TR-92-13, 1992.

[8] K. Mani Chandy, Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[9] K. Mani Chandy, Stephen Taylor. A Primer for Program Composition Notation. Caltech-CS-TR-90-10, 1990.

[10] William Douglas Clinger. Foundations of Actor Semantics. MIT AI Lab Technical Report AI-TR-633, May 1981.

[11] Ole-Johan Dahl, Edsger W. Dijkstra and Charles A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[12] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, Norwell MA, 1987.

[13] Edsger W. Dijkstra. Cooperating Sequential Processes. in *Programming Languages*, edited by F. Genuys, Academic Press, 1968.

[14] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[15] Charles Donnelly and Richard Stallman. BISON: The YACC-compatible Parser Generator. The Free Software Foundation, June 1992.

[16] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[17] Ian Foster and Stephen Taylor. *STRAND: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[18] A. Gotlieb *et al.* The NYU Ultracomputer — Designing and MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers,* 175–189, February 1983.

[19] Per Brinch Hansen. Structured Multiprogramming. *CACM,* 15(7): 574–578, July 1972.

[20] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. in *Artificial Intelligence: An MIT Perspective,* edited by Winston and Brown, MIT Press, 1979.

[21] Carl Hewitt and Henry Baker. Laws for Communicating Parallel Processes. IFIP-77, Toronto, August 1977, pp. 987–992.

[22] C. A. R. Hoare. Communicating Sequential Processes. *CACM,* 21(8): 666–677, August 1978.

[23] Waldemar Horwat, Andrew A. Chien, William J. Dally. Experience with CST: Programming and Implementation. in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* 1989.

[24] Kirk Johnson, Anant Agarwal. The Impact of Communication Locality on Large-Scale Multiprocessor Performance. MIT/LCS/TM-463, MIT, February 1992.

[25] Charles R. Lang, Jr. The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture. Caltech Computer Science Technical Report 5014:TR:82, 1982.

[26] Sigurd L. Lillevik. The Touchstone 30 Gigaflop DELTA Prototype. IEEE 0-8186-2290-3/91/0000/0671/$01.00 671–677, March 1991.

[27] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing,* pp. 294–299, Prentice-Hall, 1975.

[28] Charles L. Seitz. The Cosmic Cube. *CACM,* 28(1): 22–33, January 1985.

[29] Charles L. Seitz. Multicomputers. Chapter five in *Developments in Concurrency and Communication,* edited by C. A. R. Hoare, Addison-Wesley, 1990.

[30] Charles L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizovic, C. S. Steele and W-K. Su. The Architecture and Programming of the Ametek Series 2010 Multicomputer. in *The Third Conference on Hypercube Concurrent Computers and Applications,* Pasadena, California, 1988.

[31] Charles L. Seitz, Jakov Seizovic, Wen-King Su. The C Programmer's Abbreviated Guide to Multicomputer Programming. Caltech-CS-TR-88-1, 1988.

[32] Jakov N. Seizovic. The Reactive Kernel. Caltech-CS-TR-88-12, 1988.

[33] Jakov N. Seizovic. The Architecture and Programming of a Fine-Grain Multicomputer. Caltech-CS-TR-93-18, 1993.

[34] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, 1987.

[35] K. Stuart Smith, Arunodaya Chatterjee. A C++ Environment for Distributed Application Execution. MCC Technical Report ACT-ESP-275-90, 1990.

[36] Craig S. Steele. Affinity: A Concurrent Programming System for Multicomputers. Caltech-CS-TR-92-08, 1992.

[37] Leon Sterling, Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[38] Wen-King Su. Reactive-Process Programming and Distributed Discrete-Event Simulation. Caltech-CS-TR-89-11, 1989.

[39] Stephen Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, 1989.

[40] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1987.