1993

C++ : a concurrent object-oriented programming language

Huang, He

LAKEHEAD UNIVERSITY

CC++ - A CONCURRENT OBJECT-ORIENTED PROGRAMMING LANGUAGE

BY

He Huang ©

A THESIS SUBMITTED TO
LAKEHEAD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICAL SCIENCES

THUNDER BAY, ONTARIO

OCTOBER, 1993

ISBN 0-315-86174-6

ABSTRACT

Developing concurrent object-oriented programming (COOP) languages becomes an attractive research area since COOP languages are more suitable for simulation of real world objects and their interactions. After reviewing fundamentals of COOP languages and analyzing existing COOP languages, we propose a concurrent object-oriented programming language: CC++. CC++ is an extension to object-oriented programming (OOP) language C++. It is extended by introducing five keywords, and incorporating process concepts and communication and synchronization mechanisms into C++. Meanwhile it retains the syntax and semantics of C++. The language distinguishes class (or passive) objects from process (or active) objects. The class objects are the ordinary objects in C++. The process objects, however, are coarse-grain concurrent entities. Each process object has a sequential control thread and an optional public interface that can be accessed by other process objects. The communication and synchronization between interacting process objects are accomplished through remote function call(RFC), which is an extension of remote procedure call(RPC)[Han78]. Two types of RFCs, blocked RFC and unblocked RFC are distinguished in CC++. The blocked RFC, along with guarded function and forward mechanism performs synchronous message passing. The unblocked RFC, however, provides a way to express asynchronous message passing. The guarded function is based on guarded commands[Dij75], and is introduced for expressing and controlling indeterminism. The language has been implemented as an experimental system running on UNIX. The important objectives of CC++ are ease of programming, simple syntax, clear semantics and strong expressive power.

ACKNOWLEDGEMENTS

I would like to express my thanks and appreciation to my supervisor, Dr. Xining Li, for his guidance, advice and financial support throughout this thesis.

I would to thank my wife for her understanding, encouragement and typing the draft during many evenings and weekends she spent on this thesis.

I am grateful to the Natural Science and Engineering Council of Canada, the Senate Research Committee of Lakehead University and Lakehead University Library for their support.

Finally, I would like to thank my external examiner Dr. Brain Unger and internal examiner Dr. Maurice Benson for their comprehensive comments.

Contents

List of Figures

Chapter 1

INTRODUCTION

The object-oriented programming (OOP) paradigm is having a profound impact programming methodology. OOP fundamentally changes our way of thinking. In conventional procedure-oriented (POP) paradigm, data and operations which manipulate the data are separate concepts and are separately defined. Control sequences must be specified to manipulate the data. With this paradigm, a given problem is decomposed into a sequence of tasks and procedures, the solutions are found out by executing these tasks and procedures on a set of data. By contrast, OOP paradigm glues data and related operations into one entity, and specifies collections of coordinated objects to simulate components of the problems to be solved. For a given problem, entities, which are either physical objects or abstract concepts, are considered first. The relations among the entities are then considered. The solutions are obtained through constructing templates of objects and establishing their relations.

OOP paradigm provides people with a method of solving problems using object modeling. It is thus very suitable for describing or simulating real-world objects as well as their relations and interactions. However, real world objects usually do not exist in isolation. Many objects may coexist and interact or communicate with one another. Hence, an OOP language should be a concurrent language in nature. Designing concurrent object-oriented programming(COOP) languages becomes an attractive research area.

A concurrent object-oriented programming language is a language combining

object-orientedness and parallelism. It should be suitable for expressing and modeling the concurrent computation by object-oriented approach. COOP languages are essentially the same as concurrent object-based programming (COBP) languages with the exception that latter do not support any inheritance.

The major aspects in designing COOP languages are object-orientedness, concurrency, communication and synchronization. Although objects support concurrency in nature, some issues arise when we incorporate concurrency into programming. In this chapter, fundamental aspects in COOP language design, and issues that arise due to the combination of object-orientation and concurrency are discussed.

1.1 Object-Orientation

Supporting object-orientation makes the COOP languages differ from other concurrent languages. In this section, we give a brief review on the subject of object-orientedness. The more detailed discussion can be found in [KM90, Boo91, Weg89, Tho89, Weg90, Pok89, MNC+91].

In an object-oriented system, objects are the basic run-time entities. An object is a collection of operations that share a state. The operations are a set of functions or methods which determine the messages to which the object can respond. The shared state is private data that records the effect of operations. A public interface of an object determines accessibility of users to the object. Object-oriented programming languages are then built on this object model.

There are three important properties which characterize the essence of the object-orientedness. They are abstract data type (ADT), inheritance and polymorphism.

An ADT is a user-defined data type which glues the passive data and active operations into a single entity. In most OOP systems, an ADT is defined by the construct *class*. A class is a template from which objects can be created, and that other classes can reuse through inheritance. A class consists of two parts: specification and implementation. The specification part describes a user interface and the implementation part specifies the control sequences of manipulating data. In other words, an ADT encapsulates data and a set of operations. Encapsulation provides at least three advantages in programming. First, encapsulation makes data hiding possible. It provides protection to the state of an object. Secondly, encapsulation enhances modularity. It promotes the reusability of existing code, and encourages separate compilation. Third, encapsulation separates users of an ADT from its implementer. A user needs no longer to know the implementation details of an ADT. An implementer of an ADT, on the other hand, can ignore where the ADT is to be used. This makes implementation and application of an ADT relatively independent of each other and in turn makes software development and maintenance easier.

By contrast, procedure-oriented programming systems view data and operations as separate entities. A data type is simply a data structure. Data can be accessed by arbitrary operations without any protection. Hence it is the programmer's responsibility to apply correct operations on data. This results in tightly bound of implementation and application of data types. The modularity is therefore greatly reduced.

Inheritance refers to relationships among classes. Through inheritance, one class can share the structure and behaviors defined in other classes. Inheritance is accomplished through the class hierarchy which specifies the definition and implementation

of one class to be based on that of other existing classes. Two kinds of inheritance are distinguished: single inheritance and multiple inheritance. Single inheritance refers to the hierarchy in which one class is based on a single class. Multiple inheritance corresponds to the hierarchy in which one class is based on two or more classes. With single inheritance, a class which wants to inherit from many classes must be defined many times. The derived classes of this kind have a deep structured hierarchy. The deep structured hierarchy sometimes makes it awkward to use. With multiple inheritance, this can be done in one step. However, multiple inheritance raises another problem: a base class occurring more than once in a derivation. This is solved in C++ by declaring the base class as a virtual class.

Inheritance promotes reusability. It encourages programmer to construct new ADTs by reusing the code of existing ADTs. Without this property, each minor variation of a class would require code replication. The pith and marrow of the inheritance mechanism is allowing users to construct their classes reusing some related existing classes without introducing unwanted side effects. Inheritance is a unique contribution of the OOP paradigm that distinguishes OOP from OBP and other programming models.

Polymorphism is another feature supported by OOP languages. Polymorphism means that one form may be used to execute different actions. It provides a mechanism to handle different operations with the same name and hence provides a manner to maintain a generic interface for reusability. Polymorphism reduces code modification during system extending. Without this property, the reusability is also limited.

Polymorphism is implemented through either early (static) binding or late (dynamic) binding mechanisms. In C++, for example, polymorphism is achieved through

overloading functions and operators, and inheriting from virtual functions. Function and operator overloading is supported by early binding, while virtual functions are supported by late binding. The main advantage to early binding is efficiency and the disadvantage is lack of flexibility. For the late binding, however, the advantage and the disadvantage are just the other way round.

1.2 Concurrency

Concurrency refers to the potentially parallel execution of parts of a computation [Agh90]. Logically, the object model and concurrency are nicely in mesh because of the attributes of the object model. However, object-orientation and concurrency are different concepts. Object-orientation emphasizes data abstraction, encapsulation, inheritance and polymorphism, whereas concurrency stresses on *process* abstraction and cooperation, *i.e.*, communication and synchronization. Hence a concurrent executable object model should unify these two aspects. That is, each executable object in a concurrent system should represent both a data abstraction and a process abstraction (threads of control) [Boo91]. Such objects are called active objects. To distinguish active objects from the objects that lack control threads such as that in sequential systems, we call the latter passive objects. An active object model is more complex than a passive object model due to the fact that the former involves scheduling, communication and synchronization mechanisms. The inter-process and intra-process interactions result in a more complicated interface and internal structure of active objects than that of passive objects.

A process is a formal representation of a program in execution [LM89]. In a

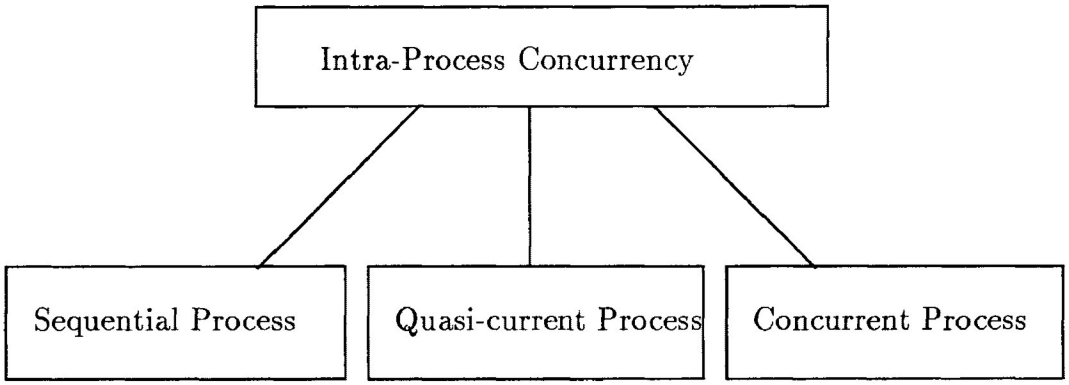

Figure 1.1: Classification of processes

COOP system, two kinds of concurrency should be considered. One is inter-process concurrency which refers to the concurrency between active objects. The other is intra-process (or internal) concurrency, which allows concurrency occurring within an active object. Active objects are concurrently executable. Such objects are represented by processes in execution. When intra-process concurrency is introduced, there will be more than one concurrent executable elements within a process. The smallest executable element within a process (or active object) is called a *thread.*

Processes can be classified according to the properties of intra-process concurrency as shown in Figure 1.1[Weg89, Weg90]. A process that has only a single thread of control is called a sequential process. A process which has more than one threads of control but has at most one active thread of control at a time is a quasi-concurrent process. A concurrent process with multiple threads of control active simultaneously is called a concurrent process.

A sequential process has a single thread of control. It may be suspended while waiting for receiving a message from other processes. An incoming message must

wait until the currently executing process is ready to accept it. As long as the message is accepted, the invoking operation must run to completion. An example of this class is the task in Ada where a rendezvous occurs when a called task accepts a message from a calling task.

A quasi-concurrent process has at most one active thread at anytime. The process, however, may have suspended threads in one or more waiting queues. An executing thread is suspended while waiting for a condition to become true. When the condition is satisfied, the suspended thread is resumed in execution. The difference between a sequential process and a quasi-concurrent process is that the latter has condition queues of suspended threads, and entry queues of threads. An incoming message invokes an active thread only if the current thread terminates or is suspended. To allow at most one thread be active is in fact to ensure mutual exclusion of accessing local data, while the ability to suspend threads internally provides more flexibility. However, suspending threads in the middle of transactions violates mutual exclusion in some extent. Transactions should not be interrupted during execution since they may temporarily be associated with a collection of resources. The monitor-like object-based languages such as ABCL/1 [BR89]and Orient84 [YT87] are based on quasi-concurrency.

A concurrent process has multiple active threads. A thread can be freely created in the process as soon as a incoming message invokes an operation. However, this gives the rise to the problem of how to ensure mutually exclusive access to the shared data. A thread is suspended if it attempts to access the shared data in critical regions until the access can be accomplished safely. Concurrent processes provide a mechanism to implement fine-grained concurrency with the increase in complexity

of control mechanism.

In sequential and quasi-concurrent processes, the units of modularity and concurrency are the same. These result in a simple design and implementation of the language. By contrast, concurrent processes allow units of modularity to contain multiple units of concurrency. This, on one hand, makes it more natural to model some applications. On the other hand, it makes the language design and implementation more complex because it requires distinct synchronization and communication mechanisms for inter- and intra-process concurrency at both the language level and system level.

COOP models can also be classified as either active object models or passive object models. The active object model unifies object-orientation and process abstraction. An active object is an object bound to a process with one or more threads of control. The active objects are run-time entities in concurrent systems. They are activated upon the creation of processes bound to them. They are disabled when the processes are destroyed. An active object has a public interface composed of operations to be accessed by other active objects. The invocation of an operation can not be performed by a normal function call as in sequential systems. It is performed through message passing: a caller sends a request, a callee receives the request, executes the operation on behalf of the caller and finally sends back the results. In the active object model, the control threads of an object can be either active or suspended depending on messages received by the object or some other conditions.

In this model, the incoming messages may be handled one by one or concurrently depending upon the mechanism of thread control and types of the process. In the case that incoming messages are handled concurrently, two variations of the model are

developed: static variation and dynamic variation. In the static variation, a fixed number of threads are created for each object upon its activation. An incoming request is randomly assigned to an idle thread to perform the request. If all threads are busy when a request message comes, the message is placed in a queue for service at a later time. This scheme limits the degree of intra-process concurrency. In the dynamic variation, however, a thread is dynamically created for an incoming request. The thread is destroyed when the request is serviced. This scheme provides a flexible degree of concurrency as needed within an object, but has additional cost for dynamically creating and destroying threads [CC91]. Many languages, for example, ABCL, POOL, Concurrent Smalltalk and BETA, are based on the active object model.

Passive object model is another object model used by some concurrent languages, such as Clouds [Das86] and Emerald [BHJL86, CC91]. By contrast to the active object model, an object in the passive object model is not bound to a specific process. The objects and processes are independent entities. An object is passive in the sense that the object can be visited by processes but can not initiate communications to others. Invocation of an operation in an object by a process is performed by normal procedure call. A process may execute within several objects to perform various operations. When this model is applied to a physical distributed system, a message passing mechanism must be used to communicate between machines, and a worker process must be created at the called machine to execute the requested operation on behalf of the calling process.

1.3 Communication and Synchronization

In concurrent and distributed computation, one of the major issues is communication and synchronization among the active objects or processes. Communication refers to the exchange of information between two processes [LM89]. Synchronization is a mechanism to establish some form of agreement among a set of processes [Li89]. In other words, synchronization mechanism ensures events in a computing system occurring in a proper order and with suitable temporal relationship [LM89]. Synchronization and communication are closely related and intertangled with one another. On one hand, synchronization mechanism is needed for handling communication and for ensuring mutually exclusive accessing shared resources among processes. On the other hand, synchronization is often achieved through communication. There are two basic patterns for accomplishing a communication: shared variables and message passing. For example, semaphores, conditional critical regions and monitors are three models which provide synchronization mechanisms via communication with shared variables. Client/sever model is based on asynchronous message passing, and remote procedure call(RPC)[Han78] combines concepts of procedure call with synchronous message passing. In this section, we first briefly review the basic communication patterns and process coordinations. More detailed discussion of communication and synchronization models will be addressed in Chapter 2.

In the shared variable pattern, shared variables are used as a means of communication. To exchange information, one process writes to a shared variable, another process simply reads from it. The interacting processes and shared variables are usually in the same address space, so the shared variables can be directly accessed

by each process. The advantage of this pattern is that it provides the fastest and simplest communication for the processes in the same address space. The disadvantage of the pattern is that it raises some other problems such as mutual exclusion of the shared variables and synchronization of accessing the shared variables between the writing process and the reading process.

With the pattern of message passing, processes share channels, which are abstraction of physical communication networks. These channels provide communication paths between processes and are accessed by two primitives: *send* and *receive*. To exchange information, one process explicitly sends a message while another process explicitly receives it. Channels can be global to processes or directly associated with processes, and they can provide one-way or two-way information flows. According to different schemes, message passing is classified into synchronous message passing and asynchronous message passing.

Synchronous message passing provides a two-way communication mechanism. It tightly couples the communication and synchronization in the sense that sending and receiving processes must be both in ready state to perform an exchange of information. A process sending a message is blocked until the other process is ready to receive the message. Since each communication synchronizes the sending and the receiving processes, the sending process is free to continue execution only after the communication is accomplished. With synchronous message passing, there is no need to buffer a message in a channel since the message is received as long as it is sent. Hence a channel serves only as a communication path.

By contrast, asynchronous message passing provides a one-way communication mechanism. With asynchronous message passing, the *send* primitive does not block

the sending process. The sending process does not care if the receiving process is ready to receive a message. It sends a message, and is free to continue its execution. The message sent is appended to the end of the channel's queue. The sending and the receiving processes execute independently, and a message might be received arbitrarily long after it has been sent. The channels serve both as communication paths and as message buffers that should have unbounded capability.

There is a special case with asynchronous communication: all processes share a single communication channel. This is so called generative communication [And91b, Gel85]. The shared channel is called tuple space. With generative communication, associative naming is used to distinguish different kinds of messages stored in tuple space.

The message passing provides a flexible communication and synchronization scheme and is suitable for physically distributed processes which do not share the same memory space. However, this pattern is less efficient than the shared variables in general.

Basically, there are two types of interactions (relationships) between concurrently executing processes that communicate via message passing. One is principal-and-subordinate type such as client/server model. The other is reciprocity type, *e.g.*, peers model. In the client/server model, only clients can initiate communication. Clients make requests to servers, and servers provide services to clients. In the peers model, however, both interacting processes can make requests to the partner and both processes can provide services. A problem that arises due to the interaction between processes and due to multiple processes sharing resources is interference among processes. Synchronization plays a role of avoiding interference between concurrently executable processes, either by mutual exclusion or conditional synchro-

nization. Mutual exclusion groups statements into atomic actions and thus hiding intermediate states to prevent undesired interleaving. Conditional synchronization delays processes in execution until required conditions hold.

1.4 Thesis Outline

This thesis presents a proposal for introducing concurrency to a sequential object-oriented programming language - C++. We chose C++ as the base language because it is more popular than many other OOP languages and it allows the reusability of existing C code as well as the C programming environment. Our proposal, named CC++, exploits coarse-grain concurrency within the sequential object-oriented programming paradigm. Important objectives of the language design are simple syntax, clear semantics, strong expressive power and high portability.

Chapter two gives the survey of closely related work. A brief survey of COOP languages can be found in [Nel91]. Some of these languages are commercial products, and some are research proposals. Broadly speaking, COOP language research falls into two categories: extensions to sequential OOP languages and designing new COOP languages. Research proposals in the former category inherit most features from more conventional OOP languages. and proposals in the latter category are either predicated on specialized parallel architectures or new programming model. The emphasis of our survey is placed on different communication/synchronization models and COOP languages extended from C++.

Chapter three presents our proposal - CC++. CC++ is an extension to C++. It distinguishes two categories of objects: process objects and class objects. Each

process object has its own sequential control thread and an optional public interface accessible by other process objects. A set of process objects is synchronized and coordinated by a Remote Function Call(RFC) mechanism. Indeterministic RFC's are selected or sequenced by guarded function declarations.

To illustrate usage of CC++, several typical concurrent applications are presented in chapter four. We show how CC++ is used in different problem solving patterns, such as pipeline, divide-and-conquer and inter-process cooperation.

The implementation issues of CC++ are discussed in chapter five. CC++ is designed to be a practical, usable, concurrent object-oriented programming language. An experimental compiler of CC++ has been developed. The run-time system of CC++ consists of a process scheduler based on context-switch technique and an IPC (Inter-Process Communication) kernel.

The last chapter gives conclusions and describes future work. Several questions, such as inheritance among active and passive objects and granularity of concurrency, remain to be answered. In addition, a detailed experimental study of different concurrent programs in CC++ is required.

Chapter 2

SURVEY OF RELATED WORK

Concurrent object-oriented programming(COOP) language design is still fairly new research area. Although much work has been done in this field, there are very few mature languages or systems available [BST89, Nel91]. Efforts in COOP language design have being made in two directions. One direction is to extend existing OOP languages to COOP languages. The other is to develop new COOP languages. In this chapter, we first briefly describe two approaches in design of COOP languages, then compare different communication and synchronization models. Finally, we focus on several COOP languages extended from C++.

2.1 Approaches in Designing COOP Languages

One way of designing a COOP language is to extend an existing OOP language. The languages and research proposals in this category inherit most characteristics from conventional OOP languages. The purpose of extending existing languages is to maximize the reusability of existing languages and system facilities. One advantage of doing so is less expensive in design and implementation of COOP languages. Another advantage is that it is easier for people to accept or adopt a familiar language. The third advantage is that it helps reuse of existing application software. The difficulty one might have is to make the derived languages consistent with the base languages in aspects of syntax and semantics. The expressive power is also a main issue of

concern.

Several sequential OOP or OBP languages, such as Smalltalk, Ada, Lisp and C++, are often chosen as base languages. For example, CST [DC89], DCST [NYT+89] and Actalk [Bri89] are based on Smalltalk-80. DRAGOON [DMCB+89] is an extension to Ada. Acore [Man89] and Lamina [DS89] are Lisp based Languages. ACT++ [KL89a], Concurrent C++ [GR88], μC++ [BDSY92], Sim++ and SimD [Bae91] are languages developed on C++.

These languages essentially use the concepts of process to expose concurrency. Either synchronous or asynchronous message passing mechanism is used with the combination of some sort of sequencing mechanism to coordinate and synchronize concurrently executing objects. For instance, CST and DCST extend the message passing facility of Smalltalk-80 and combine it with guarded communication mechanism. Actalk, Acore and ACT++ combine their base languages with the actor model. Sim++ and SimD, however, incorporate C++ with Time Warp paradigm. These languages are often implemented by introducing a few new keywords and structures, or by using build-in libraries.

On the other hand, designing new COOP languages is more expensive than extending existing OOP languages. It is, however, possible to design a new language that smashes the trammels of old languages. In addition, some special applications may need languages with special properties that can not be incorporated into the the existing languages. Often, the languages of this group are either designed for special parallel architectures or based on some kinds of unconventional programming models. For instance, POOL [AdH90] is a COOP language designed to run on a specialized parallel architecture called the decentralized object-oriented ma-

chine(DOOM). A'UM [YC89] is based on stream-based model, while CLIX [HC87] and O-CPU [Mel89] are on logic-based model.

2.2 Communication and Synchronization Models

In this section, we discuss several communication and synchronization models, such as client/server, peers, monitors, RPC, rendezvous and the actor model, which are suitable for or extensible to concurrent object-oriented computing.

Client/server model [And91a] is based on asynchronous message passing. In this model, there are two kinds of processes, namely client processes and server processes. A client is a triggering process while a server is a reactive process. The relationship between a client and a server is the principal and the subordinate. A client initiates communication by sending a message to a channel, and a server acquires the message by receiving from the same channel. When a client sends a message to a server, the server might be able to respond to the request immediately if it is currently available, or it might save the message in a queue for later processing if it is now busy with some other job. A client process continues execution and terminates after execution of its body. A server is usually a non-terminating process and can provide services to many clients. It waits for requests, then reacts on them. Figure 2.1 shows the interaction between a client and a server.

The *receive* primitive plays two important roles. First, it specifies actions of receiving a message. A process has to explicitly apply *receive* to a certain channel to accomplishing communication. Secondly, it provides a synchronization mechanism. A process is blocked at the point of executing a *receive* statement if there is no

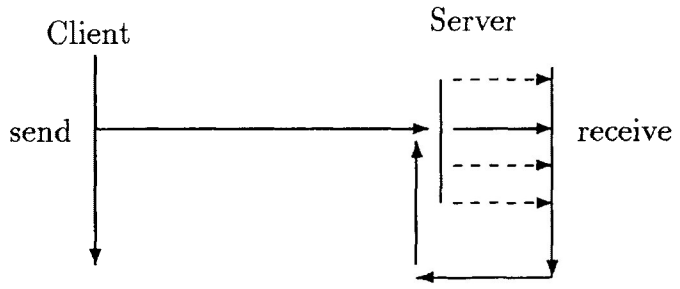

Figure 2.1: Interaction between a client and a server

message available. Thus, a client can simulate a synchronous communication using a *send* statement followed by a *receive* statement.

A peer [And91a] or an agent [Boo91] is an active object that can both operate on other active objects and be operated on by other active objects. A peer is a combination of client and server. As a client, a peer can request services from other peers. As a server, a peer can accept request from and provide service to other peers. Synchronization between two peer processes can be achieved by the proper arrangement of *send* and *receive* primitives. The peers model provides a way of conversation between processes. Figure 2.2 shows a possible conversation between two peers.

Indeterminism is an issue related to conditional synchronization. As long as a process provides more than one service, *i.e.*, the process has more than one operation that can be requested by other processes, indeterministic selection of execution of an operation may occur. The reason is that the interactions among processes are not always known in advance, but rely on some run-time conditions. When faced with several requests, the process must decide which one should be served. Two commonly accepted strategies are used in handling this problem. One is based on

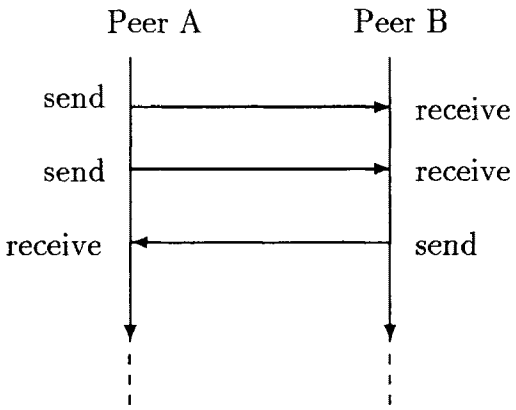

Figure 2.2: Conversations between two peers

guarded commands [Dij75], such as guarded communication [And91a] and select statement [BST89, Weg90]. Another is based on the actor model [Hew77].

With the guarded communication or select statement, statements are guarded by a Boolean expression followed by a communication statement. The general form of a guarded communication is:

$$B; C \rightarrow S$$

where B is an optional Boolean expression, C is a communication statement, and S stands for a statement list. B and C constitute a guard. Omitting B implies a true Boolean value. When B is evaluated true and C is performed, the guard succeeds and then S is executed. If the evaluation of B is false, the guard fails and the statements can not be executed. If B has true value but C can not be executed without causing delay, the guard is blocked. The statement list S can not be executed until the guard succeeds later. The Boolean expression B must be side-effect free. Otherwise inconsistent process state may be caused due to multiple evaluation of the Boolean expression.

By contrast, the actor model uses the behavior changing mechanism in solving the problem of indeterminism. A behavior of an actor decides what message the actor can receive. An actor has only one behavior at a time, and can change its behavior dynamically at run time. An example will be discussed later in this section.

The concept of monitor [Han73, Hoa74, And91b, And91a] was developed for managing mutually exclusive access to shared concurrently accessible resources and for synchronizing processes. To ensure mutual exclusion, a monitor allows at most one process at a time to execute within the monitor. This guarantees that no processes can interfere with each other for a shared resource.

In monitor model, condition variables are used for synchronizing processes. A condition variable delays a process that can not safely continue executing until the monitor's state satisfies some Boolean condition. Two statements, *wait* and *signal*, are used in a monitor for this purpose. The execution of *wait* causes the executing process to be suspended at the rear of a delay queue related to some condition. The execution of statement *signal* then awakens the process at the front of the delay queue related to the condition.

Processes access to monitors by procedure calls. A calling process transfers control to the called monitor and gets the control back when its request has been served. In case of some condition not been satisfied, the calling process is suspended, and the monitor can be accessed by another process. Thus communication and synchronization among processes are realized in mutually exclusive manner. Since a monitor serves as a shared resource, it is suitable for the concurrent systems that share memory space or have memory space overlapped. Figure 2.3 shows the interaction between processes and a monitor.

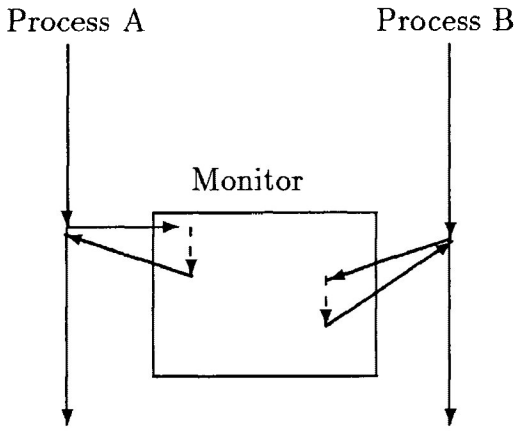

Figure 2.3: Interaction between processes and a monitor

A monitor has the functionality similar to a server. They both provide services to some kinds of processes, and provide communication and synchronization mechanisms between those processes. However, they are different in essence. First, a server is an active object, whereas a monitor is a passive object. An active object has one or more control threads, but a passive object does not. Secondly, a client operates on a sever by means of message passing, but a process makes procedure calls to access a monitor. Third, a server can not suspend a client whereas a monitor can. When a server starts servicing some request, it has to continuously execute to the end of the service. Incoming messages are buffered in the communication channels if the server is busy. The server responds to the next message only after finishing the current service. A monitor, however, can suspend a process at the middle of an operation by appending the process in a waiting queue. The monitor is then ready to be accessed by another process. The suspended process is awakened later when some Boolean condition is satisfied. This provides flexibility to handle some operations

which depend on some conditions in execution.

As mentioned in client/server model, the primitives *send* and *receive* are powerful enough to express synchronous message passing. To simulate a synchronous communication or the two-way information flow between a client and a server, however, both client and server must explicitly execute two message passing statements, *i.e.*, the client executes a *send* followed by a *receive* and the server executes a *receive* followed by a *send.*

Remote procedure call(RPC) is a higher level message passing construct [And83]. It combines aspects of procedure call and synchronous message passing, and thus can be used to express synchronous message passing directly. In RPC, a caller executes a *call* statement to initiate a synchronous message passing. The *call* could be translated into a *send* immediately followed by a *receive.* Thus after the caller sends the values of arguments to an appropriate callee, the caller process delays until the service has been performed and the results have been returned(if any). The callee is a module declared as a set of procedures. This process receives messages (values of parameters), executes the procedure on behalf of the caller, and sends back to caller the results or acknowledgements. Figure 2.4 shows the diagram of the control flow of a RPC between a client and a server.

With RPC, the synchronization between a calling process and a called process is guaranteed by the RPC mechanism. However, we still need some way to synchronize within a called process. When a process receives a remote call, it must decide whether or not the call can be executed right away. The complexity of the decision mechanism depends on the type of the called process. If a called process is a sequential or a quasi-concurrent process, *i.e.*, at most one procedure at a time is active,

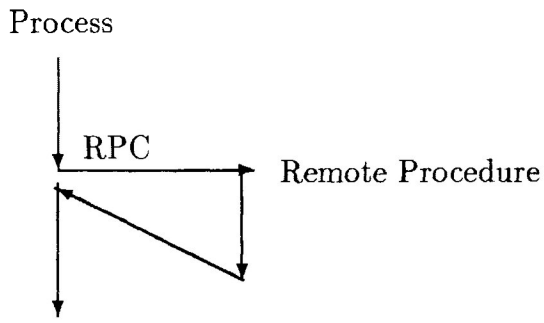

Figure 2.4: An RPC between a client and a server

no concurrent execution of procedures occurs. In this case, shared variables in a process are automatically protected against concurrent access. However, conditional synchronization mechanism is still needed for the problem of indeterminism. The common way is using the behavior change mechanism or guarded communication as discussed earlier. If the called process is a concurrent process, *i.e.*, more than one procedure are allowed to be active simultaneously, concurrent execution of multiple threads in a process happens. In this case, both mutual exclusion and conditional synchronization mechanism are needed for the intra-process concurrency. The implementation of the latter is more complex than the former. A shortcoming of RPC paradigm is that a process can not explicitly specify acceptance of a message. The acceptance of a message is implicitly accomplished. This makes the RPC mechanism unable to simulate conversations between processes.

Rendezvous is another high-level construct that provides a mechanism for expressing synchronous message passing. It combines the aspects of RPC with that of the peers model. With rendezvous, a calling process still uses a *call* statement to perform a remote procedure call. The *call* has the same properties as that in

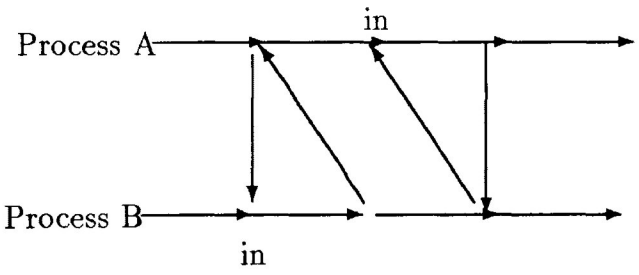

Figure 2.5: Control transfer between processes in rendezvous

RPC. But the called process must specify clearly what message to be accepted at the point of acceptance by using *in* or *accept* statement. The *in*(or *accept*) has a special property. It delays the called process until there is at least one message asking for the service provided by the called process at the suspended point. A rendezvous occurs only when the caller is executing a *call* statement and the callee is executing an appropriate *in* statement that accepts the caller's request. Thus rendezvous provides a mechanism to perform synchronization communication at the point chosen by called processes. Like in RPC, guarded communication can be employed in rendezvous for the problem of indeterminism. Figure 2.5 shows how two processes make a rendezvous.

RPC and rendezvous mechanisms are high-level constructs for expressing synchronous message passing. However, with RPC, the interacting processes are client and server processes, whereas the processes involving in rendezvous are peers. With RPC, a called process does not explicitly specify acceptance of requests, and concurrent execution of procedures in a process can be implemented by either creating new threads or assigning an idle thread to handle the request as soon as requests have been received and some conditions are satisfied. By contrast, a called process

in rendezvous has an execution body which specifies the acceptance of messages. The requests are serviced one at a time rather than concurrently. Hence the called process is a sequential process. The advantage of rendezvous is that it provides an inter-process communication mechanism that RPC does not support. But rendezvous does not support concurrency within a process.

The actor model was introduced into concurrent computing as a way to view control structures in artificial intelligence applications [Hew77]. It was later extended by many others [HA79, Agh86, Agh90].

Actors are self-contained active objects. They interact with one another through asynchronous message passing. In this model, behaviors of objects are viewed as functions of incoming communications. This model has three simple but powerful primitives: *create*, *send* and *become*. The first primitive *create* spawns new actors according to behavior description and parameters. It provides a mechanism to create resources dynamically in concurrent systems. The second primitive *send* carries out asynchronous communication among actors. The last primitive *become*, which is the most novel concept of the actor model, specifies replacement of behaviors. That is, actors can replace their old behaviors by new behaviors. Figure 2.6 conceptually presents the behaviors of actors and their interactions.

In the actor model, communications are accomplished through mail boxes. Each actor has a unique mail box whose address serves as its identification. The address of the mail box is determined at the time an actor is created. The incoming messages to an actor are buffered in a queue of its mail box. They are read one at a time in the order of first in first out(FIFO). An actor can send a message to another actor only if the sender knows the address of the receiver.

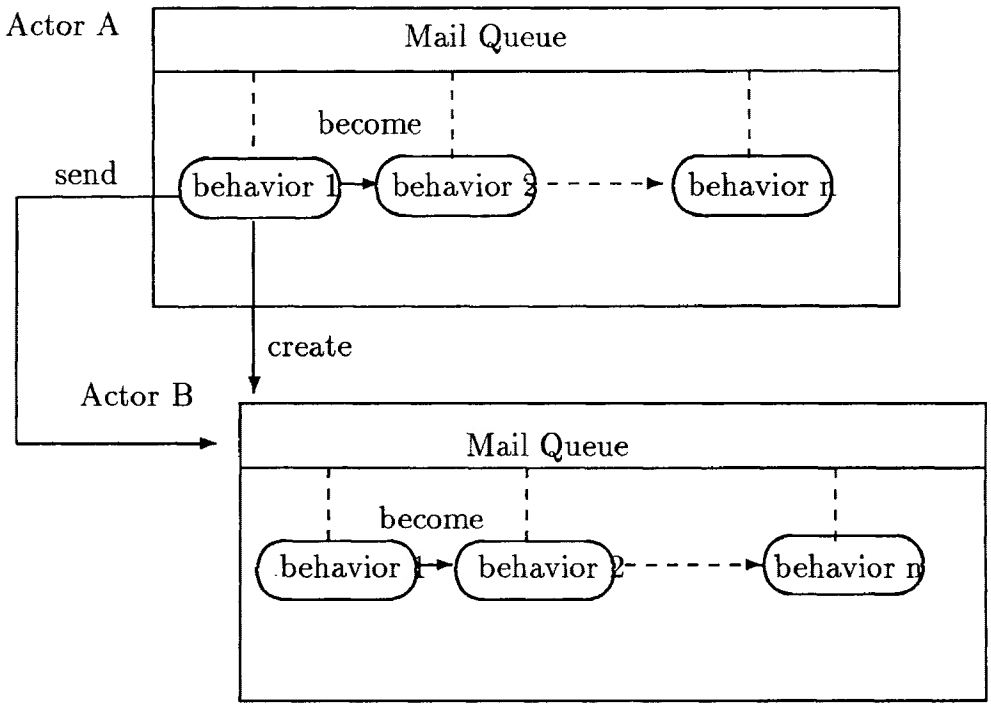

Figure 2.6: Actors, their behaviors and interactions

The distinguishing feature of the actor model is the *become* operation, which is used to solve the problem of indeterminism. In the actor model, the state change is specified using behavior replacement. The behaviors of an actor determine how the actor reacts to requests from other actors. The *become* operation explicitly specifies the transformation of the behaviors from one to another. Consider a bounded buffer as an example. A bounded-buffer actor may perform two actions, putting an item into a buffer or getting an item from the buffer. It may accept two kinds of messages:*put* and *get*. It may have three behaviors: *put_behavior*, *get_behavior* and *put_or_get_behavior*. In the *put_behavior*, the actor only receives *put* request. If there is no such kind of message in its mail box, the actor simply waits until one comes. In the *get_behavior*, only *get* message is received. In the *put_or_get_behavior*, both *put* and *get* messages can be received. Hence, if the buffer is empty, the actor must be in *put_behavior*, and if the buffer is full, it must be in *get_behavior*. Otherwise, the actor must be in *put_or_get_behavior*. In this behavior, the requests are serviced indeterministically upon the order of FIFO. The primitive *become* then acts like a switch that transforms the actor's behaviors from one to another.

2.3 COOP Languages Based on C++

In this section, we discuss some COOP languages that are closely related to our design. They are all extensions of C++ which is in turn extended from C. C++ is more popular than many other OOP languages because of its reusability of existing C code, its programming environment and C's popularity. The languages that we are going to discuss in this section are Concurrent C++, μC++ and ACT++. The

comparison of major properties of these languages against our proposal can be found in the final chapter.

Concurrent C++ [GR88] was developed by AT&T Bell Laboratories in 1988. The objective of this language was to integrate Concurrent C [GR86] and C++ to produce concurrent programs running on non-shared memory multicomputers. The language is a superset of Concurrent C and C++. The former provides concurrent programming facilities into C, while latter introduces object-oriented programming paradigm into C.

In Concurrent C, the keyword *process* is used to expose concurrent executable entities. A process is an instance of a *process definition.* A process definition consists of a *process type* and a *process body.* The process type is the specification of the public interface of the process. The process body specifies the implementation of the process. Each process is a sequential process that has a single control thread. A process must be explicitly(dynamically) created at run-time.

The communication model used in Concurrent C is synchronous and asynchronous message passing mechanism. Synchronization is achieved through rendezvous. A rendezvous is accomplished through a synchronous *transaction call* declared in a process specification. A synchronous transaction call is performed by two steps. First, a calling process initiates a transaction call. Secondly, a called process accepts the call through guarded *accept* statement and then executes the transaction and finally returns results, if any. The guarded *accept* statement and guarded *select* statement are used to decide indeterministically the transaction to be executed at run-time. A *by* clause is used to order the execution of outstanding transactions of the same kind.

The Concurrent C++ merges Concurrent C and C++. It inherits all the con-

current programming facilities from Concurrent C. Two kinds of user defined data types are distinguished: *class* and *process*. Classes instantiate passive objects, while processes instantiate active object.

The structure of a process is different from that of a class. A process structure has the following properties.

1. A process type(specification) only provides a public interface. It does not contain a data structure. All members are in public scope and are so called transactions similar to public functions in a class.

2. A process does not have constructors or destructors. It has a *body* which specifies the execution of the process, and contains the definitions of all the transactions.

3. Transactions are indeterministically chosen for execution through guarded *select* statements and guarded *accept* statements.

The above properties have caused some drawbacks. First, the specification without data structure violates the encapsulation of an ADT. This reduces the modularity of a data type. An implementer of a process has to decide the data to be manipulated by transactions. Secondly, a process does not have the inheritance hierarchy due to its body structure. In addition, transaction overloading is not allowed in a process because this requires that parameter types be specified in *accept* statements. These problems, in some extent, limited the reusability and polymorphism of an OOP language.

μC++[BDSY92] is a COOP language developed by Department of Computer Science at University of Waterloo. The language is also an extension to C++, and

is designed to run on uniprocessor or multiprocessor shared memory systems. In μC++, synchronous communication mechanism is supported via routine calls. Data is transmitted by argument passing, and results are returned as values of routine calls.

μC++ provides five user defined data types, *i.e.*, *class*, *coroutine*, *monitor*, *coroutine-monitor* and *task*. The *class* is the original construct in C++. The rest are new types extended from the *class* construct. Their instances are called *class-object*, *coroutine*, *monitor*, *coroutine-monitor* and *task* respectively. A coroutine is a passive object with execution-state. Its execution can be suspended and resumed explicitly. A monitor is also a passive object. It is a class-object with mutual exclusion implicitly implemented within the object. In a monitor, guarded statements can be used to determine the member to be executed next. A coroutine-monitor is a coroutine with mutual exclusion. It combines the properties of a coroutine and a monitor. Since monitors and coroutine-monitors are embedded mutual exclusive mechanism, they have the capability to protect simultaneous access to their data. In μC++, tasks are the only active objects with their own threads of control and execution states. All kinds of objects in μC++ can be created statically or dynamically as in C++. Their syntax and semantics are similar to that of C++. For example, an μC++ block cannot terminate until all statically declared tasks within it terminate. A dynamically created task in the block does not have to terminate before the block terminates. Such a task may be destroyed by *delete*.

In μC++, each user defined data type can inherit from the same type. However, inheritance among different types and multiple inheritance among the same types are not supported. The new types allow function overloading, but the overloaded func-

tions are not distinguishable in the *accept* statements. Hence the *accept* statements accept any calls with the same name [BDSY92].

ACT++ [KL89a] is another COOP language based on C++. This approach differs from Concurrent C++ and μC++ by combining C++ with the actor model [Agh86], and being designed for distributed real-time applications.

In ACT++, there are two kinds of objects: *actors*(active objects) and *passive-objects*(normal class-objects). Actors are concurrently executable entities. They are dynamically created at run-time. The communication between actors is accomplished through asynchronous message passing through mail boxes. Two types of messages are distinguished. They are *request messages* and *reply messages.* Each actor contains a unique *Mbox* variable and may have many *Cbox* variables. These variables serve as identification of the actor. The *Mbox* variable is used to send request messages, while the *Cbox* variables are used to send reply messages.

The most distinguishable property of the language is that of indeterminism mechanism adopted from the actor model. In an actor, its behaviors determine how it reacts to requests from other actors. Different behaviors respond to different request messages. A *become* primitive is then used to specify the behavior replacement of an actor. Specifying behavior replacement by *become* operation indicates that the current behavior has finished modifying the state of the actor, and the new behavior can start to process the next message. This results in the potential for concurrency inside an actor. One side effect of the *become* operation is that it sometimes forces the splitting of a coherent object definition. One example of this case is given in [KL89a].

ACT++ supports both inter-process and intra-process concurrency. The intra-

process concurrency is at the behavior-level rather than the instruction-level. The behavior-level concurrency refers to the concurrency achieved through the *become* operation. Inheritance of actor classes is also supported in ACT++. The conflict of concurrency and inheritance is solved using the model of object manager and behavior abstraction [KL89b]. The language, however, does not support method overloading in actor classes.

Chapter 3

CC++: A COOP LANGUAGE

CC++ is an extension to C++[LH93]. It exploits coarse-grain concurrency within the sequential object-oriented programming paradigm and distinguishes two categories of objects: class objects and process objects. Class objects are passive objects while process objects are active objects. A class object is an aggregation of data with associated operations. Such objects retain the same syntactic forms and semantics as in C++. On the other hand, a process object is a self-contained, coarse-grain concurrent entity with its own (implicit or explicit) control thread and an optional public interface accessible by other process objects. A set of process objects is synchronized and coordinated by a Remote Function Call(RFC) mechanism. Indeterministic RFC's are selected or sequenced by guarded function facilities. The proposed language supports multiple inheritance among process types, and from class type to process type. In addition, it supports function overloading and virtual functions within both class and process types.

Syntactically, CC++ is extended from C++ by introducing five new keywords and a form of guard declaration. The new keywords are *process*, *self*, *unblock*, *create* and *forward*. Key word *process* is used to define processes. *Self* is an inherent pointer to a process object like *this* pointer to a class object in C++. Process objects must be created dynamically by using *create*. Functions declared in a process can be either blocked or unblocked, where blocked functions provide a tool for synchronous communications among process objects while unblocked functions for asynchronous

communications. Guard and *forward* are used to solve indeterministic problems in concurrent programming.

CC++ is designed to be a practical, usable, concurrent object-oriented programming language. Important objectives of the language design are simple syntax, clear semantics and strong expressive power.

3.1 Process Declaration, Creation and Execution

A class is a user-defined type for creating passive objects, *i.e.*, class objects. A process is also a user-defined type for creating active objects, *i.e.*, process objects. The syntax of a process declaration is almost the same as the class declaration in C++ except for a few restrictions and extensions. First, keyword *process* is used instead of *class*. For example:

```
process A {
   int i;
   double d;
protected:
   int h();
public:
   A();
   int f(int);
   void g(double);
   ~A();
};
```

Like in C++, there are three access-specifiers in CC++: *public*, *protected* and *private*. The *public* label declares a public interface (excluding constructors) of the process. A process object can be accessed only through its public interface. The

protected and *private* labels have the same meanings as in C++. In the above example, A() is a constructor, and ~A() is a destructor. Process member functions f(), g() and ~A() constitute the public interface. The member function h() is a protected function, and variables *i* and *d* are private members. From now on, we use "process interface" to refer to the public part of a process declaration.

Inheritance is one of the major properties that characterize the object-orientedness. CC++ retains the inheritance conventions of C++ and defines that a process can be derived from one or more processes or classes but a class can not be derived from any process. For example,

```
class B1 { ... };
process B2 { ... };
process D1 : public B1 { ... }; // legal
process D2 : public B2, public D2 { ... }; // legal
class D3 : public B1 { ... }; // legal
class D4 : public B2 { ... }; // illegal
```

The visibility of a derived process are same as those of a class. For example, a destructor inherited by a derived process is invisible to other process. When a process is derived from a set of base classes, the semantic change from C++ inheritance is that objects created through this newly derived type become active objects.

Polymorphism is implemented through overloaded function and virtual function in C++. CC++ extends these language features to process type. For example, the following process definition overloads function *mult(...)* by different argument signatures.

```
process multiplier {
```

```
    ...
public:
    int mult(int,int);
    double mult(double, double);
    ...
};
```

The next example demonstrates the use of virtual function in process types:

```
process employee {
    char *name[20];
public:
    virtual unblock report();
    ...
};


process manager: employee {
    ...
public:
    unblock report();
    ...
};
```

A process object is created dynamically by using the keyword *create*, which is similar to *new* in C++. CC++ does not allow the static creation of process objects. For example,

```
A *pa;
pa = create A(); // legal: dynamic creation
A a(); // illegal: static creation
```

When a process object is created successfully, a pointer to the process object is returned. However, a process pointer is not the common sense pointer of C++,

i.e., it is no longer a memory address. A process object is referenced by the pointer obtained in the process creation. A special process pointer, *self*, is associated with each process object and is initialized upon the creation of the process object, just like the special pointer *this* assigned to each class object.

A process object starts execution as soon as it is created. The execution thread of a process object usually contains two phases: explicit control and implicit control. Explicit control is defined by the constructor of a process. This phase constitutes the explicit behaviors of a process object. It is executed only once immediately after the creation of a process object. If a process is derived from one or more processes(or classes), its execution will also invoke the constructors of base processes(or classes). These constructors are executed in the order of derivation. A process can have explicit control or implicit control or both. A process with no public interface has only explicit control. In the absence of the constructor or at the end of constructor's execution, a process object falls into the implicit control phase. This phase performs a control sequence iteratively: selecting and executing a function upon incoming RFC's. Figure 3.1 shows the control flow diagram of explicit and implicit control phases.

The termination of a process object can be controlled by one of the following three ways:

1. the process terminates upon the completion of its constructor if the process does not have a public interface;

2. the process is destroyed when it is in implicit control phase and its destructor is called explicitly;

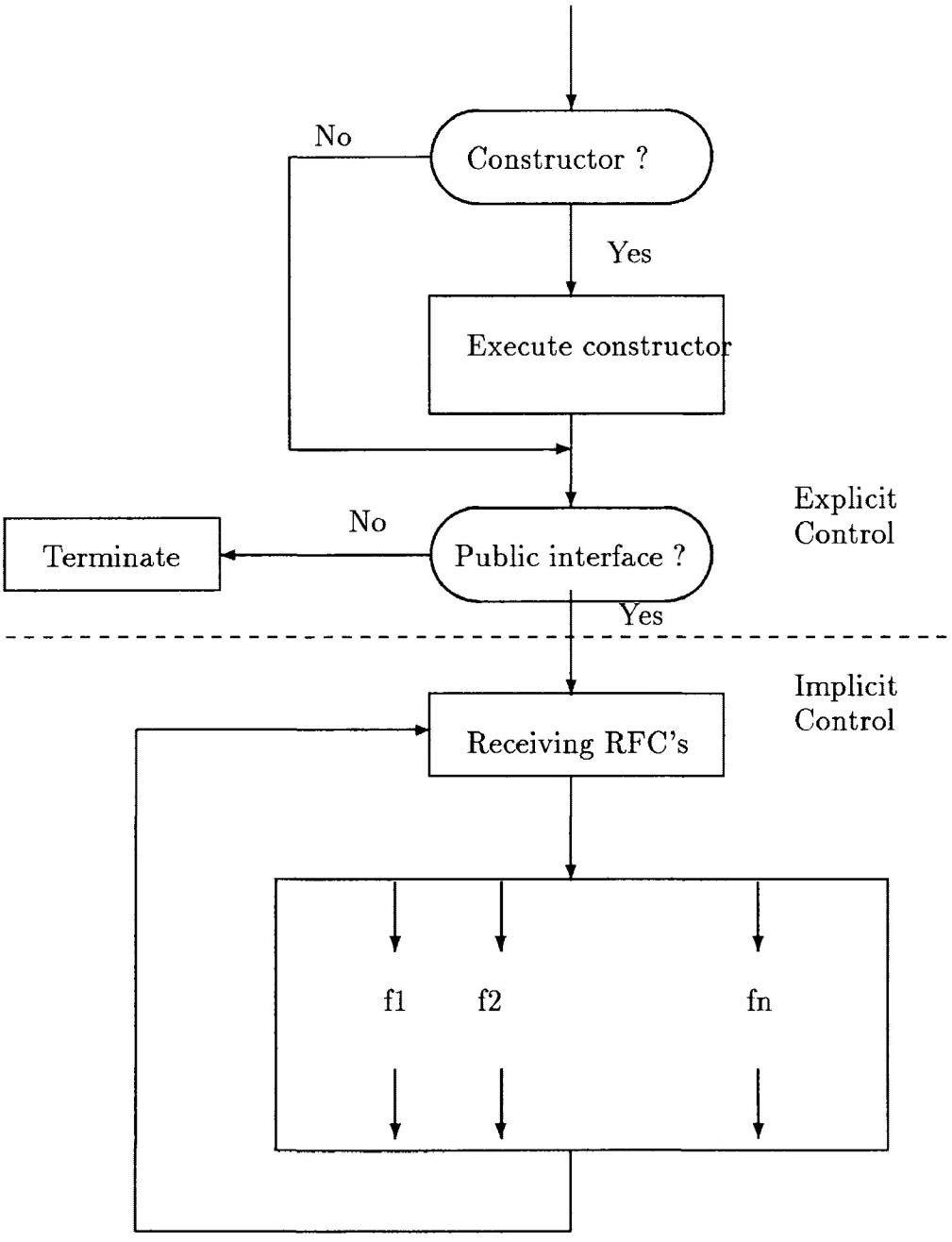

Figure 3.1: Explicit and implicit control of a process object

3. the process terminates when the whole program terminates.

An exception is the root process object which assumes the *main()* function as its constructor. The root process object is destroyed upon termination of the *main()* function. A program in CC++ terminates if all its process objects have been destroyed or if all the remaining process objects are in their implicit control phase and no RFC's are pending or in transmission.

3.2 Remote Function Call

Function call is a well-defined and well-understood mechanism for transfer control and data between class objects. Although this mechanism can not be directly applied to process objects, it is desired that the same mechanism be extended to synchronize process objects and to transfer control and data across process objects. CC++ introduces Remote Function Call to achieve this goal. This mechanism borrows the concept of Remote Procedure Call (RPC) [Han78] and extends it in two significant aspects: (1) An RFC (to a process member function) has either blocked (synchronous) or unblocked (asynchronous) semantics depending on the type of the function declared. (2) A variant of Dijkstra's guard concept [Dij75] is applied to process member functions for deciding the eligibility of incoming RFC's indeterministically.

A principle in designing CC++ is that the syntactic form of RFC should be as close as possible to that of a local function call. A process issues an RFC to another process by referencing the function through the callee's pointer. For example:

```
process A {
  ...
```

```
public:
    ...
    void f(int);
    unblock g();
};


main(){
    A *pa;
    ...
    pa = create A(...);
    pa→f(10);
    pa→g();
}
```

The semantics of an RFC depends on the type of the function being called. If the function has a type of *unblock*, the corresponding RFC is explained by unblocked RFC semantics. Otherwise, the RFC is explained by blocked RFC semantics. Any function declared in a process interface without the *unblock* declaration is of blocked type. In the above example, pa→f(10) is a blocked remote function call since a::f() has the *blocked* type by default. On the other hand, pa→g() is an unblocked RFC.

When a blocked RFC is issued, the function name, arguments and control are transferred to the callee, and the caller is then suspended. The callee executes the desired function on behalf of the caller. The results (if any) and control are passed back to the caller after the function is completed. The caller is then resumed to continue execution as if returning from a local function call. A blocked RFC provides communication and synchronization as well, between participants in the RFC. Such a mechanism has been adopted in many concurrent programming languages. It is suitable for client and server applications. A theoretical flaw of the blocked RFC

paradigm is its lack of recursive semantics - any sequence of blocked RFC's will result in a deadlock if the sequence forms a circular call-chain.

On the other hand, an unblocked RFC only passes the function name and arguments to the callee process and then the caller is free to continue its execution. The caller's behavior after an unblocked RFC must be independent of the RFC's execution, because there is no guarantee that the function has ever been executed. An unblocked RFC can be used as a tool of communication as well as a tool of forking control between participants in the RFC. Unblocked RFC mechanism dilutes the original RPC semantics, but increases the flexibility and the expressive power of the language. It is most suited to process-interactive applications.

Inline functions and data members can not be defined via a process interface. These restrictions arise from the common assumption in most concurrent/distributed systems: no address space is shared among concurrent processes. The practical semantics of accessing inline functions or public data members conflict with this assumption. However, for user's convenience, member functions of a process still can be written like "inline", although they lack the *inline* semantics.

Another restriction concerns the semantics of pointer-containing arguments of process-interface functions. CC++ provides a flat argument-level shared address space emulation for user's convenience. In other words, CC++ defines once-only indirection semantics for pointer arguments but no semantic provision for pointers contained in arguments or multiple-indirection pointer arguments.

The *self* pointer associated with each process object has two major purposes: (1) to simulate an RFC to the process object itself; (2) to be passed to other process objects to establish connections for communication and synchronization. For

example:

```
process B;


process A {
   ...
public:
   unblock f();
   void g(B *pb) {
      ...
      self→f();
      ...
      pb→k();
   }
};


process B {
   A *pa;
   ...
public:
   void h() {
      ...
      pa→g(self);
      ...
   }
   unblock k();
};
```

In the example, an object of process A initiates an RFC to itself by self→f() when its member function g(...) is called. A process object can only issue unblocked RFC's to itself, or dead lock occurs otherwise. On the other hand, the statement pa→g(self) in function h() passes the pointer to a process object of type B to the process object pointed by pa so that a connection is established for further communication and

synchronization.

3.3 Mutual Exclusion and Indeterminism

In concurrent computation, process objects may share common resources and still work in parallel and independently of each other. Thus, the mutual exclusion problem is intrinsic in any concurrent system and must be solved to maintain the consistency and integrity of these shared resources from arbitrary concurrent-access attempts.

CC++ includes a variant of Dijkstra's guard concept for solving mutual exclusion problems. Functions defined in a process (whether in the private part or the public part) can be guarded. A *guard* is a side-effect free boolean expression enclosed in parentheses. A function is guarded if its definition is followed by a colon and a guard. A function without a guard is assumed to have a truth guard value. For example:

```
process buffer{
   int buf[20];
   int count, in, out;
public:
   buffer();
   void put(int i):(count < 20);
   int get():(count);
};
```

Two member functions, put(...) and get(), are guarded to cope with the full/empty situations. In response to an incoming RFC, the implicit control mechanism of a process object executes the corresponding function only if its guard is evaluated to be true. If several functions are eligible (having true guard values) to respond to a set of pending RFC's, then one is selected indeterministically. In the above example,

two functions are eligible to response their RFC's when count is not zero and less than 20.

Guards can not be used in either constructors of processes or member functions of classes. Furthermore, *unblock* functions and guarded functions must be called through RFC's, because local function calls do not offer "forking control" or "conditional call" semantics.

Sometimes a process object can not make any decision to response its caller based upon the current state. The response has to be delayed until some condition is fulfilled. CC++ provides another language feature to handle this situation. This feature is called *forward return*: a blocked member function forwards the responsibility of returning control and results (if any) to another same typed function by using RFC. For example:

```
process A {
   int i;
   ...
public:
   void f():(i>5);
   ...
};


process B {
   A *pa;
   ...
public:
   void g() {
      ...
      forward pa→f();
   }
};
```

The forward statement in g() shifts the return responsibility to the A::f() pointed by pa. The function which issues a forward-return implicitly passes its original caller to the new callee and is then free of duty. The responsibilities of resuming the original caller and returning results are shifted to the new callee as if the new callee is called by the original caller directly. Usually, the shifted function has a guard whose truth value depends on some conditions. In our example, the guard expression is (i>5). The function A::f() is executed when the condition (i>5) is true. The function issuing "forward return" terminates immediately, and its control thread is switched back to the implicit control phase to accept other RFC's. Forward-return is a very important facility in organizing synchronization among process objects.

A process object can also issue a forward-return to itself. This happens when a blocked function has to delay the return to its caller and this delay is to be determined by this process object based upon some other upcoming conditions. For example:

```
process A {
   ...
public:
   int f():(...);
   int g() {
      ...
      forward self→f();
   }
};
```

This programming style is especially useful in process-interactive applications when a process object in the middle of execution wants to receive messages from other partners before making any answer. After the forward-return, the called function g() terminates and the implicit control mechanism of the process object is resumed to

accept other RFC's to change its state. However, the whole transaction of forward-return is transparent to the original caller.

The function which receives a forwarded RFC can further forward it to another process object, but it can not forward back to the original caller in case of causing a circular call-chain. An interesting feature is that a process object which receives a forwarded RFC from some process object can still forward the call to itself. However, this does not imply the recursive semantics. For example,

```
#define N 10

process fac {
   int f;
   int true; // guard
   int computefact(int n){
      if (n == 1) {
         true = 1;
         return f;
      }
      f = f * n;
      forward self→computefact(n-1);
   }
public:
   fac() { f = true = 1; }
   int factorial(int n):(true){
      true = 0;
      forward self→computefact(n);
   }
};

main(){
   fac *p = fac();
   cout << N << "! = " << p→factorial(N) << nl;
}
```

The purpose of this program is to show another usage of the forward mechanism. In process fac, factorial(...) is a guarded member function. Its guard prevents concurrent access by many process objects. In other words, an RFC to factorial(...) is an atomic action, no other RFC's to this function can interfere with its execution. Function computefact(...) is declared in the private section to protect from public access. It carries out the factorial computation by simulating RFC's to itself. However, it is not a recursive function, only the final result will be returned directly to the original caller. Note the syntactical difference between *return* and *forward*. *return* may be followed by a value, an expression, a local function call, a blocked RFC or nothing, but *forward* must be followed by a single blocked RFC.

Chapter 4

APPLICATIONS

Expressive power is one of the most important issues in language design. The expressive power defines the application domain of a language. CC++ has strong expressive power which can be shown in writing various kinds of applications. In general, there are three commonly used concurrent problem solving patterns [Agh90]. They are pipeline, divide-and-conquer and process-coordination.

The pipeline pattern is applied to the problems in which all potential solutions are known. The task of problem solving is to verify the given solutions. The way of computing in this pattern is then to enumerate all the potential solutions and to test them concurrently as they are enumerated. A typical example of this kind of problem is the prime sieve: to find all prime numbers for a given natural number series.

The second pattern is divide and conquer concurrency. The problems of this category can be recursively divided into many subproblems. The result of the overall problem is obtained by joining partial solutions of the subproblems. The solutions of subproblems may be obtained by concurrent computing. There is no interaction (communication) between these sub-computations. An example that can be solved with this method is computing factorial.

The third pattern is cooperative problem-solving. In this pattern, problems are solved by cooperation and interaction of concurrent processes. The interprocess communication and synchronization are extremely important in this kind of problem-

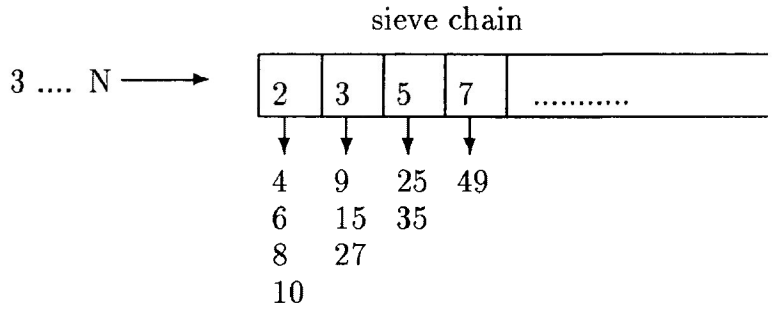

Figure 4.1: Pipeline diagram of prime sieve

solving. Indeterminism is one of the major problems to deal with. Many simulation problems, in which physical objects are represented by logic objects, are applications of this category.

In practical problem solving, however, many problems do not just fall into one of the above categories. They may be entangled with two or even all patterns.

4.1 Prime Sieve

Prime sieve is a good example of showing pipeline concurrency. Prime sieve finds all the prime numbers which are less than or equal to a given N in this problem. The algorithm is to test all numbers through an ordered chain of sieves. The conceptual diagram is shown in Figure 4.1.

In Figure 4.1, a chain is constituted by sieves with each containing a prime number. Initially, there is only one sieve in the chain which contains an integer 2. The successive integers from 3 to N are trying to go through the chain. Each number is tested when it goes into a sieve. If the number is divided by the prime number

in the sieve, the number is discarded. Otherwise, it goes into another sieve. If a number can go through the chain, *i.e.*, no prime number in the chain can divide the number, a new prime number is found. A new sieve is then added to the end of the chain to contain the new prime number. Finally, all the prime numbers are found in the chain. It is easy to use CC++ to write a process sieve to construct a sieve chain for this problem.

```
process sieve {
   int prime;
   sieve *ps;
public:
   sieve(int n) {
      prime = n;
      ps = NULL;
   }
   unblock test(int n) {
      if ((n%prime) == 0);
      else if (ps == NULL)
         ps = create sieve(n);
      else ps→test(n);
   }
};


main ()
{
   sieve *ps = create sieve (2);
   for (int i=3; i<=N; i++) ps→test(i);
}
```

After the first sieve process being created, the root process *main*() feeds the sieve chain with integers from 3 to N for testing. The initial sieve chain contains only one sieve process. A new sieve object will be created and linked to the chain whenever

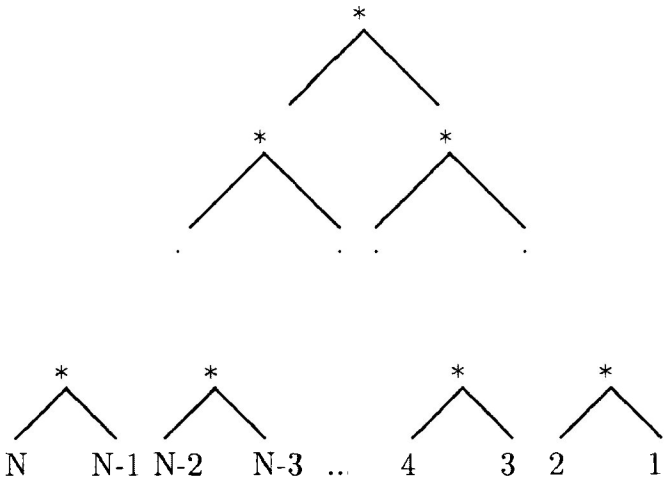

Figure 4.2: A tree structure of $N!$

a new prime is found.

4.2 Computing Factorial

Divide and conquer method can be used to compute factorial concurrently. As is known, $N! = N*(N-1)*(N-2)*...*2*1$. It can be expressed in a tree structure in Figure 4.2.

From this binary tree structure, we can see that the factorial of N is finally divided into a set of successive numbers. The multiplications of every two adjacent numbers in the set constitute a new set. Repeat the computation on every new set until a set containing only one element is obtained. This element is factorial of N. A process type *multiplier* is defined as follows:

```
process multiplier {
   multiplier *pm;
   int i;
```

```
    int partial;
public:
    unblock join(int n) {
        partial *= n;
        // pass result to parent only when two children join
        if (++i == 2){
            if (pm != NULL) pm → join(partial);
            else cout << partial << nl;
        }
    }
    multiplier (int low, int high, multiplier *p) {
        pm = p;
        partial = 1;
        i = 0;
        if (low >= high) pm → join(low);
        else {
            int mid = (low + high) / 2;
            create multiplier(low, mid, self);
            create multiplier(mid+1, high, self);
        }
    }
};


main ()
{
    create multiplier(1, 20, NULL);
}
```

The execution of the program consists of two steps. First, *multiplier* processes are spawned to form a binary tree. Then these processes start to join the results in a bottom-up manner. If a process is a leaf node, it stops creating new processes and passes the result to its parent immediately in its constructor. On the other hand, if a process is an intermediate node, then after its children being created, it goes into the implicit control waiting for RPC's. It will send the partial result back to its

parent only if it has received two RFC's from its children.

4.3 Bounded Buffer

This is a classical concurrent problem which tries to synchronize a set of producers and consumers to access a bounded buffer. A bounded buffer has limited slots to hold elements. For the sake of simplicity, we suppose that the buffer may hold a limited number of integers. Because of the fixed size of the integer buffer, a producer will occasionally find the buffer full, which means that it must wait until a consumer empties a buffer slot. Similarly, a consumer might wait for a producer to deposit data to an empty buffer. Therefore, guarded functions are used to handle these two situations.

```
process buffer {
   int buf[10], count, in, out;
public:
   buffer() { count = in = out = 0; }
   void put(int i) : (count < 10) {
      buf[in] = i;
      in = (in + 1) % 10;
      count++;
   }
   int get(): (count > 0) {
      int i = buf[out];
      out = (out + 1) % 10;
      count--;
      return i;
   }
};


process producer {
public:
```

```
    producer(buffer *pb) {
        ...
        for (int i=1; i<=50; i++) pb→ put(i);
        ...
    }
};


process consumer {
public:
    consumer(buffer *pb) {
        ...
        for (int i=1; i<=50; i++) cout << pb→ get();
        ...
    }
};
```

The interface of the buffer process consists of two guarded functions: *put()* and *get()*. The variable count is initialized to zero by constructor. It is increased each time an element is deposited, and is decreased when an element is taken away. Therefore, comparisons of boundary conditions against count become the guards to provent processes from putting an element to a full buffer or extracting an element from an empty buffer. We define the type of function *put(...)* to be void. However, we can also use unblock type for that function. The difference is that when void type is used, a producer is suspended until its RFC to *put(...)* is responsed. Using unblock type, however, the producer continues its execution without care of when its RFC will be executed. A potential problem of using unblock type in this application is that the system mail queue might overflow.

4.4 Readers and Writers

Readers and writers problem is a classical example of resources management in concurrent programming. Two types of process objects, readers and writers can access a shared file. The file is allowed to be read by many readers simultaneously, but to be written by a single writer at a time when no reader is reading. This example ingeniously uses guarded function to solve the mutual exclusion of a shared file among readers and writers.

```
process filemanager {
   int w, r;
   void delay() : (!r) { }
public:
   filemanager() { w = r = 0; }
   void start_read() : (!w) { r++; }
   void start_write() : (!w) { w++; forward self→ delay(); }
   unblock stop_read() { r--; }
   unblock stop_write() { w--; }
};


process reader {
public:
   reader(filemanager *pf) {
      pf→start_read();
      // reading ...


      pf→stop_read();
      ...
   }
};


process writer {
public:
```

```
    writer(filemanager *pf) {
      pf→start_write();
      // writing ...

      pf→stop_write();
      ...
    }
};
```

The transaction of reading or writing must be enclosed in a pair of RFC's. As long as no writer requests writing, readers can always start reading (through *start_read()*) without delay. When a writer's request is accepted (through *start_write()*), any other requests are blocked. The writer accepted by the function *start_write()* is not allowed to write immediately. Instead, its request is forwarded to function *delay()* which is guarded by the number of current readers. If no reader is reading, the writer is granted permission to write, otherwise, the writer has to wait until all readers terminate their reading transactions. As soon as the writer finishes writing, a *stop_write()* is executed, and blocked readers and writers can start their competitions for accessing the file. This solution maximizes concurrent access to a shared data without starving either writers or readers.

4.5 Shortest Job Next Scheduler

Shortest job next (SJN) scheduler schedules multiple processes sharing a single resource in the order of shortest job first. This example shows how to sequencing RFC's from multiple process objects in the way that priority is granted to the process which will hold the shared resource for the shortest time.

```
process SJN_scheduler {
   Timequeue *tq;
   int free, tm;
   void delay(int time): (free && tm == time) {
      free = 0;
      return;
   }
public:
   SJN_scheduler() { free = 1; tm = 0; }
   void request(int time) {
      if (free && tm == 0 ){
         free = 0;
         return;
      }
      tq→insert(time);
      forward self→delay(time);
   }
   unblock release(){free = 1; tm = tq→first();}
};
```

A process object requests the resource by an RFC to *request()*. A parameter passed to this function is the time interval of how long the process will use the resource. A process is granted to use the resource if no other process is holding or waiting for the resource. Otherwise, its request is inserted into a queue in ascending order of the time. A process issues an RFC to the unblocked function *release()* to return the resource. Thus a delayed job with shortest time is removed from the queue and takes possession of the resource. In the process declaration, Timequeue is a class which manipulates an integer queue. We suppose that its member function *first()* removes the first element from the queue and returns the stored value; however, if the queue is empty, the returned value is 0.

4.6 Game of Life

This example demonstrates the simulation of the game of life [Gar70]. A grid of cells is postulated consisting of living cells and vacant cells initially. The living state of a cell may change in each generation, depending on the states of its eight neighboring cells in the last generation. If a living cell has two or three neighbors alive, it will be alive in the next generation, otherwise it will die. An empty cell will become a living cell if it has exactly three neighbors alive in the last generation. The crucial points of the problem are that (1) cells should be synchronized in every generation transition, and (2) in each generation, every cell exchanges state information with its neighbors. Our simulation program defines two process types: *cell* and *display.*

```
process display;

process cell {
        cell *pc[8]; // pointers to 8 neighbors
        display *pd;
        int cx, cy; // cell position coordinates
        int mail; // # of mails received within one generation
        int state; // 1:alive 2:dead 3:empty
        int alive; // # of alive neighbors
        int initialized;

public:
        cell(int, int, int, display*);
        unblock init(cell*, cell*, cell*, cell*, cell*, cell*, cell*, cell*);
        unblock next(int):(initialized || (mail < 7));


};

process display {
        int cells; // # of total cells
        int count;
        int done_draw;
        void resume():(done_draw);
```

```
public:
        display(int);
        void draw(int, int, int):(!done_draw);


};

cell::cell(int s, int x, int y, display *p){
        mail = alive = initialized = 0;
        cx = x; cy = y; pd = p; state = s;

}

unblock cell::init(cell *l, cell *r, cell *u, cell *d,
              cell *ul, cell *ur, cell *dl, cell *dr){
        pc[0] = l; pc[1] = r; pc[2] = u; pc[3] = d;
        pc[4] = ul; pc[5] = ur; pc[6] = dl; pc[7] = dr;
        pd→draw(state, cx, xy);
        for (int i = 0; i < 8; i++) pc[i]→next(state);
        initialized = 1;

}

unblock cell::next(int s){
        if (s == 1) alive ++;
        if (++ mail == 8) {
           if ((state == 1) && (alive < 2 || alive > 3)) state = 2;
           else if ((state == 3) && (alive == 3)) state = 1;
           pd→draw(state, cx, cy);
           mail = 0;
           alive = 0;
           for (int i = 0; i < 8; i++) pc[i]→next(state);
        }

}

display::display(int c){
        cells = c;
        ce = done_draw = 0;

}
```

```
void display::draw(int s, int x, int y){
        if (s == 1) printxy('*', x, y);
        else if (s == 2) printxy('o', x, y);
        else printxy(' ', x, y);
        if (++count == cells) done_draw = 1;
        return self→resume();

}

void display::resume(){
        if (--count == 0) done_draw = 0;

}
```

The *main()* (in absence) will create a *display* process object and a collection of *cell* process objects and establish the connections among *cells* by using the unblocked *init(...)* function. The behavior of *display* looks like a scheduler to synchronize *cells* from generation to generation. Such synchronization is implemented by guarded functions, *draw(...)* and *resume()*, plus the forward mechanism: *cells* start to decide their next generation states only when their current generation states have been drawn. On the other hand, *cells* use the unblocked function *next(...)* to exchange their states. The guard of *next()* is true if a process object has been initialized or less than 7 *next()* RFC's have been accepted before initialization. The latter condition is used to prevent the process object from entering the second generation without finishing the initial one. The unblock type of *next()* makes it possible to achieve maximum concurrency for *cells* in communication.

Chapter 5

IMPLEMENTATION ISSUES

An experimental CC++ has been developed on a network of SUN workstations. This version consists of a preprocessor and a run-time system. The preprocessor translates a CC++ program into a C++ counterpart. The run-time system includes a process scheduler based on context-switch technique and an Inter-Process Communication (IPC) kernel. In this chapter, we first discuss the execution model of CC++, then briefly describe the implementation essentials of the preprocessor and the run-time system.

5.1 Execution Model of CC++

In CC++, active objects are concurrently executable entities. An active object is an instance of a user defined abstract data type, and is bounded to a sequential process. Active objects are peers which can operate on and be operated by others.

Each active object contains a single control thread. It starts execution from its constructor and then falls into implicit control phase to iteratively select and execute member functions upon incoming RFC's. The execution of a member function is an atomic transaction. That is, as soon as an RFC is accepted the relevant function is executed from the beginning to the end. The atomic transaction ensures mutual exclusive access to the process state.

From programmer's point of view, active objects are coordinated and synchro-

nized by RFC's. RFC is a high level language feature, it is implemented through message-based interprocess communication. The syntactic form to channel messages in CC++ is by process pointers. As the matter of fact, a process pointer is a pointer to an object of the following type:

```
class cc_pid {
   int network_id;
   int local_id;
public:
   cc_pid();
   cc_pid(const cc_pid &pid);
   int operator == (const cc_pid &pid);
   int operator != (const cc_pid &pid);
   void operator = (const cc_pid &pid);
   ...
};
```

A message is a block of information formatted by the sending process and interpreted by the receiving process. Typical message format and the associated operations in our implementation are shown below. The same message format is used for both issuing an RFC and returning results.

```
class cc_message{
   cc_pid sender;
   cc_pid receiver;
   char function_name[MAX_NAME_SIZE];
   int number_of_arguments;
   char arguments[MAX_ARG_SIZE];
   public:
   cc_message(...);
   ~cc_message();
   // functions for accessing information      };
```

Each message indicates a one-to-one communication connection. The IPC mechanism is used to transmit previously agreed on information between two parties. In other words, the strong type semantics of CC++ force the IPC mechanism to archive type checked transmission of information.

Two layers of communication protocols are adopted in our execution model. Here we assume that the low layer is supported by the underlying operating system which provides a reliable, end-to-end mechanism for transmitting bytes of information among nodes of a network. On the other hand, the IPC kernel constitutes the high layer protocol which supports both synchronous and asynchronous communication. A set of IPC primitives (functions) and a mail queue are associated with each active object. Messages transmitted to a process are buffered in its mail queue for later reception. Figure 5.1 shows a synchronous communication transaction between two active objects.

The major concern of our execution model is how to use C++ codes to implement CC++ semantics. Certainly, one can not find one-to-one mapping of these two language features, especially the features concerning concurrency. However, most C++ implementations provide some kind of co-routine facility, which can be used to simulate concurrent processes. We borrow the same idea to implement CC++ active objects. Each process type in CC++ is converted to a class type derived from a base class cc_process. The base class cc_process is a system-defined class that provides primitive operations for RFC's and swapping control among process objects. The following gives a brief description of the base class.

```
enum cc_operation {
   Create, Creating, New_born, Ready, B_call, Forward
```

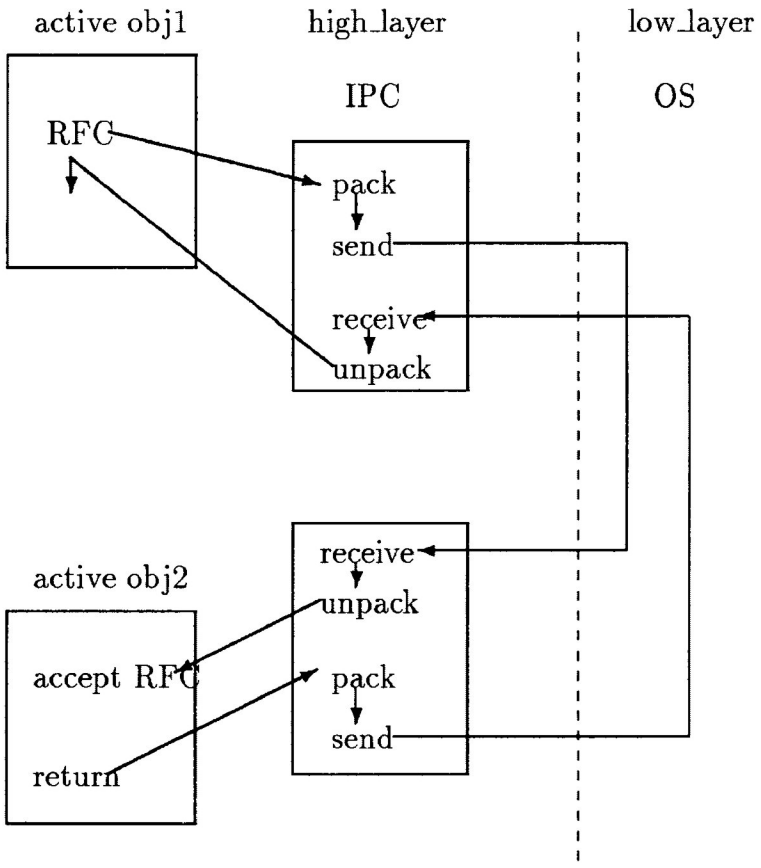

Figure 5.1: A synchronous communication transaction

```
    U_call, Block, Receive, Return, Terminate};

class cc_process: public co_routine {
    cc_pid parent;
    cc_pid child;
    cc_operation operation;
    cc_msg_queue *msg_q;
protected:
    cc_pid cc_self;
    cc_funct *f;
    void cc_swap();
public:
    cc_process();
    ~cc_process();
    char* cc_b_call(...);
    void cc_u_call(...);
    void cc_shift(...);
    cc_pid* cc_create_pobj(...);
    // other functions for accessing process states
};
```

5.2 Preprocessor

A preprocessor is designed to convert a CC++ program to a C++ counterpart. The reason of using a preprocessor in our implementation is that we can adopt the existing C++ compilers which not only make the implementation easier, but also enhance the portability of the language. The preprocessor is written in Lex [Les75] and Yacc [Joh75]. In this section, we describe how to translate typical CC++ features to C++ codes and how those C++ codes match the corresponding CC++ semantics. This discussion places emphases on the major translation procedures to avoid implementation details.

As mentioned in the previous section, a user defined process type is converted to a new class type derived from cc_process. However, as the process interface may involve CC++ features, such as guards, *unblock* functions, inline declarations, *etc.*, the following rules are taken by the preprocessor.

1. A separate guard function is defined for each interface function; if an interface function has no guard in its original definition, then the corresponding guard function simply returns true value.

2. *Unblock* declaration is removed and an UNBLOCK flag will be attached to the function implementation.

3. The specification and the implementation of each inline function is separated to eliminate the inline semantics.

For example, consider the following CC++ definition:

```
process A{
   int guard;
public:
   A(...);
   int f(...):(guard);
   unblock g(...){ ...}
};
```

The preprocessor will generate the C++ counterpart below:

```
class A: public cc_process{
   int guard;
public:
```

```
    A(...);
    int f(...);
    int _f_guard();
    unblock g(...);
    int _g_guard();
};

int A::_f_guard(){ return guard; }

guard_registration _f_guard_regis("f", _f_guard, ...);

int A::_g_guard(){ return 1; }

guard_registration _g_guard_regis("g", _g_guard, ...);
```

The above translation converts process interface functions to normal functions and separates their guards to different guard functions. All these functions are local to the newly derived class. A registration event is associated with each guard function and is maintained by the runtime system. In order to realize the RFC semantics, the preprocessor will generate a stub function as well as a registration event for each process interface function, and such a registration information will also be maintained by the runtime system. For example, the original process interface function f(...) is translated to the following C++ code:

```
int A::f(...){ ... }

_f_stub(...){
    unpack message;
    call f(...);
```

```
    pack result;
    decide process state (Return, ...);
    return;
}

rfc_registration _f_regis("f", _f_stub, ...);
```

The translation procedure of process interface functions only tells one side story of implementing the RFC semantics. For a complete scenario, we have to show the C++ counterpart for RFC invocations. For example, suppose pa is a pointer to a process object of type A, then statement

$$\text{int i} = \text{pa}\rightarrow\text{f(...)};$$

incicates a blocked RFC to function f(...) which will be invoked by the process pointed by pa. The preprocessor translates such an RFC expression directly to the following local function call:

$$\text{int i} = *(\text{int}*)\ \text{cc\_b\_call}(*\text{pa}, \text{"f"}, ...);$$

where cc_b_call(...) is a member function of the base class cc_process and its major algorithm is:

```
char* cc_b_call(...){
    pack the outging message;
    operation = B_call;
    swap to system;
    unpack the returned message;
    return results;
}
```

The final major consideration of the preprocessor is the way of translating a process constructor. Similar to the procedure of translating interface functions, a stub function and a process registration event is generated for each process type. The stub function reflects the two-phase control semantics of a process object and is triggered when the process object is created.

```
A::A(...){ ...}


void _A_ctor(...){
   unpack message;
   call constructor A(...);
   while (has interface functions and not Terminate){
      receive an eligible RFC;
      call the corresponding stub function;
      swap to system;
   }
}


process_registration _A_regis("A", _A_ctor, ...);
```

5.3 Run-Time System

The run-time system consists of a process scheduler based on context-switch and an interprocess communication kernel. The process scheduler creates, schedules and coordinates process objects. The interprocess communication kernel manipulates communications within and between processors.

Communication between process objects are accomplished through synchronous and asynchronous message passing. A set of communication primitives, such as

cc_block_send(), cc_unblock_send() and cc_receive(), are implemented in the IPC kernel. The IPC kernel maintains two FIFO message queues, cc_m_queue1 and cc_m_queue2, for each process object. The first queue is used to hold incoming messages (RFCs). When an RFC request taken from cc_m_queue1 can not be executed because of its false guard, the message is appended to cc_m_queue2, and will be evaluated later. The same strategy is applied to cc_m_queue2 except that the unexecutable RFC's will be appended to itself. When both queues are not empty, then RFC's in cc_m_queue2 have higher priority to be invoked than RFC's in cc_m_queue1.

All process objects are created by the process scheduler. The process scheduler maintains a scheduling queue. Each process object is inserted into the queue upon its creation. The main() function is the root process object created when a program starts execution. It is also the first element in the process queue. Process objects are scheduled in FIFO order. Each process object may be in one of the eleven states: Create, Creating, New_born, Ready, B_call, U_call, Forward, Block, Receive, Return and Terminate. Whether a process object can be put into running or suspending depends on its state. Figure 5.2 shows the possible state transitions of a process object.

A process object starts execution from New_born and ends at Terminate. A process object is eligible to run only if it is Ready. A process object may change its Ready state to another state and then switches its control to the scheduler. On the other hand, changing a process object state from others to Ready is made by the process scheduler. Context-switch happens when a process object interaction occurs. The frequency of context-switch is proportional to that of interactions among process objects. The more the process objects interact, the more frequently the process

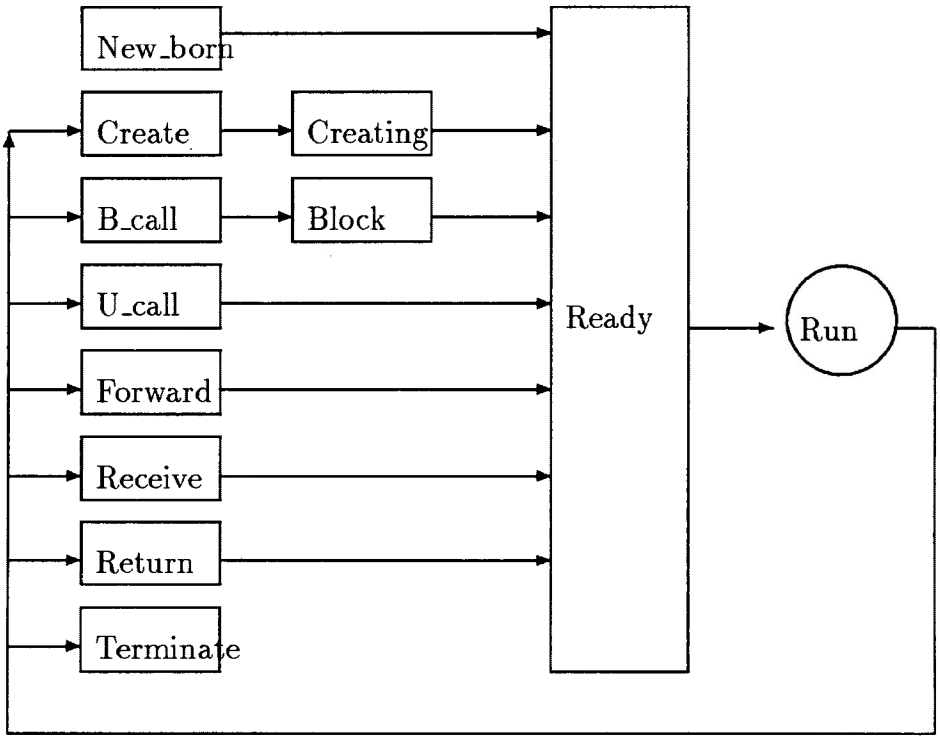

Figure 5.2: Possible state transitions of a process object

objects exchange their control threads.

The advantage of this scheduling strategy is that it guarantees that all process objects involved in interaction have chances to execute in turn, although it does not guarantee the execution order. The order of process object execution is not very important as long as the logic order is correct, which is ensured by guards of accepting RFC's. The disadvantages of this strategy are that the overhead of context-switch is significant if there are frequent unblocked RFC's among process objects and that it is not a fair strategy for computation-bound processes *vs* communication-bound processes.

Chapter 6

Conclusions

Programming in C++ has taught us about sequential object-oriented solutions to problems. However, it is inadequate for solving coordinated applications. Our proposal, CC++, promotes the idea of concurrent computing. We emphasize the consistence of semantics and syntax of the extended language.

The primary characteristics of CC++ are explicit processes, blocked and unblocked RFC's, guarded functions, and forward-return mechanism. The pragmatics of CC++ emphasize ease of program expression: only five new keywords are added to the C++ vocabulary. Synchronization and communication among process objects are achieved by RFC mechanism. Guarded functions provide a concise way of evaluating RFC's either in a mutually-exclusive or indeterministic fashion. Unblocked RFC's maximize the concurrency of process objects when they are involved in communication instead of synchronization. Forward-return mechanism defers the response to a caller by shifting the duty to another function which may be guarded to expect a new state of the process object. Our examples show the simplicity, readability and expressivity of CC++.

To compare CC++ against other languages based on C++ that have been discussed in chapter three, we list the major features of each language in the following table.

	Concurrent C++	uC++	ACT++	CC++
Active object	process	task	actor	process-object
Passive object	class-object	class-object coroutine monitor	class-object	class-object
Active object creation	dynamic	dynamic and static	dynamic	dynamic
Abstract data type	class process	class coroutine	class actor	class process
Architecture	distributed	shared memory	distributed	distributed
Concurrency	inter-process (coarse-grain)	inter-process (coarse-grain)	inter- and intra-process (medium-grain at behavior-level)	inter-process (coarse-grain)
Communication	synchronous & asynchronous msg-passing (transaction calls)	synchronous communication (routine calls)	asynchronous msg-passing (mail deliver)	synchronous & asynchronous msg-passing (blocked and unblocked RFCs)
Synchronization	rendezvous	rendezvous-like	behavior change	guarded function, blocked RFC, and forward return
Inheritance	no inheritance among processes or between process and class	no inheritance among different kinds of types; no multiple inheritance within same types except class types	no multiple inheritance among actor classes	multiple inheritance among same types and from class to process
Polymorphism	no overloaded transactions	overloading allowed but not distinguished in invocation	no overloading in actor class	both virtual & overloading allowed in processes and classes

From this summary, we can see that the most significant differences among the four languages exist in inheritance and polymorphism. For example, Concurrent

C++ does not support inheritance among processes or between process and classes. μC++, which provides five pre-defined data types, only supports single inheritance from the same type. Multiple inheritance or inheritance from different types are not supported. In addition, function overload is not distinguished by a server process although function overload is syntactically allowed. Hence the function overload may lead to unexpected results. ACT++ supports both single and multiple inheritance from normal class to Actor class. However, it does not support function overload in Actor classes. Furthermore, the **become** operation splits the coherent object definition, which somewhat violates the spirit of object-orientation. CC++, however, supports multiple inheritance in process types and from class types to process types, in addition, CC++ supports both virtual functions and function overloading in process types.

A restriction of CC++ is that the language is not suitable for real-time applications because it does not have any mechanism to handle emergency. For instance, a process object in execution can not be interrupted. If a process object sent a message to another executing process object, the earliest response from the receiver is the time after the execution of current function.

Inheritance remains to be a research topic in further work. Currently, CC++ allows inheritance within the same user defined types. It also allows that process types inherit from class types. However, It does not allow class types inherit from process types. This restriction is forced by the following reasons. First, there is a difficulty for C++ compiler to handle a class that is derived from a process type because the process type may involve some new language components such as "unblock", "create" and "forward". Secondly, we have not found a reasonable semantics

explanation of inheritance from process types to class types. For instance, what is the semantics of "unblock", "forward" and "create" when they are inherited by class types? How to get rid of the guards of RFCs in derived classes? In a word, to allow a class inheriting from a process will require a change of syntax and semantics of the original class type in C++.

CC++ is designed to be a practical, usable, concurrent object-oriented programming language. An experimental compiler of CC++ has been developed. The runtime system of CC++ consists of a process scheduler based on context-switch technique and an IPC (Inter-Process Communication) kernel. The first version of CC++ system does not cover all the features discussed in this thesis. An efficient implementation and a detailed experimental study of different concurrent programs in CC++ are our future research directions.

Bibliography

[AdH90] J. K. Annot and P. A. M. den Haan. *POOL and DOOM: The Object Oriented Approach, Collected in Parallel Computers: Object-Oriented, Functional, Logic.* John Wiley & Sons, 1990.

[Agh86] G. Agha. *A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh90] G. Agha. Concurrent object-oriented programming. *Communication of the ACM*, 33(9):125–144, September 1990.

[And83] G. R. Andrews. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[And91a] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[And91b] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–89, March 1991.

[Bae91] D. Baezner. Language design for parallel simulation. *University of Calgary, Master Thesis*, 1991.

[BDSY92] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, and B. M. Younger. uc++: Concurrency in the object-oriented language c++. *Software: Practice & Experience*, 22:137–172, February 1992.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. Technical Report TR86-04-03, Department of Computer Science, University of Washington,Seattle, Washington, 1986.

[Boo91] G. Booch. *Object-Oriented Design With Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[BR89] J. Briot and J. Radeld. Design of a distributed implementation of abcl/1. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 1988, San Diego, CA, SIGPLAN Notices*, volume 24, pages 15–17. ACM, April 1989.

[Bri89] J. Briot. Actalk: a testbed for classifying and designing actor languages in the smalltalk-80 environment. In *Proceedings of European Conference on Object-Oriented Programming(ECOOP'89)*, 1989.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21:262–322, September 1989.

[CC91] R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[Das86] P. Dasgupta. A probe-based monitoring scheme for an object-oriented, distributed operating system. In *ACM Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 57–66. ACM, 1986.

[DC89] W. J. Dally and A. A. Chien. Object-oriented concurrent programming in cst. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 1988, San Diego, CA, SIGPLAN Notices*, volume 24, pages 28–31. ACM, April 1989.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, August 1975.

[DMCB+89] A. Di Maio, C. Cardigno, R. Bayan, C. Destombes, and C. Atkinson. Dragoon: An ada-based object-oriented language for concurrent, real-time, distributed systems. In *Ada-Europe International Conference*, 1989.

[DS89] B. A. Delagi and N. P. Saraiya. Elint in lamina: Application of a concurrent object oriented language. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 1988, San Diego, CA, SIGPLAN Notices*, volume 24, pages 194–196. ACM, April 1989.

[Gar70] M. Gardner. The fantastic combinations of john conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123, October 1970.

[Gel85] D. Gelernter. Generative communication in linda. *ACM Transaction Programming Language Systems*, 7:80–112, January 1985.

[GR86] N. H. Gehani and W. D. Roome. Concurrent c. *Software: Practice & Experience*, 16:821–844, September 1986.

[GR88] N.H. Gehani and W. D. Roome. Concurrent c++: Concurrent programming with class(es). *Software: Practice & Experience*, 18:1157–1177, December 1988.

[HA79] C. Hewitt and R Atkinson. Specification and proof techniques for serializers. *IEEE Transactions on Software Engineering*, SE-5(1), January 1979.

[Han73] B. Hansen. *Operating System Principles.* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[Han78] P. B. Hansen. Distributed processes: A concurrent programming concept. *CACM*, 21(11):934–941, November 1978.

[HC87] J. H. Hur and K. Chon. Overview of a parallel object-oriented language clix. In *European Conference on Object-Oriented Programming Proceedings*, pages 265–273, June 1987.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, pages 323–364, June 1977.

[Hoa74] C. A. R Hoare. Monitors: An operating system structuring concept. *CACM*, 17:549–557, October 1974.

[Joh75] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, Bell Laboratories, 1975.

[KL89a] D. Kafura and K. H. Lee. Act++:building a concurrent c++ with actors. Technical Report TR89-18, Department of Computer Science,

Virginia Polytechnic Institute and State University, 1989.

[KL89b] D. G. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages. In *Proceedings of the 1989 European Conference on Object-Oriented Programming.* The British Computer Society, 1989.

[KM90] T. Korson and J. McGregor. Understanding object-oriented: A unifying paradigm. *CACM*, 33(9):40–60, September 1990.

[Les75] M. E. Lesk. Lex - a lexical analyzer generator. Technical Report 39, Bell Laboratories, 1975.

[LH93] X. Li and H. Huang. On the concurrency of c++. In *The 5th International Conference on Computing and Information*, 1993.

[Li89] X. Li. *CSP* - A Distributed Logic Programming Language for Discrete Event Simulation.* PhD thesis, University of Calgary, 1989.

[LM89] M. G. Lane and J. D. Mooney. *A Practical Approach to Operating Systems.* PWS-KENT Publishing Company, 1989.

[Man89] G. Manning. A peak at acore, an actor core language. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 1988, San Diego, CA, SIGPLAN Notices*, volume 24, pages 84–86. ACM, April 1989.

[Mel89] P Mello. Concurrent objects in a logic programming framework. In *SIGPLAN Notices*, volume 24, pages 37–39. ACM, April 1989.

[MNC+91] G. Masini, A. Napoli, D. Colnet, D. Leonard, and K. Tombre. *Object-Oriented Languages.* Academic Press, 1991.

[Nel91] M. L. Nelson. Concurrency & object-oriented programming. *ACM SIGPLAN Notices*, 26(10):63–72, October 1991.

[NYT+89] T. Nakajima, Y. Yokote, M. Tokoro, S. Ochiai, and T. Nagamatsu. Distributed concurrent smalltalk, a language and system for the interpersonal environment. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 1988, San Diego, CA, SIGPLAN Notices*, volume 24, pages 43–45. ACM, April 1989.

[Pok89] B. P. Pokkunuri. Object-oriented programming. *SIGPLAN Notices*, 24(11):96–101, November 1989.

[Tho89] D. Thomas. What's in an object? *BYTE*, pages 231–240, March 1989.

[Weg89] P. Wegner. Learning the language. *BYTE*, pages 245–253, March 1989.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1:7–87, August 1990.

[YC89] K. Yoshida and T. Chikayama. A'um = stream + relation. In *SIGPLAN Notices*, volume 24, pages 55–58. ACM, April 1989.

[YT87] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* MIT Press, 1987.